Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

Non-iterative SDC modulo scheduling for high-level synthesis

Leandro de Souza Rosa [a,*], Christos-Savvas Bouganis [b], Vanderlei Bonato [a]

[a] Institute of Mathematics and Computer Sciences, The University of São Paulo, Av. Trabalhador São-carlense, 400, São Carlos, Brazil
[b] Department of Electrical and Electronic Engineering, Imperial College London, Exhibition Rd, South Kensington, London SW7 2BU, United Kingdom

ARTICLE INFO

Keywords:
High-level synthesis
Loop pipelining
Scalability
Scheduling

ABSTRACT

High-level synthesis is a powerful tool for increasing productivity in digital hardware design. However, as digital systems become larger and more complex, designers have to consider an increased number of optimizations and directives offered by high-level synthesis tools to control the hardware generation process. One of the most explored optimizations is loop pipelining due to its impact on hardware throughput and resources. Nevertheless, the modulo scheduling algorithms used at resource-constrained loop pipelining are computationally expensive, and their application through the whole design space is often non-viable. Current state-of-the-art approaches rely on solving multiple optimization problems in polynomial time, or on solving one optimization problem in exponential time. This work proposes a novel data-flow-based approach, where exactly two optimization problems of polynomial time complexity are solved, leading to significant reductions on computation time for generating a single loop pipeline. Results indicate that, even for complex loops, the proposed method generates high-quality designs, comparable to the ones produced by existing state-of-the-art methods, achieving a reduction on the design-space exploration time by 2.46× (geomean).

1. Introduction

High-Level Synthesis (HLS) has emerged as a powerful tool for accelerating the hardware design process. HLS allows a high-level capture of the architecture of the intended system through a high-level programming language such as C and Java, enabling software programmers and hardware designers to elaborate digital systems in a fraction of the time required through traditional RTL level tools. In an HLS context, the programmer can explore multiple ways of mapping the source-code into hardware by using appropriate directives. However, this flexibility leads to an exponential increase in the design space dimensions [1], which is often explored partially by the user over many iterations [2]. A common way to handle such large DSE is to use heuristics and statistical models to reduce the number of explored solutions [3].

A direct approach to reducing the DSE time is by reducing the time taken to estimate the impact of the directives in the code. Loop transformations are usually the most critical directives in an HLS scenario since loop acceleration can lead to significant performance improvements and to more efficient hardware resources usage. A key optimization to achieve high throughput is loop pipelining, which is performed by modulo scheduling algorithms, which are responsible for most of the HLS compilation time [4], making the DSE non-viable for large and complex codes.

Towards creating loop pipelines, early approaches used an iterative "greedy" heuristic, which allows the representation on scheduling and hardware design-specific aspects in a single optimization formulation [5]. The introduction of a back-tracking heuristic [4] was used to improve the search, leading to better solutions in reduced execution time.

Opposing to the heuristics, Integer Linear Problems (ILP) have been proposed to completely model the pipelining [6–8]. By encoding all aspects of the problem, ILP formulations eliminate the need for iterative searches and guaranteeing to achieve optimal solutions. However, the expanded formulation has an exponential solving time, which makes its application impractical for large problems or design-space exploration tasks.

Recent works propose to divide the loop pipelining problem in two parts, leaving the allocation part of the problem to be solved by a Genetic Algorithm (GA), while the scheduling part is solved as a standard optimization problem [9], speeding up the process when compared against the heuristic approaches.

In this work, we propose to substitute the GA search by a heuristic to construct a solution, which reduces asymptotically the computation time needed to create a loop pipeline, implying in considerable time savings on design-exploration tasks. This paper extends the initial version of this work [10], with a complete evaluation of the proposed

* Corresponding author.
E-mail addresses: leandrors@alumni.usp.br (L. de Souza Rosa), christos-savvas.bouganis@imperial.ac.uk (C.-S. Bouganis), vbonato@usp.br (V. Bonato).

https://doi.org/10.1016/j.micpro.2021.104334
Received 22 September 2020; Received in revised form 15 April 2021; Accepted 29 August 2021
Available online 11 September 2021

approach, including a full description of the method, along with a broad number of examples, and a detailed result analysis over the scheduler impacts on the final hardware quality.

The key contributions of this work are:

- An heuristic approach to construct a solution for the allocation part of the modulo scheduling problem, aiming to reduce the final design latency, greatly improving the computation time to achieve a solution.
- An investigation on the relationship between the latency and hardware resource usage of the generated design.
- An evaluation of the proposed methodology and state-of-the-art approaches regarding the impact on the generated hardware design's performance and area, and their impact on a design-space exploration context.

The rest of this paper is organized as follows: Section 2 introduces the basic nomenclature and definitions on the topic. Section 4 presents the state-of-the-art modulo schedulers. Section 5 presents the proposed approach. Section 6 compares the proposed approach against state-of-the-art modulo schedulers regarding the computation time and achieved hardware quality. Section 7 summarizes future works, and Section 8 concludes the paper.

2. Background

Loop pipelining improves the system's throughput by creating a hardware structure that allows the initiation of the next loop iteration before the current finishes. A loop structure without pipelining would take $total_{cycles} = tc * l$ to complete its execution, where tc is the loop trip count (number of loop iterations), and l is the loop latency (number of clock cycles to finish one loop iteration). A pipelined loop would take $total_{cycles} = l + II * (tc - 1)$ to finish its execution, where II is the initiation interval (the number of clock cycles between loop iterations).

The problem of designing a hardware structure that would allow a loop pipeline is addressed through modulo scheduler algorithms that find a schedule for the loop instructions given a target II such that no dependency or resource constraint is violated. The smaller the II value is, the fewer the cycles the loop will take for completion. In the general form, a modulo scheduler algorithm is used to find a schedule for an estimated minimum II. In case the scheduler fails, the candidate II is increased, and the modulo scheduler algorithm is performed again. The minimum II estimation is given as $MII = \max(ResMII, RecMII)$, where $ResMII$ and $RecMII$ are the resource-constrained and recurrence minimum II, respectively [4]. This work focuses on the problem of identifying a schedule for a candidate II, and the scheduler algorithm to be iteratively applied by increasing the candidate II.

As input, the modulo schedulers consider the loop source code in its Data-Flow Graph (DFG) form. A DGF is a directed graph $DFG = \{V, E\}$, with vertices $V = \{I_i \mid i = \{1, \dots, n\}\}$ representing the n instructions in the loop body, and edges $E = \{(i, j)\ \forall i, j \mid I_j \text{ depends on } I_i\}$ representing that instruction I_j uses the results of instruction I_i. Hereinafter, we define $n = |V|$ as the `loop size`.

As in [9], we consider the modulo scheduling problem to be separated into two parts. The scheduling part consists in finding the starting time t_i for all instructions $I_i \in V$ such that no dependencies are violated. The allocation part consists in determining a resource instance r_i and congruence class m_i for instructions I_i to be executed. m_i is defined as $m_i = t_i \% II$ (the remainder of the integer division by II) and encodes the starting time of I_i for all iterations of the loop in a single variable.

The scheduling part of the problem can be captured as Formulation 1, in which the constraints (line 3) capture that instruction I_j should start after instruction I_i. D_i is the delay of instruction I_i, which accounts for multi-cycle instructions. $b_{i,j}$ measures the intra-loop dependencies (represented as a back-edge in the loop DFG), where $b_{i,j} > 0$ for back-edges between I_i and I_j, and $b_{i,j} = 0$ for forward edges.

Formulation 1: Scheduling part of the modulo scheduling. All equations are implicitly applied $\forall \{i, j\} \in [1, \dots, n]$

(1)	**minimize:**	$\sum_i t_i$	
(2)	**subject to:**		
(3)	$t_i - t_j$		$\leq -D_i + b_{i,j} II$

Formulation 1 has a Totally Uni-Modular (TUM) matrix of constraints, allowing it to be solvable in polynomial time with respect to the `problem size` [11], which is defined as the total number of decision variables and constraints.

The allocation information is represented as a Modulo Reservation Table (MRT), with II rows and $\sum_k a_k$ columns, where a_k is the instance number of resource type k. Each MRT row and column represent a congruence class m and a resource instance r, respectively. Finding an allocation means to find a unique pair (m_i, r_i) for each instruction in $I_i \in V$.

For the rest of this paper, we define an MRT as "valid" if each entry is assigned to one instruction at maximum. A "conflict" is defined as when two instructions are assigned to the same MRT entry. We will use "MRT" and "allocation" interchangeably hereinafter.

3. Related works

Recent works focus on improving the loop pipeline indirectly, by transforming the source code to obtain better pipelines later on in the compilation flow. [12] introduces the Prioritized Code Motion (PCM) heuristic that modifies the source code to improve the loop pipelining. [13] focus on optimizing loop pipeline to enable the pipelining of nested loops. [14] splits loops in smaller loops with fewer dependencies which can be more easily pipelined. Such works are complementary and can be applied on top of modulo schedulers to further improve the results.

The CCC HLS tool [15,16] particularly compiles C code into ADA code, which is then translated into hardware by its back-end. All code optimizations are applied at front-end level, e.g. code motion, expression simplification, loop unrolling, and loop pipelining. In this specific case, the loop pipeline is a non resource-constrained software-based implementation [17], which is a sub-problem of the problem stated in this work.

[18] presents an iterative loop pipelining method which pairs the standard SDC problems with Boolean Satisfiability Problems (SAT) to solve the resource constraints. This work has two contributions. The first is a method to divide the DFG into sub-graphs, which are scheduled independently. The second is the usage of a SAT problem to solve MRT conflicts in the MRT, which substitutes the SDCS backtracking heuristic. As such, the improvements presented here are complementary and can be applied on top of modulo schedulers to further improve the results.

[19] presents an ILP formulation which includes the resource constraints as decisions variables in the problem, resulting in a multi-objective optimization, focusing on minimizing both the latency and number of functional units (nFUs). [8] extends the ILP formulations to find schedules with rational IIs, which greatly improved the final design throughput, however without guaranteeing the rational II optimality. These approaches are generalizations of [6], which is defined to solve the modulo scheduling problem as defined in Section 2.

Next, Section 4 presents in detail the closest approaches to our work, which apply a resource-constrained loop pipelining to generate a schedule.

4. State-of-the-art approaches

This section provides a high-level description of four state-of-the-art modulo schedulers, namely the Iterative SDC Modulo Scheduler (SDCS), the ILP Modulo Scheduler (ILPS), the Genetic Algorithm Modulo Scheduler (GAS), and the Swing Modulo Scheduler (SMS). The first three schedulers cast the modulo scheduling problem as an optimization problem, considering all instructions of the loop simultaneously, while the last is a representative of the list-based family of schedulers, and it is based on a set of heuristics to provide a solution for the modulo scheduling problem.

4.1. Iterative SDC Modulo Scheduler (SDCS)

The System of Difference Constraints (SDC) Modulo Scheduler [4] is considered to be the state-of-the-art scheduler, and it is used by academic [20] and industrial [21] HLS tools. SDCS casts the scheduling problem as presented in Formulation 1, which is called to be in the SDC form since the left-side of the constraints are differences. Such type of formulation will be referred to as an "SDC problem" hereinafter.

Since Formulation 1 does not consider any allocation information, SDCS applies an iterative heuristic approach that solves $\mathcal{O}(n^2)$ SDC problems to find a schedule with a valid MRT, where n is the `loop size`. The heuristic used by SDCS extends the "greedy" search presented by [5] by including a backtracking function during the search, leading to better solutions with less computation time.

To search for a valid MRT, the heuristic modifies the SDC formulation by adding constraints related to the starting times t_i. After solving the modified formulation, the heuristic checks whether the MRT associated with the solution is valid. As such, SDCS searches for a valid MRT indirectly.

4.2. ILP Modulo Scheduler (ILPS)

Instead of searching for a valid MRT, [6] proposes to represent the MRT explicitly in the problem formulation by using overlap variables [22].

Formulation 2 presents a summarized version of the Integer Linear Program (ILP) formulation; The constraint on line (3) is the same dependency constraint on Formulation 1. ϵ_{ij} and μ_{ji} are the overlap variables which ensures the MRT validity through constraints (4) to (10). Constraint (11) links the instructions start time (t_i) with their congruence class using a helper variable (y_i).

Formulation 2: Summarized ILP formulation for modulo scheduling presented in [6]. All equations are implicitly applied $\forall\{i,j\} \in [1,\dots,n]$

$$\begin{array}{lll}
(1) & \textbf{minimize} & \sum_{i\in V} t_i \\
(2) & \textbf{subject to:} & \\
(3) & t_i + D_i & \le t_j + b_{i,j} II \\
(4) & \epsilon_{ij} + \epsilon_{ji} & \le 1 \\
(5) & r_j - r_i - 1 - (\epsilon_{ij} - 1)a_k & \ge 0 \\
(6) & r_j - r_i - \epsilon_{ij} a_k & \le 0 \\
(7) & \mu_{ij} + \mu_{ji} & \le 1 \\
(8) & m_j - m_i - 1 - (\mu_{ij} - 1)II & \ge 0 \\
(9) & m_j - m_i - \mu_{ij} II & \le 0 \\
(10) & \epsilon_{ij} + \epsilon_{ji} + \mu_{ij} + \mu_{ji} & \ge 1 \\
(11) & t_i & = y_i II + m_i
\end{array}$$

Formulation 2 does not have a TUM constraints matrix since constraint (10) has $II \ge 1$ as coefficient for the variable y_i, while TUM matrices can only have coefficients 0 or ± 1 [23]. Hence, the problem is a general ILP with exponential solving time.

With all scheduling and allocation information captured in the problem formulation, ILPS guarantees to find the optimal schedule for a given II, if such a solution exists, guaranteeing both II and latency optimality. However, $\mathcal{O}(n^2)$ overlap variables are created in the formulation, where n is the `loop size`, which can make the approach not practical in many cases, especially considering the exponential ILP solving time. Nevertheless, when optimal solutions are targeted, ILPS is considered the state-of-the-art approach.

To avoid prohibitive time consumption, ILPS constricts the solver with a time budget. If the solver does not find the optimal solution during this time, it returns the current best solution. If no solution was found, the candidate II is increased and the process is repeated.

4.3. Genetic Algorithm Modulo Scheduler (GAS)

The GAS modifies Formulation 2 to explicitly separate the problem into its scheduling and allocation parts [9]. GAS considers valid MRTs as individuals for evolution, and uses the SDC Formulation 3 to calculate a schedule for such individuals.

By considering a valid MRT as an individual, constraints (5) to (11) can be removed from Formulation 2, whose purpose is to guarantee MRT validity. Then, $t_i = y_i II + m_i$ (constraint (12)) is used to substitute t_i in all other constraints, resulting in Formulation 3.

Formulation 3: SDC base problem for scheduling and allocation separation. All equations are implicitly applied $\forall\{i,j\} \in [1,\dots,n]$

$$\begin{array}{lll}
(1) & \textbf{minimize:} & \sum_i y_i * II + m_i \\
(2) & \textbf{subject to:} & \\
(3) & y_i - y_j & \le \left\lfloor \frac{-D_i + b_{i,j} II - (m_i - m_j)}{II} \right\rfloor
\end{array}$$

In contrast to Formulation 1, Formulation 3 incorporates the MRT values m_i and m_j, linking the resulting schedule to a valid MRT.

However, [9] shows that some MRT configurations can make Formulation 3 infeasible. A "feasible" MRT is defined as an MRT for which Formulation 3 is feasible (has at least one solution) for its m_i, m_j, and II values.

To handle infeasible MRTs, [9] proposes a GA evolution that aims to find feasible MRTs while optimizing the schedule latency concurrently. During evolution, GAS solves $\mathcal{O}(n)$ SDC problems, where n is the `loop size`.

GAS computation time can be controlled by the parameters (α, β), which define the GA population size $pop = \alpha n$ and minimum number of generations $gen = \beta n$, where n is the `loop size`. As such, (α, β) control the GAS' trade-off between quality of solution and computation time.

4.4. Swing Modulo Scheduler (SMS)

The Swing Modulo Scheduler [24] is a representative of the list-based schedulers family. This type of schedulers can generally provide fast solutions when compared to other modulo schedulers, as they do not rely on formulating and solving high complexity optimization problems, and do not support incremental and back-tracking searches. However, list-based schedulers often fail to produce valid schedules, where the non-list-based approaches can provide valid schedules [25].

Previous works showed that SMS outperforms the other list-based schedulers in terms of Quality of Results (QoR) [25] and considers SMS to be the state-of-the-art in this category of schedulers.

SMS is based on two heuristics. The first creates an ordered list of the loop instructions, aiming to reduce the topological distance between dependent instructions. To do so, SMS first enumerates and sorts all sub-cycles (ψ) in the loop Data-Flow Graph (DFG) according to their MII_ψ; then, instructions from cycles are added to a list according to its obeying an MII_ψ decreasing order and dependencies between instructions.

The second step utilizes the generated list to find a schedule for each instruction (i.e., evaluates a t_i value for each I_i), obeying the resource and dependency constraints. To create the schedule, an instruction is retrieved from the list and it is scheduled according to its dependencies, respecting the MRT validity.

In one hand, SMS schedules and allocates each instruction individually (hence the "list-based" nomenclature), elaborating only one schedule and allocation. On the other hand, non-list-based schedulers schedule all instructions at the same time by solving an optimization formulation. Hence, SMS is generally faster than the other schedulers, but it often fails to create schedules due to its limited search capabilities.

5. Non-Iterative Modulo Scheduler (NIS)

This paper proposes the Non-Iterative SDC Modulo Scheduler (NIS), which uses a heuristic to generate a valid and feasible MRT by construction, followed by a search for a respective schedule utilizing Formulation 3. As such, the proposed approach avoids to solve several SDC problems to find a valid MRT (in contrast to SDCS), avoids to address simultaneously the scheduling and allocation parts (in contrast to ILPS), avoids evolving several MRTs to find a feasible one (in contrast to GAS), and evaluates several schedules for the constructed MRT (in contrast to SMS) to minimize the resulting schedule latency.

MRT construction is a key element of the proposed algorithm. NIS focuses on two objectives, the MRT feasibility, and the schedule latency reduction, which are analysed in Sections 5.1 and 5.2 respectively. The proposed heuristic and its detailed implementation are presented in Section 5.3.

5.1. Objective 1: MRT feasibility

Let us consider a loop for modulo scheduling, with a target II, and its corresponding DFG. Consider also all elementary cycles ψ in the DFG, which are lists of nodes starting with I_j and ending in I_i such as $\exists\{(I_j \rightarrow I_i; b_{j,i} := 1)\} \in E$ (i.e., the node I_i) is a destination of an DFG back-edge, and the list ends on I_j). Code 1 presents a simple example to illustrate the proposed approach steps. Fig. 1 presents the DFG for Code 1, which contains 3 cycles: $\psi_0 = \{I_1, I_3, I_4, I_6\}$, $\psi_1 = \{I_1, I_2, I_4, I_6\}$, and $\psi_2 = \{I_1, I_5, I_6\}$.

```
for(int i=LB; i<UB; i++){
  I1 = v[i-1]; //mem. - 1 cycle delay
  I2 = I1+2; // add - 1 cycle delay
  I3 = I1*3; // mult - 2 cycles delay
  I4 = I2+I3;
  I5 = I1+4;
  v[i] = I5+I4; //I6
  I7 = I1+5;
  I8 = I7+6;
  I9 = I7*7;
}
```

Code 1: Loop source code example.

As shown in [9], the feasibility of Formulation 1 only depends on the formulation constraints related to the instructions that are part of DFG cycles. The scheduling constraints for the instructions in a cycle $\psi = \{I_\alpha, I_{\alpha+1}, \ldots, I_{\omega-1}, I_\omega\}$, are captured as the set of Inequalities (1).

$$\begin{cases} t_\alpha - t_{\alpha+1} & \le -D_\alpha \\ t_{\alpha+1} - t_{\alpha+2} & \le -D_{\alpha+1} \\ \quad\ldots & \\ t_{\omega-1} - t_\omega & \le -D_{\omega-1} \\ t_\omega - t_\alpha & \le -D_\omega + b_{\omega,\alpha} II \end{cases} \tag{1}$$

Summing up Inequalities (1) leads to Inequality (2).

$$0 \le b_{\omega,\alpha} II - \sum_{i\in\psi} D_i \tag{2}$$

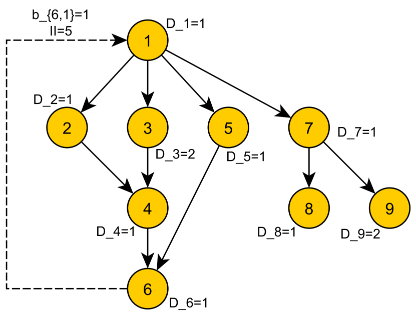


Fig. 1. DFG of Code 1, containing 3 cycles created due to the back-edge (dotted arrow from I_6 to I_1).

We define $l_f^\psi = \sum_{i\in\psi} D_i$ as the forward path length of a cycle ψ, and $l_b^\psi = b_{\omega,\alpha} II$ its back-edge length. The slack of a cycle s_ψ can be defined as Inequality (3).

$$s_\psi = l_b^\psi - l_f^\psi \ge 0 \tag{3}$$

Now, define t_i^0 as the time when all dependencies of I_i are ready. However, there is no guarantee that a resource will be available at $m_i^0 = t_i^0 \% II$. In this case, t_i should be scheduled in another congruence class, creating a delay represented by $\{g_i = t_i - t_i^0 \mid 0 \le g_i \le II - 1\}$. Hence, Inequality (3) is redefined in Inequality (4) as the effective slack, which represents the final schedule slack.

$$es_\psi = l_b^\psi - l_f^\psi - \sum_{i\in\psi} g_i = s_\psi - \sum_{i\in\psi} g_i \ge 0 \tag{4}$$

The condition $es_\psi \ge 0 \mid \forall\psi$ is necessary for Inequalities (1) to be satisfied. Thus, it is also a condition for the MRT to be feasible.

Even though g_i can be only calculated with the final schedule in place, Inequality (4) shows that the larger the slack s_ψ is, the larger $\sum_{i\in\psi} g_i$ can be, without violating $es_\psi \ge 0$. In other words, the larger the slack s_ψ, the more space there is for the instructions of cycle ψ to be "re-arranged" in the MRT without violating $es_\psi \ge 0$.

The proposed heuristic exploits the above observation and sets instructions in DFG cycles with shorter s_ψ to be arranged in the MRT with higher priority. This sorting has similarities with SMS presented in Section 4.4, where the DFG cycles are sorted according to their MII_ψ values. In SMS, a cycle with larger MII_ψ means a longer forward path, which is equivalent to a shorter s_ϕ in NIS. However, using s_ψ instead of MII_ψ allows the algorithm to distinguish between cycles that exhibit the same MII_ψ.

5.2. Objective 2: Latency reduction

As noted in Section 5.1, the position of an instruction in the MRT can add possible delays g_i in the final forward length of a schedule. Furthermore, whenever an instruction I_i is delayed due to MRT conflicts, all instructions I_j that depend on I_i should also be delayed by the same amount in order to match $m_j = m_j^0$.

Thus, we can conclude that the instructions should be arranged in the MRT such that all instruction dependencies are arranged before themselves; i.e., they should be arranged in a "topological order".

NIS applies a topological sorting over the instructions s_ψ-sorted list (elaborated according to Section 5.1). SMS opposes to the proposed approach since it maintains the MII_ψ order (which is the analogous ordering to the proposed s_ψ-sorting) over the topological order.

5.3. Proposed heuristic

The main idea behind the proposed heuristic is to find an ordering for the instructions targeting the objectives presented in Sections 5.1

and 5.2 . The algorithm takes as input the loop's DFG, the constraint set, and the candidate II, and produces an MRT and a schedule associated with it. The proposed heuristic is presented in Algorithm 1.

The first step calculates the DFG's elementary cycles ψ and slacks s_ψ for all back-edges $b_{i,j}$ (line 1) using a DFG recursive deep-search (detailed in Algorithm 2). Cycles are sorted according to their s_ψ values (line 2). Instructions in cycles are parsed into a list that contains a single instance of each instruction (lines 3 and 4), handling instructions that appear in multiple cycles (as I_5, shared by ψ_0 and ψ_2 in Fig. 1) by avoiding repeating their entries in the list. This list is topologically ordered (line 5) and results in a topologically ordered instruction list with slack priority.

For the given example in Fig. 1, the slacks values are $s_{\psi 0} = 0$, $s_{\psi 1} = 1$, and $s_{\psi 2} = 2$. The list without repeated elements is $\{I_1, I_3, I_4, I_6, I_2, I_5\}$, and topologically ordered list is $\{I_1, I_3, I_2, I_4, I_5, I_6\}$,

The second step creates the topological order of all instructions that are not contained in any cycle. The length of instruction I_i is set the maximum length of all forward-paths containing I_i (lines 6 and 7) (note that paths within cycles are naturally excluded from this step). Then, the DFG edges are sorted according to the destination length (lines 8 and 9). The instructions are then parsed into a list in depth-first order (line 10), which is topologically sorted (line 11). The "topological sort" (lines 5 and 11) indicates a "stable" order one among many valid topological sorts of the given DFG. Finally, this list is appended to the ordered list (lines 12).

Following the example in Fig. 1, instructions $\{I_1, I_7, I_8, I_9\}$ have lengths $\{4, 4, 3, 4\}$. Ordering the DFG according to the instructions lengths results in the edge $I_7 \rightarrow I_9$ to be swapped with the edge $I_7 \rightarrow I_8$. The list ordered according to the path length is $\{I_7, I_9, I_8\}$. Finally, the final order is presented in Eq. (5).

$$oList = \{I_1, I_3, I_2, I_4, I_5, I_6, I_7, I_9, I_8\} \tag{5}$$

The third step of the proposed heuristic creates the MRT. An ASAP scheduling is performed by solving Formulation 1, resulting in the base congruence class values $m_i^0 = t_i^{ASAP}\%II$ (line 13). The base m_i^0 value is updated in case one of its predecessors had a conflict (line 15). Non-resource-constrained instructions are allocated within m_0 (lines 16 and 17). Resource-constrained instructions are attempted to be allocated into an MRT entry $m = m_i^0$ (lines 18 to 26). However, in the case the MRT entry is not available, the congruence class is incremented, and a corresponding increment delay is introduced (line 25). Then, the delay increments are recursively passed to all instructions that depend on i (line 26). The schedule is calculated solving Formulation 3 for the resulting MRT (line 27).

Finally, In the case where the resulting MRT is infeasible (line 28), NIS is considered to fail to the candidate II. In this case, the candidate II must be incremented, another MRT elaborated, and another scheduling attempt performed, in the same fashion as all other modulo schedulers.

As presented in Fig. 1 example, a back-edge can belong to multiple DFG cycles with different forward paths in the DFG. Algorithm 2 presents a DFG recursive deep-search function (called $getCyclesAndSlacks$) that returns all cycles for a given back-edge. The function takes as arguments the target II, a current instruction (ci), the final node from which the back-edge is sourced (bi), the partial slack from the paths starting at ci (ps), and the back-edge distance ($b_{i,j}II$). The goal of the function $getCyclesAndSlacks$ is to find all paths from the current instruction ci that lead to the back-edge bi. Note that II, bi, and $b_{i,j}II$ are constants during Algorithm 2 execution for a given back-edge.

Initially, an array of cycles is created to store all cycles related to the back-edge and their respective slack (line 1). Then ci is checked if it has been marked as "not leading" to bi (lines 2 and 3), what can occur if ci belongs to multiple paths. If ci has not been marked as "not in the

Algorithm 1: NIS.

```
    input  : Data-Flow Graph, Constraints, $II$
    output: Valid MRT, schedule
    /* Topological-slack order */
1   $\Psi \leftarrow \bigcup_{b_{i,j}}^{DFG} getCycleAndSlacks(II, j, i, 0, b_{i,j}II)$;
2   sort $\Psi$ increasingly according to $s_\psi$;
3   for $\psi \in \Psi$ do
4       add all instructions $i \in \psi$ to $oList$ if $i \notin oList$;
5   Topologically sort $oList$;
    /* Topological-path length order */
6   for $i \notin \Psi$ do
7       $length_i \leftarrow$ maximum length between all paths containing $i$;
8   for $i \notin \Psi$ do
9       sort $j \in uses(i)$ decreasingly according to $length_j$;
10  Fill $list$ with the Deep-First Search order $\forall i \notin \Psi$;
11  topologically sort $list$;
12  $oList \leftarrow oList + list$;
    /* Filling the MRT with $oList$ */
13  calculate the ASAP schedule;
14  for $i \in oList$ do
15      $m_0 \leftarrow (t_i^{ASAP}\%II + delay_i)\%II$;
16      if i is not resource constrained then
17          MRT$(m_0, 0) \leftarrow i$;
18      else
19          $m \leftarrow m_0$; $inc \leftarrow 0$;
20          while i is not allocated do
21              for $r \in 0$ to $a_k$ | i is type k do
22                  if MRT$(m, r)$ is available then
23                      MRT$(m, r) \leftarrow i$;
24                      set $i$ as allocated;
25              $inc \leftarrow inc + 1$; $m \leftarrow m + 1$;
26          recursively propagate $inc$ to all $i$ successors;
27  Calculate the schedule for the MRT using Formulation 3;
28  return isFeasible(MRT);
```

path", all uses of ci are searched to check if one of them leads to bi (line 6 to 18 loop).

While searching if ci leads to bi, two cases must be considered. The first case is when ci leads directly to bi (lines 6 to 12), then a new cycle is created (line 7), the cycle slack is deduced from the delays of instructions bi and ci (line 8), which are added the cycle (lines 9 and 10). The newly created cycles are added to the return cycles array (created on line 1) The second case is when ci does not lead directly to bi, then, the function $getCycleAndSlacks$ is called recursively on all uses of ci (line 14). Each call will return an array of cycles, to which ci is added (lined 15 and 16). All cycles are added to the return cycles array (created in line 1).

If no path between ci and bi is found, ci is set as "not in the path". This step saves some computation time by avoiding re-visiting sub-paths (lines 19 and 20). In the end, the return cycles array (line 21) is returned to the calling function.

It is worthy to note that the proposed approach applies to any optimization that can be mapped in the SDC form, e.g. instruction chaining [4], without loss of generality.

Table 1 presents examples of MRTs, targeting $II = 5$, created with the proposed approach following the DFG presented in Fig. 1 and the $oList$ in Eq. (5). Table 1(a) corresponds to the ASAP schedule, which is obtained with 4 FUs. Tables 1(a) and Tables 1(b) present the cases with 2 and 3 FUs, respectively, which have some instructions delayed. Note that the instructions within DFG cycles are less likely to suffer delays.

Algorithm 2: *getCycleAndSlacks* function.

```
input  : $ci$, $bi$, $ps$, $b_{i,j}II$
output: An array of $CycleVec$
/* A $CycleVec$ is an array of instructions $I$ and a
   slack values $s_\psi$ */
create $CycleVec$ $retCycles \leftarrow$ new $CycleVec$;
if $ci$ is not in the path then
    return $retCycles$;
define $inodeSlack \leftarrow$ -1;
for each $i \in uses(ci)$ do
    if $i == bi$ then
        create $cycle \leftarrow$ new instructions array;
        $inodeSlack \leftarrow b_{i,j}II - (D_{bi} + D_{ci} + ps)$;
        add $i$ in $cycle$;
        add $ci$ in $cycle$;
        $cycle_{slack} \leftarrow inodeSlack$;
        add $cycle$ to $retCycles$;
    else
        create
        $tempCycles \leftarrow getCycleAndSlacks(i, bi, ps + D_{ci}, b_{i,j}II)$;
        for each $entry \in tempCycles$ do
            add $ci$ to $entry$;
        add all cycles from $tempCycles$ to $retCycles$;
        delete $tempCycles$;
if $retCycles$ is empty then
    set $ci$ as not in the path;
return $retCycles$;
```

Table 1
Examples of MRTs obtained with the proposed method for the DFG in Fig. 1 and different nFUs.

(a) The schedule with 4 FUs is the ASAP.

m	FU_0	FU_1	FU_2	FU_3
0	I_1			
1	I_3	I_2	I_5	I_7
2	–	I_8	I_9	
3	I_4			
4	I_6			

(b) Schedule with 2 FUs. Instructions $\{I_5, I_7, I_8, I_9\}$ are delayed.

m	FU_0	FU_1
0	I_1	I_8
1	I_3	I_2
2	–	I_5
3	I_4	I_7
4	I_6	I_8

(c) Schedule with 3 FUs. Instructions $\{I_7, I_8, I_9\}$ are delayed.

m	FU_0	FU_1	FU_2
0	I_1		
1	I_3	I_2	I_5
2	–	I_7	
3	I_4	I_8	I_9
4	I_6		

6. Performance evaluation

This section presents a comparison between SDCS, ILPS, GAS, SMS, and the proposed NIS methodology in five parts. First, Section 6.1 compares the main characteristics of the modulo schedulers. Second, Section 6.2 compares the achieved quality of the produced schedules and computation time. Third, Section 6.3 elaborates on the relationships between the modulo schedulers quality and final design hardware usage. Fourth, Section 6.4 demonstrates how the modulo schedulers influence the computation time and quality of results when performing a design-space exploration. Fifth, Section 6.5 compares the resulting hardware resources utilization and maximum frequency over increasingly large benchmarks.

The results and analysis presented in this section are the first, to the best of our knowledge, to provide valuable insights between the nFUs definition, the modulo schedulers, and the final hardware quality, complementing and providing explanations for the results observed in [10].

Two sets of benchmarks are used for the evaluation of the modulo schedulers. The first set, presented in Table 2, is composed of synthetic benchmarks designed to test modulo scheduler algorithms in an HLS environment and were used in [4,5,9,10,26], where the constraints are chosen following [4]. Differently from HLS benchmarks, these codes are composed of a single loop, preventing the hardware inferred for the non-loop parts to influence the loop-related results.

The second set is presented in Table 3, which is a subset of the CHStone benchmark set containing traditional HLS applications, and aims to demonstrate the impact of the schedulers on source codes which are not solely composed by loop structures. Note that not all the loops from this benchmark set can be pipelined due to compiler specific restrictions. As such, the benchmark sets provide complementary results to each other.

Table 2
Loops benchmark characterization.

Name	Short name	Loop size	$\frac{\text{\# Operations}}{\text{Constraints}}$ (+/*/%/f+/f*/f%/[])	RecMII/ResMII-I/MII
multipliers	mt	28	$\frac{8/2/0/0/0/0/7}{3/1/x/x/x/1}$	3/4/4
dividers	dv	72	$\frac{12/0/4/0/0/0/11}{3/x/2/x/x/x/2}$	3/6/6
faddtree	fat	81	$\frac{7/0/0/21/0/0/22}{3/x/x/1/x/x/2}$	27/11/27
add int	ai	82	$\frac{0/0/0/21/9/0/12}{x/x/x/3/3/x/2}$	1/12/12
complex	cp	98	$\frac{21/7/2/0/0/0/25}{3/3/1/x/x/x/2}$	20/13/20
adderchain	ac	92	$\frac{48/0/0/0/0/0/24}{3/x/x/x/x/x/2}$	3/12/12

Tables 2 and 3 provide the `loop sizes`, number of operations per type ({+/*/%/f+/f*/f%/[]}, where “f” and “[]” indicates floating point and memory access operations, respectively), resource constraints for each operation type, and minimum II. Results considering other nFUs constraints are presented in Sections 6.3 and 6.4 .

All discussed schedulers were implemented as part of LegUP [20] infrastructure, where the SDCS is natively implemented, as LLVM 3.5 opt passes, using **Gurobi** 7.5 solver [27]. LegUP imposes that all loop bounds and array sizes are statically determined, only the innermost loops can be pipelined, and conditional paths have to be merged into a single basic-block. As a LegUP 4.0 specific restriction, local memories can only be inferred in the main function, which is obtained by fully “inlining” the benchmarks. It should be noted that the above restrictions are due to LegUP infrastructure and not due to the proposed method.

The results were obtained on a computer with Ubuntu 14.04, 16 GB of RAM, and an Intel(R) Core(TM) i7-2600 CPU @ 3.40 GHz. SDCS is set with $INCREMENTAL\_SDC = 1$ and time budget set to 6 (empirically found by [4,28]). ILPS time budget is set to 10 minutes, which represents a significant computation time by being orders-of-magnitude more than the other modulo schedulers require to elaborate a schedule in our tests. The GAS hyper-parameters that control the computation time and results quality trade-off (α, β) were empirically tuned in [9] to the benchmarks in Table 2, avoiding under or over-exploration.

Table 3
CHStone benchmark characterization.

Name	Short name	Loop sizes	# Operations (+/*/%/f+/f*/f%/[])	RecMII/ResMII/MII
adpcm	ad	13, 13, 298	{2/x/x/x/x/x/2}, {2/x/x/x/x/x/2}, {54/x/1/x/x/x/33}	1/1/1, 1/1/1, 2/17/17
aes	ae	13, 96, 25, 90, 25	{2/x/x/x/x/x/2}, {8/x/x/x/x/x/5}, {4/x/x/x/x/x/8}, {5/x/x/x/x/x/12}, {5/x/x/x/x/x/8}	1/1/1, 1/3/3, 1/4/4, 1/6/6, 1/4/4
blowfish	bf	14, 36	{2/x/x/x/x/x/4}, {1/x/x/x/x/x/6}	1/2/2, 1/3/3
gsm	gs	67	{18/x/x/x/x/x/9}	1/5/5
jpeg	jp	13, 9	{2/x/x/x/x/x/3}, {1/x/x/x/x/x/2}	1/2/2, 1/1/1
mips	mi	7	{1/x/x/x/x/x/1}	1/1/1
sha	sh	25, 27, 25, 26	{1/x/x/x/x/x/6}, {1/x/x/x/x/x/6}, {1/x/x/x/x/x/6}, {1/x/x/x/x/x/5}	1/2/2, 1/2/2, 1/2/2, 1/2/2

Table 4
Comparison between the state-of-the-art and proposed schedulers.

	SDCS	ILPS	GAS	SMS	NIS
Type	SDC	ILP	SDC	Heuristic	SDC
Complexity	Poly.	Expo.	Poly.	Lin.	Poly.
Constraints	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	n	N/A	n
# Problems	$\mathcal{O}(n^2)$	1	$\mathcal{O}(n)$	1	2
Optimal II	No	Yes	No	No	No
Optimal latency	No	Yes	No	No	No

To evaluate the different schedulers effects in the produced hardware quality, all designs in the scaling tests have been implemented using Quartus II 16.1, targeting a Stratix V FPGA, with `OPTIMIZATION_TECHNIQUE=speed`, `auto_dsp_recognition off`, `dsp_block_balancing ‘‘logic elements’’`, and `max_balancing_dsp_blocks 0` to remove the usage of hardcore DSPs, avoiding as such its influence in the ALMs and Registers usage during the comparison.

6.1. High-level comparison

Table 4 presents a high-level comparison between SDCS, ILPS, GAS, NIS, and SMS. ILPS has exponential solution time according to `problem size`, and its `problem size` scales quadratically with the `loop size`, n. GAS and NIS `problem size` scale better than the SDCS one, as GAS and NIS do not add allocation-related constraints to their problems as SDCS. SMS is a heuristic that calculates a single solution for the loop-pipelining problem, in contrast to the other methods which are based in optimization formulations.

It is worth to note that even though NIS utilizes an ordered list to create the MRT, it is not classified as a list-based schedule since it considers the scheduling of all instructions concurrently by solving an SDC problem. Next, we highlight concisely the main differences between the modulo scheduler algorithms evaluated in this paper.

SDCS: It is based on SDC formulation problems. However, it does not consider the allocation stage in its formulation, leading to a subset of its solutions to have invalid MRTs. To compensate for that, it implements a heuristic that modifies and solves several SDC problems in an attempt to find a valid MRT.

ILPS: It is based on an ILP formulation problem. It represents formally all the allocation and scheduling information, guaranteeing to find the optimal solution. However, the quadratic `problem size` and exponential solving time makes it not a viable solution for large applications.

GAS: It is based on SDC formulation problems. It includes the allocation stage information in its formulation. The allocation information is represented as valid MRTs. However, the generated MRTs might lead to infeasible schedules. To handle this problem, a Genetic Algorithm (GA) is used to evolve MRTs, aiming to find a valid one while minimizing the schedule latency.

NIS: It is based on an SDC formulation problem. Uses a heuristic to construct an MRT which is valid and has a feasible schedule by construction.

SMS: It uses heuristics to schedule and allocate instructions one at a time. However, the lack of a more elaborated search results in SMS failing to find schedules and allocations often.

Note that a key difference between SMS and NIS is that the first creates an ordering for the instructions to be scheduled, while the second creates an ordering for allocating the instructions in the MRT and then perform the schedule by solving an SDC optimization problem.

6.2. Comparing generated schedules

Tables 5 and 6 present the number of problems solved, achieved II, loop latency l, $total_{cycles}$, and time required to find a schedule for the benchmarks presented in Tables 2 and 3, respectively. The last column on each table is the `geomean` normalized with respect to the SDCS `geomean`. The results for SDCS, ILPS, and NIS are integers since they are deterministic methods, while GAS has a random nature.

On Table 5, SDCS, ILPS, GAS, and NIS were able to find the minimum II for all benchmarks. NIS solves only 2 SDC problems per loop, while SDCS and GAS solve several problems. ILPS solves only one ILP problem per loop per candidate II. SMS fails for all benchmarks apart from **cp**, for which it achieves a larger II than the other approaches.

Further investigation on the failing cases reveals that SMS fails in its second heuristic (i.e., the schedule construction). This happens in the case where back-edge instructions have conflicts in the MRT, forcing their reallocation in a way that violates the back-edge constraints, forcing the algorithm to stop and return a `failure`. The obtained SMS results are not unexpected since list-based schedulers do not explore multiple solutions, making them unable to handle loops with complex data-flows, leading to no solutions or to schedules with larger IIs when compared with methods that explore multiple solutions as shown in [25].

GAS and NIS achieve 20% and 16% worse latency than SDCS, making the total number of cycles for loop completion to be affected by 5% and 1%, respectively. When considering the total computation time, NIS is 100× faster than SDCS due to its constant number of SDC problems solved. GAS is 3.17 times faster than SDCS.

On Table 6 (CHStone benchmark), ILPS, GAS, and NIS were able to find the minimum II for all benchmarks, while SDCS fails to find the minimum II for one loop on benchmark **ad**, resulting in designs with larger latency and increased computation time for the scheduler. SMS fails for 7 out of 19 loops. However, SMS being successful to most loops demonstrates the simpler nature of CHStone benchmarks, motivating the usage of benchmarks on Table 2 to test modulo scheduler algorithms.

Furthermore, GAS achieves 5% increased latencies when compared to SDCS, resulting in 1% increased $total_{cycles}$. NIS achieves 0.4% less latency than SDCS, resulting in 1% fewer cycles for loop completion. Cases where NIS achieves a smaller latency than the other methods are expected given the fact that SDCS, GAS, and SMS are also heuristics and ILPS can easily expire its time-budget leading to sub-optimal solutions.

Table 5
Performance and computation time results (50 repetitions average) for the loops benchmarks (Table 2). “x” values indicate that the modulo scheduler failed to find a schedule.

Name		mt	dv	fat	ai	cp	ac	geo. rate
# problems solved	SDCS	40	201	175	155	380	661	1
	ILPS	1	1	1	1	8	1	0.01
	GAS	17	71	174	62	120	95.64	0.37
	NIS	2	2	2	2	2	2	0.01
	SMS	x	x	x	x	n/a	x	n/a
$Total_{cycles}$ (10^3)	SDCS	0.38	0.31	1.15	0.19	1.81	1.09	1
	ILPS	0.38	0.31	1.15	0.19	1.80	1.08	0.99
	GAS	0.38	0.32	1.17	0.23	1.84	1.10	1.05
	NIS	0.38	0.31	1.17	0.20	1.82	1.10	1.01
	SMS	x	x	x	x	1.82	x	n/a
Initiation interval	SDCS	4	6	27	12	20	12	1
	ILPS	4	6	27	12	20	12	1
	GAS	4	6	27	12	20	12	1
	NIS	4	6	27	12	20	12	1
	SMS	x	x	x	x	21	x	n/a
Latency (cycles)	SDCS	24	72	101	95	127	25	1
	ILPS	23	71	101	91	122	12	0.86
	GAS	26.0	84.0	118.38	138.73	155.29	28.00	1.20
	NIS	28	73	120	102	141	36	1.16
	SMS	x	x	x	x	121	x	n/a
Time (s)	SDCS	1.34	12.35	17.08	18.23	35.61	32.92	1
	ILPS	1.79	$3.42e2$	$1.19e3$	$1.14e2$	$1.17e3$	$1.13e3$	$1.62e3$
	GAS	0.46	2.23	5.51	1.76	5.24	3.36	0.18
	NIS	0.07	0.08	0.07	0.06	0.12	0.11	0.01
	SMS	x	x	x	x	$1e{-3}$	x	n/a

Table 6
Number of problems solved (50 repetitions average) for CHStone benchmarks (Table 3). “x” values indicate that the modulo scheduler failed to find a schedule.

Name		ad	ae	bf	gs	jp	mi	sh	geo. rate
# problems solved	SDCS	10, 10, 10173	10, 33, 27, 45, 27	16, 21	69	12, 8	6	16, 16, 16, 16	1
	ILPS	1, 1, 1	1, 1, 1, 1, 1	1, 1	1	1, 1	1	1, 1, 1, 1	0.04
	GAS	14, 14, 3098.32	14, 266.67, 34, 243.67, 34	14, 63	145.32	14, 10	6	34, 34, 34, 34	1.68
	NIS	2, 2, 2	2, 2, 2, 2, 2	2, 2	2	2, 2	2	2, 2, 2, 2	0.12
	SMS	x, x, x	x, x, x, x, x	x, x	x	x, x	x	x, x, x, x	n/a
$Total_{cycles}$ (10^3)	SDCS	101, 51, 1003	17, 48, 18, 25, 18	1027, 57	762	2, 5209	32	22, 22, 22, 21	1
	ILPS	101, 51, 917	17, 45, 16, 24, 16	1027, 57	762	2, 5209	32	21, 21, 21, 21	0.97
	GAS	101, 51, 940.54	17, 46.34, 18.54, 30, 22	1027, 59	763	2, 5209	32	21, 21, 21, 21	1.01
	NIS	101, 51, 969	17, 47, 16, 30, 16	1027, 57	763	2, 5209	32	21, 21, 21, 21	0.99
	SMS	101, 51, x	17, x, x, x, x	1026, x	x	2, 5209	32	21, 21, 21, 21	n/a
Initiation interval	SDCS	1, 1, 19	1, 3, 4, 6, 4	2, 3	5	2, 1	1	1, 1, 1, 1	1
	ILPS	1, 1, 17	1, 3, 4, 6, 4	2, 3	5	2, 1	1	1, 1, 1, 1	0.99
	GAS	1, 1, 17	1, 3, 4, 6, 4	2, 3	5	2, 1	1	1, 1, 1, 1	0.99
	NIS	1, 1, 17	1, 3, 4, 6, 4	2, 3	5	2, 1	1	1, 1, 1, 1	0.99
	SMS	1, 1, x	1, x, x, x, x	2, x	x	2, 1	1	1, 1, 1, 1	n/a
Latency (cycles)	SDCS	2, 2, 72	2, 39, 6, 7, 6	5, 6	7	4, 3	1	3, 3, 3, 2	1
	ILPS	2, 2, 84	2, 36, 4, 6, 4	5, 6	7	4, 3	1	2, 2, 2, 2	0.89
	GAS	2, 2, 101.23	2, 37.42, 7, 12, 10	5, 8	8.42	4, 3	1	2, 2, 2, 2	1.05
	NIS	2, 2, 136	2, 38, 4, 12, 4	5, 6	8	4, 3	1	2, 2, 2, 2	0.96
	SMS	2, 2, x	2, x, x, x, x	4, x	x	4, 3	1	2, 2, 2, 2	n/a
Time (s)	SDCS	1.31, 1.07, $1.15e^4$	1.3, 16.28, 4.34, 13.45, 4.36	4.05, 4.59	19.94	3.22, 0.78	0.72	4.43, 4.10, 3.57, 2.40	1
	ILPS	2.29, 0.38, $2.34e^6$	0.62, $0.13e^3$, $0.30e^3$, $9.48e^3$, $0.30e^3$	11.44, $1.02e^2$	$4.79e^6$	4.41, 0.26	0.12	1.41, 1.17, 1.02, 0.91	5.39
	GAS	1.43, 1.41, $2.76e^3$	2.25, 64.81, 3.54, 57.61, 5.90	1.67, 11.72	28.73	1.53, 0.88	0.67	5.62, 5.63, 5.48, 5.46	1.25
	NIS	0.45, 0.35, 2.89	0.44, 0.86, 0.39, 0.64, 0.32	0.47, 0.58	0.74	0.50, 0.40	0.43	1.00, 0.71, 0.50, 0.44	0.11
	SMS	0.17, 0.10, x	0.18, x, x,	0.35, x	x	0.45, 0.15	0.16	0.25, 0.20, 0.19, 0.17	n/a

On Table 6 when considering the total computation time, NIS is 9.09× faster than SDCS. The smaller speed gain, when compared to the one on Table 5, is expected since NIS is designed to speed-up larger loops. GAS is 1.25× slower than SDCS, which indicates that its (α, β) parameters tuning are adequate to the benchmarks on Table 3

The results on Tables 5 and 6 show that NIS and GAS have little impact on the $total_{cycles}$, even though their impact on the latency is considerable, which can lead to hardware designs with increased register and logic requirements [29]. Furthermore, the previously reported results are based on specific configurations of the nFUs as presented in Tables 2 and 3. Section 6.3 evaluates the robustness of the modulo

Table 7
Schedulers configurations to obtain schedules with different latencies.

ILP*	Unrestricted time ILPS (optimal solution)
ILP-10	ILPS with 10 minutes time budget
GAS-{0.5,1,2,3}	GAS with $\alpha = \{0.5, 1, 2, 3\}$, respectively
NIS	NIS with the proposed topological order
NIS-DFS	NIS with Deep-First Search topological order

schedulers on their impact on the overall hardware resource usage and $total_{cycles}$ for various configurations of nFUs. Section 6.4 completes this analysis by focusing the evaluation on configurations that are part of the Pareto-front between resource usage and $total_{cycles}$.

6.3. Impact on hardware resources usage

Section 6.2 considers the case of generating a design with a fixed number of Functional Units (nFUs). However, the modulo scheduling processes depend on the nFUs configuration. Increasing the nFUs increases the number of MRT's columns, which tends to reduce the conflicts, making the allocation part of the modulo scheduling problem easier. Furthermore, the number of required resources for sharing decreases, diminishing the necessity of registers and multiplexers, which ultimately mitigate the hardware usage overheads [29].

The investigation of the modulo schedulers over different nFUs configurations indicates their robustness regarding their impact in the hardware resources usage and the $total_{cycles}$ for different nFUs configurations.

First, we analyse the impact of the modulo schedulers on the hardware resource usage. Given a FUs configuration (c), each modulo scheduler will assign an instruction to be computed in a FU and define a certain scheduler for the instructions. Depending on the schedule and FUs assignments, partial results have to be kept in registers for a number of cycles and require multiplexers to route the results between the nFUs. As such the designs generated by each modulo scheduler require different amounts of hardware resources and will achieve a different latency l and II.

Modulo schedulers generally consider optimizing either the hardware resources usage ([5] and SMS) or to optimize the generated l. Recent works have attacked the problem by reducing the loop latency (SDCS, ILPS, GAS, and the proposed NIS). In fact, targeting the resource usage leads to schedules which partial results are stored for less time, indirectly reducing the execution time between instructions. As such, the loop latency is expected to be reduced as the hardware usage is mitigated [29,30].

To demonstrate the correlation between l and hardware resources usage, we selected the benchmark **dv** as an example, whose ALMs and register utilization are strongly affected by the loop latency according to empirical tests. An unroll factor of 2 is used to the original **dv** benchmark to make the problem large enough for the modulo schedulers to return significantly different schedules, as the target latency is varied.

To create schedules with different latencies (l), ILPS, SDCS, GAS and NIS were used with the configurations as described in Table 7. Fig. 2 presents the ALMs (on the left) and the register (on the right) usage, respectively, as a function of the latency for each schedule. The figures demonstrate the relationship between resource usage and achieved latency.

To analyse the robustness of the modulo schedulers regarding variations on the nFUs we evaluate all possible nFUs configurations for the benchmarks in Table 2 as follows: First, we create pipelines for the benchmarks using ILPS, SDCS, GAS, and NIS for each possible nFUs configuration and calculate the area-latency for each solution, which is used as a metric of quality. Second, the area-latency product for SDCS, GAS, and NIS is normalized according to the respective ILPS result, as captured in Eq. (6), where c_i is a configuration of nFUs, and $s \in \{\text{SDCS}, \text{GAS}, \text{NIS}\}$.

$$AL_s^{c_i} = \frac{(ALMs \times total_{cycles})_s^{c_i}}{(ALMs \times total_{cycles})_{ILPS}^{c_i}} \tag{6}$$

Third, Fig. 3 plots the AL values for all configurations of nFUs as histogram distributions for the benchmarks presented in Table 2, which gives information on the general behaviour of the schedulers. Fig. 3 shows that SDCS, GAS, and NIS can lead to designs with an area-latency product up to 4.6× higher when compared to ILPS for a few specific configurations of nFUs, but most cases present an $AL < 1.41$. Note that $AL < 1$ values (Fig. 3(c)) can be achieved in cases which ILPS fails to find the optimal solution given the time budget. Results for **ai** benchmark are not presented since it leads to more than $1e5$ configurations, making it impractical to perform an exhaustive search over the nFUs.

Fig. 3 shows that all schedulers have a peak of AL_{NIS} for similar values of AL, meaning that the typical performance degradation in comparison to ILPS is comparable through all schedulers. Since **mt** is the simplest of the benchmarks tested, it is natural that the AL is generally smaller, making the differences in the distribution more noticeable.

Furthermore, it can be seen that all schedulers can result in designs as $AL > 2$ for all benchmarks apart from **mt**. As such, the results indicate that the 2× hardware resources usage by the designs generated by NIS in the scalability tests (Section 6.5, benchmark **dv**) are not the general trend, and similar results could have been observed for any scheduler depending on the nFUs configuration.

Further investigation shows that this situation occurs when the nFUs is small, leading the ILP solver to fail to find a solution within the 10 minutes time-budget, which makes ILPS increase the candidate II. The larger IIs result in an increased $total_{cycles}$, which leads to a larger AL as compared to the other schedulers.

6.4. Impact on a design space exploration

Sections 6.2 and 6.3 evaluate the modulo schedulers with a fixed nFUs, and over all possible configurations of nFUs, respectively, showing that the results strongly depend on the nFUs definition given by the designer. Hence, the nFUs configurations that lead to higher performance per hardware resource usage, which form the Pareto-fronts of resources and speed, are the most interesting ones from the designer point-of-view. In order to define the nFUs configurations that lead to Pareto-front designs a Design Space Exploration (DSE) is necessary, which is a time-consuming process requiring the evaluation of many different configurations [31,32]. This section complements the evaluation of modulo scheduler approaches by evaluating the impact of the modulo schedulers on the computation time and quality of solutions of a DSE process.

To compare the quality of the design points produced by each modulo scheduler belonging to the Pareto-front, we calculate the Average Distance from the Reference Set (ADRS) [33] as presented in Eq. (7).

The ADRS calculates the average distance of a set of points in Ω to a reference Γ, where Γ set is composed by the Pareto-front designs obtained with ILPS, which is chosen as a reference due to its optimality. Ω set is composed by the Pareto-front designs obtained by the other modulo schedulers. The distance selects only the points that are the closest to each other ($\min_{\omega \in \Omega}$) and considers as the distance between two points the maximum normalized difference between the coordinates of each point in percentage ($\max\{\}$ term).

$$ADRS(\Gamma, \Omega) = \frac{1}{|\Gamma|} \sum_{\gamma \in \Gamma} \min_{\omega \in \Omega} \left[\max \left\{ \frac{a_\omega - a_\gamma}{a_\gamma}, \frac{l_\omega - l_\gamma}{l_\gamma} \right\} \right] \tag{7}$$

Table 8 presents the ADRS for all modulo schedulers, showing that their impact on the quality of designs obtained by the DSE is less than 1%. NIS results in ADRS values in the range between 0.704 and 1.505

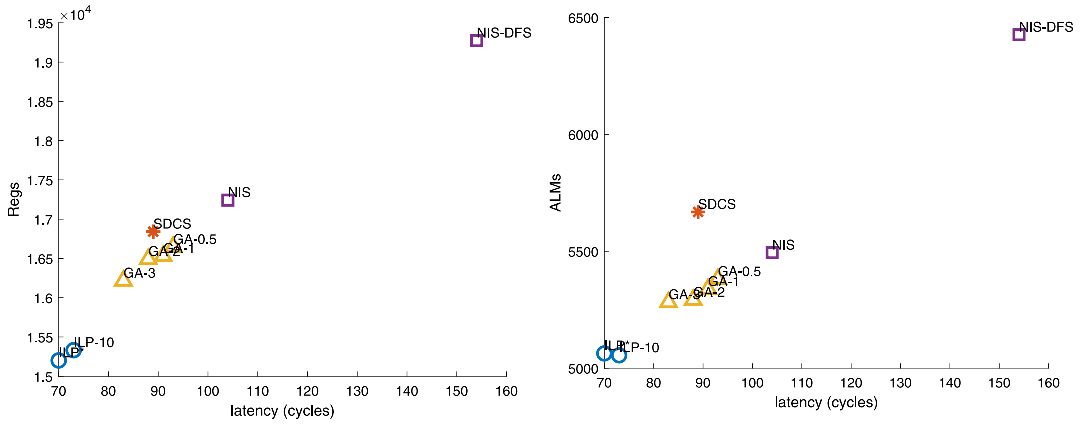
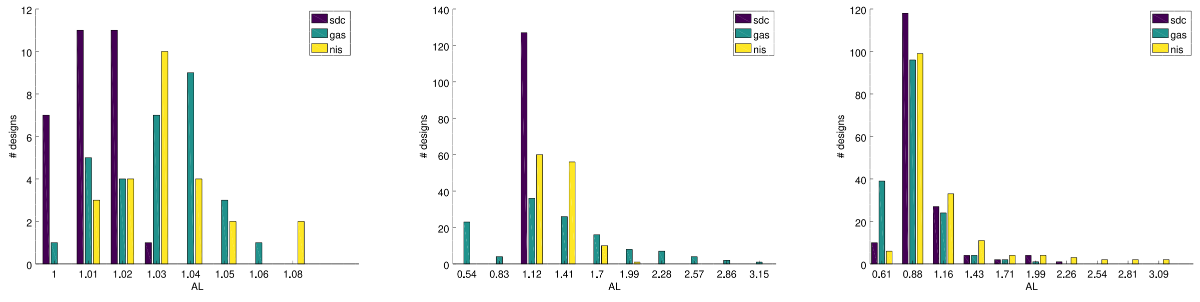
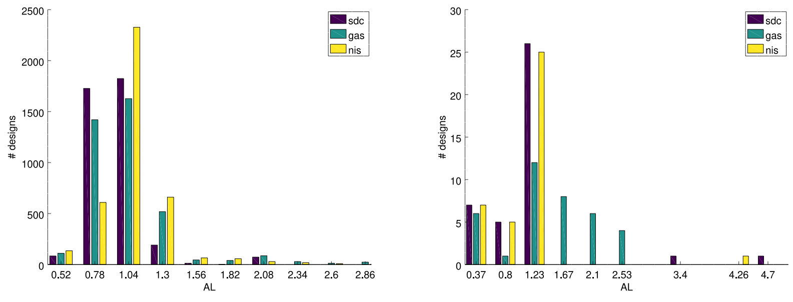


Fig. 2. Registers (on the left) and ALMs (on the right) usage scaling with achieved latency for benchmark **dv**.



(a) **mt**. (b) **dv**. (c) **fat**.



(d) **cp**. (e) **ac**.

Fig. 3. Distribution of the area-latency product for the benchmarks presented from Table ?? considering all possible configurations of nFUs. The AL is normalized according to ILPS.

Table 8
ADRS (%) between ILPS Pareto-optimal solutions and SDCS, GAS, and NIS Pareto-optimal solutions compiled with ILPS.

Benchmark	mt	dv	fat	cp	ac	Average
SDCS	0.000	0.957	0.597	0.322	0.000	0.375
GAS	0.170	0.584	0.308	1.026	1.887	0.795
NIS	1.481	1.505	0.704	0.467	0.804	0.992

Table 9
DSE time (complete HLS and hardware elaboration). Speed-up indicates the geomean speed-up against ILPS.

		mt	dv	fat	cp	ac	Speed-up
Time (hours)	SDCS	1.6	6.9	8.9	156	2.7	2.21
	ILPS	1.9	8.3	26.3	251	20.9	1.00
	GAS	1.6	6.7	8.8	146.5	2.6	2.27
	NIS	1.6	6.6	6.6	137.8	2.5	2.46

for the used benchmarks (as reference, a 1.7% value is considered “extremely low” [3]), demonstrating that the modulo schedulers result in much smaller overheads for designs with optimal nFUs configurations.

Table 9 shows the computation time to evaluate the whole design space. It should be noted that the DSE processes typically consider other directives (e.g.loop unrolling, memory partition), which increase the number of designs that need to be evaluated hence increasing the speed gains in the DSE process. Furthermore, the loop unrolling directive increases the number of instructions in the loop, increasing the NIS speed-ups when compared against ILPS, SDCS, and GAS as demonstrated in Sections Section 6.1, and 6.5.

Fig. 4 presents the ADRS with respect to the DSE speed-up obtained for SDCS, GAS, and NIS, against to ILPS, showing up to 8× speed-up. The obtained speed-up is a function of the loop size given the computation time scaling of NIS is in comparison with SDCS, GAS, and ILPS.

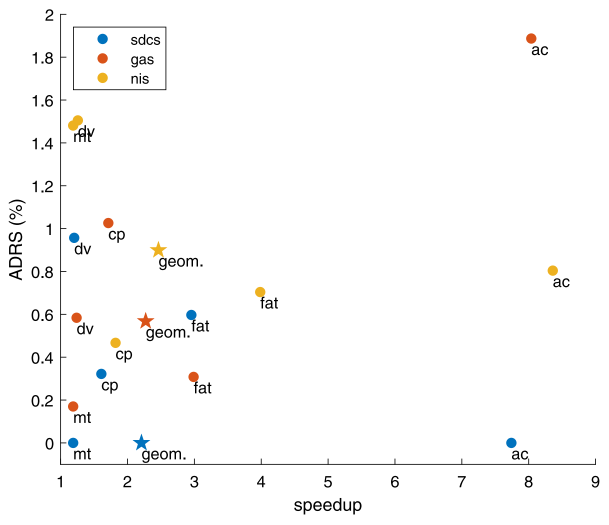


Fig. 4. ADRS and DSE speed-up for SDCS, GAS, and NIS when compared to ILPS.

Table 10
Loop size and computation time (measured by the solver) to find the optimal solution using ILPS for the **mt** benchmark when loop unrolling is applied. ILPS was not able to find the optimal solution for the test using unroll factor 3 with its 36 h budged expiring.

Unroll factor	1	2	3
Loop size	28	50	70
Time for optimal solution	1 s	2.4 h	36 h (expired)

6.5. Scalability

This section investigates the performance of the schedulers concerning their required computation time to produce a solution, the obtained II, the hardware resources, and the maximum frequency of the produced solution as the `loop size` increases. To create large loops, we apply loop unrolling to the benchmarks and resource constraints presented in Table 2 before the loop pipeline creation.

It should be noted that unrolled loops are created by copying their instructions multiple times within the loop body, leading to two patterns in the growth of the DFG according to the unroll factor, making the problem more challenging for the proposed NIS. First, the copied instructions share the same FUs as the original ones, which increases the allocation difficulty since more MRT conflicts need to be solved. Second, the DFG cycles also increase in length, making the conditions for feasibility (Eq. (4)) more difficult to be satisfied since their slack is reduced.

The results presented in this section are the average of 10 repetitions, and the unroll factor is presented besides NIS marks. Missing points mean that the scheduler fails to find a schedule in a 1-hour budget. ILPS results are not presented for this experiment given its poor scalability as demonstrated by Table 10, which presents the computation time to find the optimal solution for the benchmark **mt** when loop unroll is applied. Note that it is common to set a time limit to ILPS in order to avoid such long computation times at the cost of its optimality guarantee.

Fig. 5 presents the time to find a schedule as a function of the `loop size` when `Loop Unrolling` is applied (unrolled `loop size`) before pipelining. The results show that the NIS speed gains increase with the `loop size`, which is expected given the constant number of SDC problems solved.

Fig. 5 also shows that increasing the resource constraints does not impact the computation time for all schedulers strongly. Note that the number of functional units is fixed according to Table 2, implying that the extra required hardware resources (ALMs and registers) are purely overhead caused by extra multiplexes and registers used to route and save partial data.

Fig. 6 presents the achieved II as a function of the `loop size`, showing that NIS and GAS can achieve better IIs than SDCS for benchmarks **mt** and **fat** (base), and **ac** (base). Finding smaller II indicates that NIS and GAS can improve the $total_{cycles}$ when compared to SDC for very large and complex loops.

The missing points in Figs. 5 and 6 show that SDCS fails to produce solutions for larger problems on benchmarks **fat**, **cp**, and **ac**. The results show that GAS and NIS are able to explore solutions more efficiently, indicating the potential of the explicitly MRT exploration approach using Formulation 3 presented in [9,10].

It is important to note that LegUP 4.0 infers (at best) one memory per array in the source code, which has a maximum of 2 ports. Unrolling a loop increases the number of operations that use the same memory, turning memories into bottlenecks since the compiler does not infer more memories or more ports to the existing ones. To avoid this problem, memory buffers [34,35], partition [36], or run-ahead [37] techniques can be applied.

It should be noted that unrolling loops leads to an increase of their number of instructions, and as such, also impacting the optimal nFUs configurations. In order to define a nFUs configuration which leads to a Pareto-front design, a DSE must be performed for each unroll factor in consideration, minding that the design space grows large as the unroll factor increases, making the DSE speed-ups achieved with NIS more significant. If the nFUs are not properly defined, one can expect impacts in the final hardware area, as observed in [10], and as expected according to Sections 6.3 and 6.4.

7. Future work

NIS has shown its capabilities of creating a high-quality schedule faster than the state-of-the-art modulo schedulers. However, we list a few improvements that can be done: An As-Late-As-Possible schedule can be used to reduce the delays in Algorithm 1 (line 30); Improvements in the produced pipeline quality can be achieved thought the implementation of alternative instruction orderings or better MRT construction methods; Modifying the SDC formulation to optimize for register pressure as proposed in [7]; NIS MRT's can be used as a "seeds" for GAS, improving its convergence rate and quality of results. The DFG separation proposed by [18] can benefit NIS and its MRT allocation. Modelling the nFUs as proposed in [19] can help to guarantee the selection of configurations which will not result in designs with a high latency-area product. Finally, the NIS extension to rational-IIs solutions [8] can significantly reduce the $total_{cycles}$ by improving the hardware usage.

8. Conclusions

This paper proposes a heuristic to create a valid and feasible MRT, which is used to calculate a schedule for the loop pipeline using an SDC formulation. The proposed approach solves a constant number of SDC problems, being a massive improvement over the SDC based state-of-the-art approaches, which solve a significantly higher number of optimization problems to find a schedule.

Results show that the increased latency generated by the proposed method has a small impact on the final number of cycles for loop execution, demonstrating that our approach of focusing in to find a feasible and valid MRT was advantageous. The proposed method was also able to find solutions with a smaller IIs than SDCS in our scaling tests. Furthermore, results demonstrate that the proposed approach's impact in the hardware resources usage is minimal for configurations with optimal number of functions units.

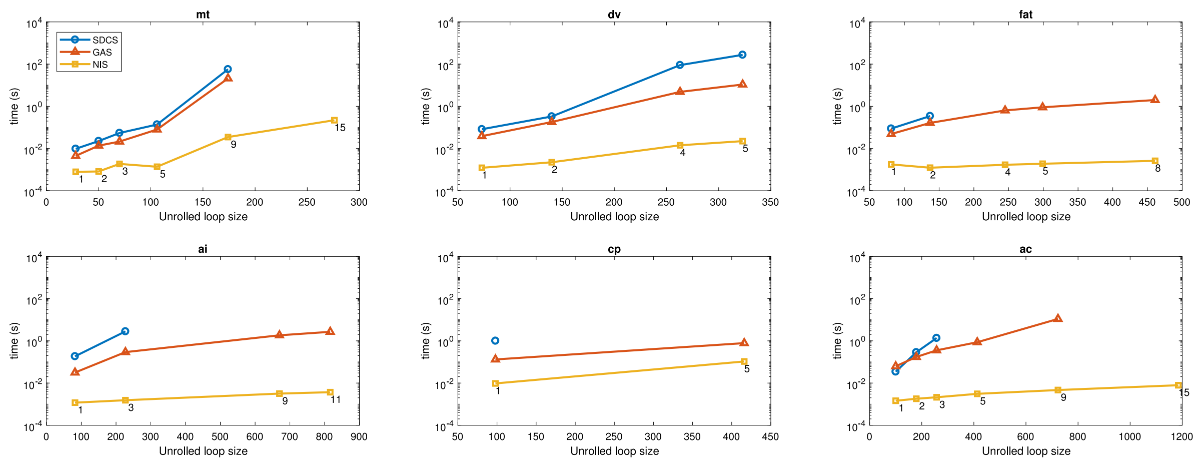


Fig. 5. Time to find a schedule as function of the `loop size` for the benchmarks on Table 2.

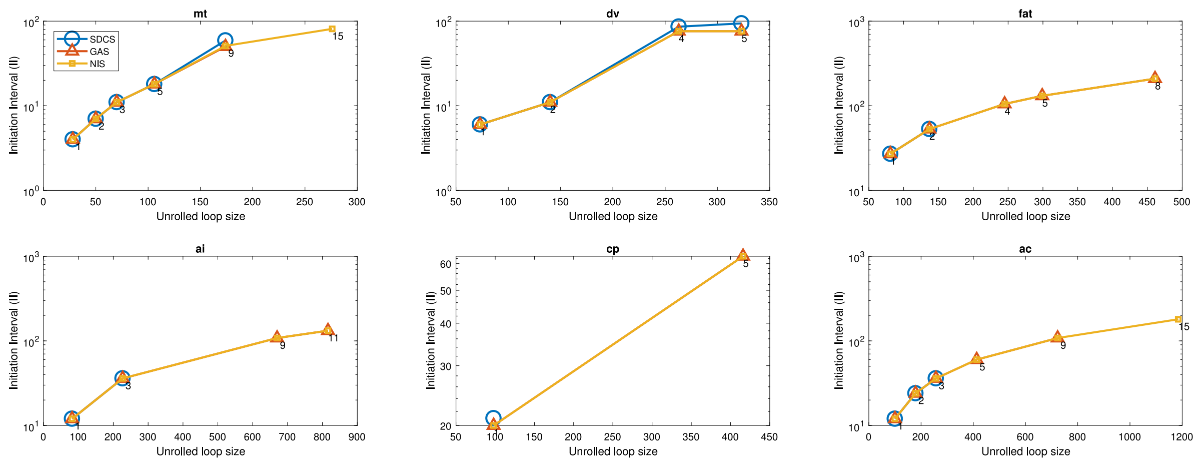


Fig. 6. II obtained as function of the `loop size` for the benchmarks on Table 2.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgement

The authors would like to thank the São Paulo Research Foundation (FAPESP), Brazil for the financial support to this project (grant number 2014/14918-2).

References

[1] B.C. Schafer, K. Wakabayashi, Divide and conquer high-level synthesis design space exploration, ACM Trans. Des. Autom. Electron. Syst. 17 (3) (2012) http://dx.doi.org/10.1145/2209291.2209302, 29:1–29:19.

[2] Z. Wang, B. He, W. Zhang, S. Jiang, A performance analysis framework for optimizing opencl applications on FPGAs, in: 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2016, pp. 114–125, http://dx.doi.org/10.1109/HPCA.2016.7446058.

[3] B.C. Schafer, Probabilistic multiknob high-level synthesis design space exploration acceleration, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 35 (3) (2016) 394–406, http://dx.doi.org/10.1109/TCAD.2015.2472007.

[4] A. Canis, S.D. Brown, J.H. Anderson, 2014. Modulo SDC scheduling with recurrence minimization in high-level synthesis, in: 2014 24th International Conference on Field Programmable Logic and Applications (FPL), 2014, pp. 1–8, doi:10.1109/FPL.2014.6927490.

[5] Z. Zhang, B. Liu, SDC-Based modulo scheduling for pipeline synthesis, in: Proceedings of the International Conference on Computer-Aided Design, in: ICCAD '13, IEEE Press, Piscataway, NJ, USA, 2013, pp. 211–218, http://dl.acm.org/citation.cfm?id=2561828.2561872.

[6] J. Oppermann, A. Koch, M. Reuter-Oppermann, O. Sinnen, ILP-Based modulo scheduling for high-level synthesis, in: Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems, in: CASES '16, ACM, New York, NY, USA, 2016, pp. 1:1–1:10, http://dx.doi.org/10.1145/2968455.2968512.

[7] P. Sittel, M. Kumm, J. Oppermann, K. Möller, P. Zipf, A. Koch, Ilp-based modulo scheduling and binding for register minimization, in: 2018 28th International Conference on Field Programmable Logic and Applications (FPL), IEEE, 2018, p. 2656, http://dx.doi.org/10.1109/FPL.2018.00053, URL https://ieeexplore.ieee.org/document/8533507/.

[8] P. Sittel, J. Wickerson, M. Kuimm, P. Zipf, Modulo scheduling with rational initiation intervals in custom hardware design, in: 2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC), 2020, pp. 568–573.

[9] L. de Souza Rosa, C. Bouganis, V. Bonato, Scaling up modulo scheduling for high-level synthesis, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. (2018) 1, http://dx.doi.org/10.1109/TCAD.2018.2834440.

[10] L. de Souza Rosa, C.S. Bouganis, V. Bonato, Scaling Up Loop Pipelining For High-Level Synthesis: A Non-Iterative Approach, in: The 2018 International Conference on Field-Programmable Technology, 2018, pp. 8.

[11] J. Cong, Z. Zhang, An efficient and versatile scheduling algorithm based on SDC formulation, in: 2006 43rd ACM/IEEE Design Automation Conference, 2006, pp. 433–438, doi:10.1145/1146909.1147025.

[12] M. Dossis, G. Dimitriou, Resolving loop pipelining issues in the CCC high-level synthesis E-cad framework, in: 2018 41st International Conference on Telecommunications and Signal Processing (TSP), IEEE, 2018, pp. 1–4.

[13] A. Morvan, S. Derrien, P. Quinton, Polyhedral bubble insertion: A method to improve nested loop pipelining for high-level synthesis, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 32 (3) (2013) 339–352, http://dx.doi.org/10.1109/TCAD.2012.2228270.

[14] J. Liu, J. Wickerson, G.A. Constantinides, Loop splitting for efficient pipelining in high-level synthesis, in: 2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), 2016, pp. 72–79, http://dx.doi.org/10.1109/FCCM.2016.27.

[15] M.F. Dossis, A formal design framework to generate coprocessors with implementation options, Int. J. Res. Rev. Comput. Sci. 2 (4) (2011) 929–936, URL https://search.proquest.com/docview/903777239?accountid=14643, Copyright - Copyright Kohat University of Science and Technology (KUST) Aug 2011; Last updated - 2012-06-29.

[16] G. Dimitriou, M. Dossis, G. Stamoulis, Minimal-area loop pipelining for high-level synthesis with CCC, in: Design Automation, Computer Engineering, Computer Networks and Social Media Conference (SEEDA-CECNSM), 2017 South Eastern European, IEEE, 2017, pp. 1–8.

[17] G. Dimitriou, M. Dossis, G. Stamoulis, Loop pipelining in high-level synthesis with CCC, in: 2017 6th International Conference on Modern Circuits and Systems Technologies (MOCAST), IEEE, 2017, pp. 1–4, URL https://ieeexplore.ieee.org/document/7937663.

[18] S. Dai, Z. Zhang, Improving scalability of exact modulo scheduling with specialized conflict-driven learning, in: Proceedings of the 56th Annual Design Automation Conference 2019, in: DAC '19, Association for Computing Machinery, New York, NY, USA, 2019, http://dx.doi.org/10.1145/3316781.3317842.

[19] J. Oppermann, P. Sittel, M. Kumm, M. Reuter-Oppermann, A. Koch, O. Sinnen, Design-space exploration with multi-objective resource-aware modulo scheduling, in: R. Yahyapour (Ed.), Euro-Par 2019: Parallel Processing, Springer International Publishing, Cham, 2019, pp. 170–183.

[20] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S.D. Brown, J.H. Anderson, Legup: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems, ACM Trans. Embed. Comput. Syst. 13 (2) (2013) 24:1–24:27, http://dx.doi.org/10.1145/2514740.

[21] T. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, D.P. Singh, From OpenCL to high-performance hardware on FPGAS, in: 22nd International Conference on Field Programmable Logic and Applications (FPL), 2012, pp. 531–534, doi:10.1109/FPL.2012.6339272.

[22] S. Venugopalan, O. Sinnen, ILP Formulations for optimal task scheduling with communication delays on parallel systems, IEEE Trans. Parallel Distrib. Syst. 26 (1) (2015) 142–151, http://dx.doi.org/10.1109/TPDS.2014.2308175.

[23] G.B. Dantzig, R.J. Duffin, K. Fan, A.W. Mather, Linear Inequalities and Related Systems, no.38, Princeton University Press, 1956.

[24] J. Llosa, M. Valero, E. Agyuade, A. Gonzalez, Modulo scheduling with reduced register pressure, IEEE Trans. Comput. 47 (6) (1998) 625–638, http://dx.doi.org/10.1109/12.689643.

[25] J.M. Codina, J. Llosa, A. González, A comparative study of modulo scheduling techniques, in: Proceedings of the 16th International Conference on Supercomputing, in: ICS '02, ACM, New York, NY, USA, 2002, pp. 97–106, http://dx.doi.org/10.1145/514191.514208.

[26] J. Liu, S. Bayliss, G.A. Constantinides, Offline synthesis of online dependence testing: Parametric loop pipelining for HLS, in: 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines, 2015, pp. 159–162, http://dx.doi.org/10.1109/FCCM.2015.31.

[27] Gurobi, Inc.,"Gurobi optimizer reference manual," 2017, 2017, URL: http://Www.Gurobi.Com.

[28] B.R. Rau, Iterative modulo scheduling, Int. J. Parallel Program. 24 (1) (1996) 3–64, http://dx.doi.org/10.1007/BF03356742.

[29] J. Llosa, E. Ayguade, A. Gonzalez, M. Valero, J. Eckhardt, Lifetime-sensitive modulo scheduling in a production environment, IEEE Trans. Comput. 50 (3) (2001) 234–249, http://dx.doi.org/10.1109/12.910814.

[30] S. Dai, G. Liu, R. Zhao, Z. Zhang, Enabling adaptive loop pipelining in high-level synthesis, in: 2017 51st Asilomar Conference on Signals, Systems, and Computers, 2017, pp. 131–135, http://dx.doi.org/10.1109/ACSSC.2017.8335152.

[31] A. Mahapatra, B.C. Schafer, Optimizing RTL to c abstraction methodologies to improve HLS design space exploration, in: 2019 IEEE International Symposium on Circuits and Systems (ISCAS), 2019, pp. 1–5, http://dx.doi.org/10.1109/ISCAS.2019.8702355.

[32] D. Reyes Fernandez de Bulnes, Y. Maldonado, L. Trujillo, Development of multiobjective high-level synthesis for FPGAs, Sci. Program. 2020 (2020).

[33] L. Ferretti, G. Ansaloni, L. Pozzi, Lattice-traversing design space exploration for high level synthesis, in: 2018 IEEE 36th International Conference on Computer Design (ICCD), IEEE, 2018, pp. 210–217, http://dx.doi.org/10.1109/ICCD.2018.00040, URL https://ieeexplore.ieee.org/document/8615690/.

[34] J. Cong, P. Wei, C.H. Yu, P. Zhou, Bandwidth optimization through on-chip memory restructuring for HLS, in: Proceedings of the 54th Annual Design Automation Conference 2017, in: DAC '17, ACM, New York, NY, USA, 2017, pp. 43:1–43:6, http://dx.doi.org/10.1145/3061639.3062208.

[35] J. Cong, Z. Fang, Y. Hao, P. Wei, C.H. Yu, C. Zhang, P. Zhou, Best-effort FPGA programming: A few steps can go a long way, 2018, arXiv:1807.01340.

[36] N.K. Pham, A.K. Singh, A. Kumar, M.M.A. Khin, Exploiting loop-array dependencies to accelerate the design space exploration with high level synthesis, in: Proceedings of the 2015 Design, Automation &#38; Test in Europe Conference &#38; Exhibition, in: DATE '15, EDA Consortium, San Jose, CA, USA, 2015, pp. 157–162, URL http://dl.acm.org/citation.cfm?id=2755753.2755788.

[37] S.T. Fleming, D.B. Thomas, Using runahead execution to hide memory latency in high level synthesis, IEEE, 2017, pp. 109–116, http://dx.doi.org/10.1109/FCCM.2017.33, URL https://ieeexplore.ieee.org/document/7966661/,

Leandro de Souza Rosa has a B.Sc. (2013) in Computer Engineering and obtained the Ph.D. in 2019 by the Institute of Mathematics and Computer Sciences at The University of Sao Paulo (ICMC-USP), Brazil, which was partially developed at the Department of Electrical and Electronic Engineering of Imperial College London. His interest areas are hardware architecture design, high-level synthesis and compilation tools, and code optimization. He is currently working as a Post-Doc. researcher at the Istituto Italiano di Tecnologia (IIT), Genova, Italy, with a focus on event-driven and neuromorphic sensors for robots. He has also served as an ad-hoc reviewer of the journals IEEE-TIM, IEEE-TCADICS, Int. J. of Embedded Systems (IJES), Science China Information Sciences (SCIS), Elsevier MICPRO, and IEEE Sensors Journal.

Christos-Savvas Bouganis is a Reader in Intelligent Digital Systems in the Department of Electrical and Electronic Engineering, Imperial College London, U.K. He is leading the iDSL group at Imperial and he is the Director of Postgraduate Studies in the same department. He has published over 100 research papers in peer-referred journals and international conferences, and he has contributed three book chapters on digital system design. His current research interests include the theory and practice of reconfigurable computing and design automation, mainly targeting the domains of Machine Learning, Computer Vision, and Robotics. He is an Editorial Board Member of the IEEE Transactions on Image Processing, IET Computers and Digital Techniques, Journal of Systems Architecture, and ACM Transactions on Reconfigurable Technology and Systems (TRETS).

Vanderlei Bonato has B.Sc. (2002) and M.Sc. (2004) degrees in Computer Science. He obtained the Ph.D. in 2008 by the Institute of Mathematics and Computer Sciences at The University of Sao Paulo (ICMC-USP), Brazil, which was partially developed at the Department of Electrical and Electronic Engineering of Imperial College London. His interest areas are hardware architecture design, modelling and synthesis tools, and computational finance. He has also several years of industrial experience in automation systems. He is a faculty member of the ICMC-USP since 2009 and became Associate Professor in 2014. Vanderlei Bonato has lately served the committee of several events, including ARC (as general chair), WRC, WSCAD, ERAD-SP, SBESC, FEEC2018, and etc. He has also served as an ad-hoc reviewer of the journals IEEE TCSVT, IEEE TVLSI, IEEE TIP, MICPRO, and JSA and is an Associated Editor of the Int. J. of Embedded Systems (IJES).