Scaling Up Modulo Scheduling For High-Level Synthesis

Leandro de Souza Rosa*, Christos-Savvas Bouganis†, and Vanderlei Bonato*

*Institute of Mathematics and Computer Sciences

The University of São Paulo - São Carlos, Brazil,

Email: {leandrors, vbonato}@usp.br

†Imperial College London - London, England

Email: christos-savvas.bouganis@imperial.ac.uk

Abstract—High-Level Synthesis tools have been increasingly used within the hardware design community to bridge the gap between productivity and the need to design large and complex systems. When targeting heterogeneous systems, where the CPU and the FPGA fabric are both available to perform computations, a design space exploration is usually carried out for deciding which parts of the initial code should be mapped to the FPGA fabric such as the overall system’s performance is enhanced by accelerating its computation via dedicated processors. As the targeted systems become more complex and larger, leading to a large design space exploration, the fast estimative of the possible acceleration that can be obtained by mapping certain functionality into the FPGA fabric is of paramount importance. Loop pipelining, which is responsible for the majority of HLS compilation time, is a key optimization towards achieving high-performance acceleration kernels. A new modulo scheduling algorithm is proposed, which reformulates the classical modulo scheduling problem and leads to a reduced number of integer linear problems solved, resulting in large computational savings. Moreover, the proposed approach has a controlled trade-off between solution quality and computation time. Results show the scalability is improved efficiently from quadratic, for the state-of-the-art method, to linear, for the proposed approach, while the optimized loop suffers a 1% (geomean) increment in the total number of cycles.

Index Terms—Loop Pipeline, Modulo Scheduling, High-Level Synthesis, Genetic Algorithm.

I. INTRODUCTION

Aiming for high productivity and high-performance hardware designs, many practitioners and researchers rely on High-Level Synthesis (HLS) tools for designing their systems. In many cases, the target systems are heterogeneous, where a CPU and an FPGA fabric are available for the computations. Mapping a complex system in such platforms requires a decision on which parts of the application are best suited for each technology, followed by their actual implementation on the corresponding technology. Currently, for the mapping of the application to the FPGA element, modern HLS tools require user provided directives to guide the compilation mapping process, aiming to increase a performance metric. These directives usually

target memory partitioning, loop unrolling, loop tiling, and loop pipelining.

Defining the most appropriated directive settings is a challenging problem, as the optimization of the code that is mapped to the FPGA leads to a combinatorial Design Space Exploration (DSE) problem [1], which is often explored partially by the user over many iterations [2]. Regarding automating the DSE, [3], [1], [4], [5] propose the analysis of many optimization directives, employing heuristics and statistic models to reduce the number of options in the design space, achieving a reduction of $19.5\times$ on average over brute force search [3], [5].

Complementary to the above approaches, a direct way to reduce the DSE total time is by reducing the time taken to apply each optimization in the code. As loops usually are responsible for the bulk of computation in many applications, loop optimizations are considered to have the most impact on performance and as such, loop pipelining is a key optimization to achieve high throughput. However, most of the HLS compilation time is taken by the modulo scheduling algorithms used to create the loop pipelines [6], restricting as such the efficient DSE in systems that contain loops with a large number of instructions.

This work proposes a new modulo scheduling problem formulation that achieves a better asymptotic complexity with the number of instructions in the loop body than the state-of-the-art approaches. Additionally, the new presented methodology allows to control the balance between time available for the problem solution and the achieved quality of results, enabling fast exploration at the initial DSE stages and higher quality designs in the final stages.

This paper is structured as follows: Section II presents the modulo scheduling problem, and Section III presents the state-of-the-art modulo schedulers in literature. Section IV presents the new formulation proposed for the modulo scheduling problem that creates an explicit separation between the scheduling and allocation parts, exploring the last with a Genetic Algorithm (GA) presented in section V. Section VI presents a theoretical comparison and results over benchmarks between the proposed and state-of-the-art schedulers. Finally, Section VII concludes the paper.

II. MODULO SCHEDULING PROBLEM

The modulo scheduling problem is defined as given a piece of code that contains a loop, the aim is to find the minimum possible initiation interval II (i.e after how many clock cycles a new loop iteration can be started), a schedule of the loop instructions, and an allocation into the available resources, without violating any dependencies.

An example of the above problem is captured in Code 1, which also introduces the basic modulo scheduling concepts. The number of loop iterations is captured by the loop trip count tc that is defined as $tc = UB - LB + 1$. Each instruction i has an associated delay D_i representing the number of cycles for its execution.

Code 1: loop code example.

```

1 for(int k = LB; k <= UB; k++){
2   I1 = v1 op1 v2; //delay: 1 cycle
3   I2 = r1 op1 v3; //delay: 1 cycle
4   I3 = r1 op1 v4; //delay: 1 cycle
5 }
```

The loop latency l is defined as the number of clock cycles that one loop iteration takes to complete. The latency l is calculated considering a single loop iteration execution as $l = \max(t_i + D_i)$, where t_i is the cycle where instruction i starts its execution (referred as “starting time”). The loop latency l depends on the number and type of the available resources and on the instructions’ schedule. Table I presents four possible schedules for Code 1, assuming a single type of resource (op1), where I_i^j represents the execution of instruction I_i on iteration j , each row represents a clock cycle and each column a resource instance.

In Table I, the first two schedules are not pipelined. Thus, all instructions in a loop iteration are scheduled to be completed before the next loop iteration starts. The first schedule uses only one resource, resulting in $l = 3$, while the second one uses two resources and has $l = 2$. A loop without pipeline takes $total_{cycles} = tc \times l$ clock cycles to complete its execution.

The last two schedules represent cases where pipelining has been performed and two and three resources are available respectively. Using two resources, I_2^k can start its execution together with I_3^k , thus $II = 2$. Analogously, with three resources, I_1^{k+1} can start its execution together with I_2^k and I_3^k , thus $II = 1$. A pipelined loop finishes an iteration every II cycles, except for the first one that takes l cycles, resulting in $total_{cycles} = l + II \times (tc - 1)$ clock cycles to finish its execution.

TABLE I: Four possible schedule for Code 1.

cycle	no pipeline			pipeline		
	1 res	2 res		2 res	3 res	
0	I_1^0	I_1^0	—	I_1^0	—	I_1^0
1	I_2^0	I_2^0	I_3^0	I_2^0	I_3^0	I_1^1
2	I_3^0	I_1^1	—	I_1^1	—	I_2^1
3	I_1^1	I_2^1	I_3^1	I_2^1	I_3^1	I_1^2
4	I_2^1	I_1^2	—	I_1^2	—	I_2^2
5	I_3^1	I_2^2	I_3^2	I_2^2	I_3^2	I_1^3

Current approaches for solving the modulo scheduling problem, and finding the minimum II that results to the fastest loop execution, rely on first trying to find a solution for an estimating lower bound for II , and in case they fail, to iteratively increase the target II and try again till a solution is found. The lower bound for II is defined as $MII = \max(RecMII, ResMII)$, where $RecMII$ is the recurrence minimum II , and $ResMII$ is the resource-constrained minimum II [6]. Thus, given a candidate II , the problem of minimising $total_{cycles}$ can be expressed as finding a schedule for the target candidate II with minimum latency l , which can be obtained through an As Soon As Possible (ASAP) schedule.

In this paper, we divide the modulo scheduling problem into its scheduling and allocation parts. The scheduling part consists of defining the starting time t_i for all instructions I_i in the loop such that no dependencies are violated. The allocation part consists of choosing which congruence class and resource instance (defined as variables m_i and r_i respectively) instruction I_i will be executed. Note that the congruence class m_i of instruction I_i is defined as $m_i = t_i \% II$. The above information is represented through a Module Reservation Table (MRT), which is a $II \times \sum_{k \in R} a_k$ matrix, where a_k is the instances number of resource type k and R is the set of all resource types.

Through the rest of this paper, we will consider the modulo scheduling problem for a given II , assuming there is an outer loop searching for the possible II , as is the case in all state-of-the-art approaches.

III. STATE-OF-THE-ART MODULO SCHEDULING APPROACHES

A. SDC Modulo Scheduler

Zhang et. al. [7] proposed SDCS, an approach that divides the modulo scheduling problem into its scheduling and allocation parts. The scheduling part includes the dependency, timing, and chaining constraints captured through a System of Difference Constraints (SDC) form. As such, a Totally Uni-Modular (TUM) matrix of constraints can be constructed, whose linear relaxation solution is integral, and thus is optimal for the original problem as well [8]. As such, the original integer-programming problem, with exponential solve complexity, is mapped to its relaxed version that exhibits polynomial solve complexity.

Definition: An MRT (an allocation) is said “valid” if all positions (slots) are assigned to up to one instruction

The allocation part is solved with a greedy heuristic, which starts solving the scheduling problem without considering the resource constraints. If the resulting MRT is not valid, the heuristic adds new SDC constraints to the SDC problem, in an attempt to force the conflicting instructions to start at different times. The process is repeated until there are no conflicts in the MRT, or until a certain number of attempts are performed. If the algorithm fails to find a valid allocation, the candidate II is increased, and the whole process is repeated.

The above iterative heuristic approach [7] is improved in [6], by introducing the possibility of backtracking on

the conflict instructions, allowing them to be sent back to the scheduling queue, significantly improving the solutions latency and also finding solutions for II candidates for which the previous approach would have failed. However, the number of iterations of the method is increased, and consequently the number of SDC problems solved is also increased.

The example below illustrates the number of SDC problems that are required to be solved by this approach in the worst-case. The scheduler (Algorithm 1) and backtracking (Algorithm 2) algorithms are replicated from [6].

Definition: "loop size" is defined as the number of instructions in the Intermediate Representation (IR) code of the loop body. "problem size" is defined as the number of variables and constraints in an optimization problem formulation.

Firstly, in Algorithm 1, the non-resource constrained schedule is calculated once, resulting in the As Soon As Possible (ASAP) schedule (line 1) that serves as the basis for setting the instructions starting time. In lines 2 to 4, the instructions are queued to be processed in order. The number of iterations is constrained by the **budget** (line 3), which is defined as $budget = budgetRatio * n$, where n is the loop size and $budgetRatio$ is a user defined constant. As such, the above limit makes the worst-case number iterations of Algorithm 1 equal to $budget$, regardless on how many instructions are sent back to the scheduling queue by the backtracking policy. In each iteration, the **solver**, i.e. the function responsible for solving the SDC scheduling problem, is invoked in lines 11 and 17 as well as a number of times within the **BackTracking** function.

As illustrated in the **BackTracking** function (Algorithm 2), an instruction I is attempted to be scheduled from its ASAP time to its current scheduled time (lines 1 to 3). In the worst-case, its ASAP time is zero and its current scheduled time is the length of the whole schedule, which, in the worst-case, is a sequential path. Therefore, in the worst-case, the **solver** is invoked $\sum_{k=0}^n D_k$ times, where D_k is the delay of instruction k . Assuming that all instructions have a maximum latency D_{max} , it results in $n \times D_{max}$ **solver** invocations in the backtracking function.

As such, in the worst-case, the total number of **solver** calls is:

$$\begin{aligned} \mathcal{O}(budgetRatio \times n(2 + n \times D_{max})) &= \dots \\ \dots &= \mathcal{O}(budgetRatio \times n^2) \end{aligned} \quad (1)$$

The rest of Algorithm 2 deals with the case where the provided schedule is not feasible. In that case, a time slot is selected (lines 4 to 8) and an instruction from that slot is removed from the SDC problem formulation (lines 9 to 12) and the necessary updates are performed in the corresponding MRT structure (lines 13 to 17). As such, a time slot is available for an instruction to be scheduled (lines 18 and 19), and the control returns to Algorithm 1.

B. ILP Modulo Scheduler

As the objective is to create a schedule for the loop instructions for a given II , the problem can be naturally

Algorithm 1: SDCS algorithm presented in [6]

```

input :  $II$ ,  $budget$ 
output : A schedule for  $II$ 
1 Schedule without resource constraints to get ASAP
  times;
2  $SchedQueue \leftarrow$  all resource constrained
  instructions;
3 while  $schedQueue$  not empty and  $budget \geq 0$  do
4    $I \leftarrow$  pop  $schedQueue$ ;
5    $time \leftarrow$  time of  $I$  from SDC schedule;
6   if scheduling  $I$  at time has no resource conflicts
     then
7     Add SDC constraint:  $t_I = time$ ;
8     Update MRT and  $prevSched$  for  $I$ ;
9   else
10    Constrain SDC with  $GE$  constraint:
       $t_I \geq time + 1$ ;
11    Attempt to solve SDC scheduling problem;
12    if LP solver finds feasible schedule then
13      Add  $I$  to  $schedQueue$ ;
14    else
15      Delete new  $GE$  constraint;
      // See Algorithm 2
16      BackTracking( $I$ ,  $time$ );
17      Solve the SDC scheduling problem;
18     $budget \leftarrow budget - 1$ ;
19 return success if  $schedQueue$  is empty, otherwise
    fail;

```

formulated as a resource-constrained scheduling-allocation Integer Linear Problem (ILP). Additionally to resources, other HLS metrics can be added in the ILP to improve or ensure hardware quality, as timing, chaining [8], and soft [9] constraints.

Instead of separating the problem scheduling and allocation parts, [10] proposes to use the complete formulation, which always allows finding the optimal solution for a given II , if it exists. This formulation requires $\mathcal{O}(n^2)$ variables and constraints to ensure MRT validity, which are called overlap variables [11], where n is the loop size. Moreover, as this approach cannot be expressed in the SDC form, its solver scales exponentially with the problem size. However, there is no need to solve the formulation multiple times as in the SDCS approach but only once, which counter-balances the computation time required for obtaining a single solution. Moreover, the ILP approach is also capable to quickly prove that the problem generated by a candidate II is infeasible, which is a secondary way to counter-balance its exponential complexity. We will refer to it by ILP Scheduler (ILPS) in this work.

Formulation 1 presents a simplified version of the ILP formulation presented in [10], where the variables domain is described in Table II (replicated from [10]).

The objective function (line 2) captures the starting time of all instruction, which needs to be minimized, leading to

Algorithm 2: SDCS BackTracking algorithm in [6]

```

input : Instruction  $I$  and previous  $time$ 
output : MRT with instruction  $I$  scheduled
1 for  $minTime = ASAP\ time\ of\ I$  to  $time$  do
2   SDC schedule with  $I$  at  $minTime$  ignoring
   resources;
3   break if LP solver finds feasible schedule;
4  $prevSched \leftarrow$  previous scheduled time for  $I$ ;
5 if  $no\ prevSched$  or  $minTime \geq prevSched$  then
6    $evictTime \leftarrow minTime$ ;
7 else
8    $evictTime \leftarrow minTime + 1$ ;
9 if  $resource\ conflict\ scheduling\ I\ at\ evictTime$  then
10   $evictInst \leftarrow$  instruction at  $evictTime \% II$  in
   MRT;
11  Remove all SDC constraints for  $evictInst$ ;
12  Remove  $evictInst$  from MRT;
13 if  $dependency\ conflict\ scheduling\ I\ at\ evictTime$ 
   then
14   for  $S$  in already scheduled instructions do
15     Remove all SDC constraints for  $S$ ;
16     Remove  $S$  from MRT;
17     Add  $S$  to  $schedQueue$ ;
18 Add SDC constraint:  $t_I = evictTime$ ;
19 Update MRT and  $prevSched$  for  $I$ ;

```

an ASAP schedule.

For each dependency on the data flow, a constraint is created (line 4), where $b_{ij} \geq 0$ is the loop-carried dependency distance (which is represented as a back-edge in the data-flow graph), and $b_{ij} = 0$ for data dependencies (forward edges in the data flow). Note that [10] considers $b_{ij} \in \{0, 1\}$, while this paper adopts the more general case where $b_{ij} \in \mathbb{Z}_0^+$, as proposed by [6].

Formulation 1: Simplified version of ILP formulation for modulo scheduling presented in [10]

$$\begin{array}{ll}
(1) & \textbf{minimize:} \\
(2) & \sum_{i \in O} t_i \\
(3) & \textbf{subject to:} \\
(4) & t_i + D_i \leq t_j + b_{ij}II \\
(5) & \epsilon_{ij} + \epsilon_{ji} \leq 1 \\
(6) & r_j - r_i - 1 - (\epsilon_{ij} - 1)a_k \geq 0 \\
(7) & r_j - r_i - \epsilon_{ij}a_k \leq 0 \\
(8) & \mu_{ij} + \mu_{ji} \leq 1 \\
(9) & m_j - m_i - 1 - (\mu_{ij} - 1)II \geq 0 \\
(10) & m_j - m_i - \mu_{ij}II \leq 0 \\
(11) & \epsilon_{ij} + \epsilon_{ji} + \mu_{ij} + \mu_{ji} \geq 1 \\
(12) & t_i = y_iII + m_i \\
(13) & r_i \leq a_k - 1 \\
(14) & m_i \leq II - 1 \\
(15) & t_i + D_i \leq SL_{max}
\end{array}$$

To guarantee that two instructions will not occupy the

TABLE II: Problem signature from the ILP modulo scheduling replicated from [10]

Input	
II	Initiation Interval;
$O = \{0, \dots, n-1\}$	IR Instructions;
$D_i, i \in O$	Instruction i delay;
$E = \{(i \rightarrow j; b)\} \subseteq O \times O \times \mathbb{Z}_0^+$	Dependency edges on datapath. $b \geq 1 \Leftrightarrow backedge$;
$R = mem, dsp$	Resource types;
$A = a_k \mid k \in R$	Instance of resource k ;
$L_k \subseteq O$	Resource limited operations of type k ;
$L = \bigcup_{k \in R} L_k$	Union of all resource limited operations;
Output	
$t_i, i \in O$	Start time of operation i ;
$r_i, i \in L$	Resource instance used by operation i ;
Decision Variables	
$t_i, i \in O$	Start time of operation i ;
$r_i, i \in L$	Resource instance used by operation i ;
$m_i, i \in L$	Congruence class of operation i ;
$y_i, i \in L$	Helper variable of operation i ;
$\epsilon_{ij}, \forall k \in R: \forall i, j \in L_k, i \neq j$	$= 1$ if $r_i < r_j$, $= 0$ otherwise;
$\mu_{ij}, \forall k \in R: \forall i, j \in L_k, i \neq j$	$= 1$ if $m_i < m_j$, $= 0$ otherwise;

same MRT slot, the binary overlap variables ϵ_{ij} (lines 5 to 7, and 11) and μ_{ij} (lines 8 to 11) are defined for each pair of instructions I_i and I_j that share the same resource type k , making the number of overlap variables to grow quadratically with the loop size. Lines 4 and 15 capture the dependency and timing constraints used in [10]. Lines 13 and 14 provide the upper bounds for r_i and m_i , respectively. Line 12 links the values of t_i to m_i using the auxiliary variables y_i . We use the notation on Table II through the remaining of this paper.

[12] presents a comparison between the existing module schedulers in the literature, including iterative, slack, swing and stage modulo schedulers, that concludes that the iterative modulo scheduler (which is the base for SDCS) achieves better II values, despite requiring longer executions times. As the achieved value of II is critical in the HLS context when targeting high-performance designs, the iterative method has been chosen to act as a baseline in this work.

IV. SDC FORMULATION FOR SEPARATING SCHEDULING AND ALLOCATION

The main idea behind the proposed approach is to separate the ILP formulation scheduling and allocation parts, and to use an SDC formulation to solve the scheduling part and another method to explicitly traverse through valid MRTs (i.e. the allocation space). This approach differs from [6], which modifies the SDC problem (scheduling part) by adding constraints trying to force its solution to have a valid MRT indirectly.

Section IV-A presents how Formulation 1 is transformed to allow the scheduling and allocation parts to be separately explored. Section IV-B presents a relaxation of the

proposed formulation, that is useful due to its feasibility properties.

A. Separating Scheduling and Allocation

On Formulation 1, the overlap variables (lines 5 to 11) guarantee that two instructions are not allocated to the same MRT slot, and thus ensuring a valid MRT. In the proposed approach, the allocation part will ensure a valid MRT construction, and as such the overlap variables are eliminated from the formulation. The rest of the problem formulation has: dependency constraints (line 4), other HLS constraints (line 15), and the linker constrains between t_i and m_i (line 12). As such, the remaining problem can be rewritten as:

$$\begin{aligned} &\textbf{minimize:} && \sum_{i \in O} t_i \\ &\textbf{subject to:} && \\ &t_i - t_j && \leq b_{ij}II - D_i \\ &t_i && = m_i + y_iII \end{aligned}$$

Substituting t_i by $m_i + y_i \times II$ in all constraints and objective function, t_i is eliminated from the problem, and y_i is promoted to a decision variable, resulting in Formulation 2. Note that t_i values can be calculated with the m_i values and the solution y_i .

Formulation 2: SDC base problem for scheduling and allocation separation

$$\begin{aligned} (1) \quad &\textbf{minimize:} && \sum_{i \in O} y_i * II + m_i \\ (2) \quad &\textbf{subject to:} && \\ (3) \quad &y_i - y_j && \leq \left\lfloor \frac{b_{ij}II - D_I - (m_i - m_j)}{II} \right\rfloor \end{aligned}$$

Other HLS constraints in the SDC can also be added in Formulation 2 in a similar way.

Assuming a valid MRT has been produced by the allocation stage, the m_i and r_i are known, and by solving the SDC Formulation 2, the y_i values are calculated in polynomial time, leading to the t_i values. Furthermore, the latency of the solution is used as an MRT measure of quality, since solving Formulation 2 for different MRTs results in schedules that differ only in latency, which is the only non-constant factor in $total_{cycles} = l + II \times (tc - 1)$, for a given II .

Summarising, formulation 2 allows us to divide the modulo scheduling problem into the allocation and scheduling parts explicitly. The proposed method traverses the MRT space and produces possible allocations, which then drive the scheduling part through the SDC Formulation 2. The traversing of the MRT space is guided by the latency of the obtained solutions.

The traversing of the MRT space can be performed by various methods such as any heuristic, meta-heuristic, evolutionary algorithm, machine learning techniques, while the scheduling part can be optimally solved using Formulation 2, in polynomial time.

Nevertheless, Formulation 2 is not feasible for any MRT construction.

Definition: We define “feasible MRT” and “infeasible MRT” as MRTs to which Formulation 2 is feasible and infeasible, respectively.

The feasibility and infeasibility of Formulation 2 is demonstrated by the example in Figure 1a. Let’s assume that we only have 2 resource instances to execute instructions $I_i = \{1, 2, 3, 4, 5\}$, thus $minII = 3$ and the MRT has 3 rows and 2 columns. Figure 1b and 1e show the Non-Resource Constrained As Soon As Possible (NRCASAP), and the optimal MRTs, respectively.

Figure 1c shows a feasible MRT, with scheduling times $\mathbf{t} = \{0, 1, 1, 3, 2\}$, and the back-edge constraint given as $t_5 - t_1 \leq b_{5,1} \times II - D_5 \Rightarrow t_1 \geq 0$, which is satisfied since $t_1 = 0$. Figure 1d shows an infeasible MRT, with scheduling times $\mathbf{t} = \{0, 3, 1, 1, 5\}$, and the back-edge constraint given as $t_5 - t_1 \leq b_{5,1} \times II - D_3 \Rightarrow t_1 \geq 3$, which is not satisfied since $t_1 = 0$.

(a) Data flow

m	a_0^0	a_0^1	—
0	i_1	—	—
1	i_2	i_3	i_4
2	i_5	—	—

(b) NRCASAP

m	a_0^0	a_0^1
0	i_1	i_4
1	i_2	i_3
2	i_5	—

(c) feasible MRT

m	a_0^0	a_0^1
0	i_1	i_2
1	i_4	i_3
2	i_5	—

(d) infeasible MRT

m	a_0^0	a_0^1
0	i_1	—
1	i_2	i_3
2	i_5	i_4

(e) optimal MRT

Fig. 1: Example of different MRTs for a DFG.

As such, the allocation space exploration needs to also consider infeasible MRTs, for which the quality of the obtained solution (i.e. latency) is not available. As the MRT is infeasible, the operations cannot be scheduled, and as such, there is no quality metric (i.e. schedule latency) associated with the MRT. In this fashion, a possible solution is to discard the infeasible MRTs and keep generating MRTs and analysing only the feasible ones, as summarized in the flow presented in Figure 2. However, experimental work indicated that finding a feasible MRT through the above algorithm is rare. More specifically, using benchmark **cp** (described in Section VI-C), 145872.23 random MRTs on average (50 repetitions) need to be created before finding a feasible one.

Instead of discarding infeasible MRTs, this paper proposes an approach that would allow the association of an infeasible MRT with a latency value, and as such would allow its incorporation in the MRT exploration stage. The proposed approach is based on relaxing Formulation 2. The relaxed formulation solution results in a schedule for the loop with an associated latency, even though this may not be an actual solution for the non-relaxed problem. However,

Fig. 2: Straight forward flow to handle infeasible MRTs.

Fig. 3: Flow to handle infeasible MRTs using relaxation of Formulation 2.

it can be used to compare infeasible MRTs with feasible ones, as illustrated in the flow in Figure 3.

Section IV-B presents a relaxation for Formulation 2 that is guaranteed to be feasible even for infeasible MRTs.

B. A Relaxation of Formulation 2

Formulation 3 is a relaxation of Formulation 2 by renaming $y_i II = x_i$.

Formulation 3: SDC base problem relaxation for scheduling and allocation separation

- (1) **minimize:**
- (2) $\sum_{i \in O} x_i + m_i$
- (3) **subject to:**
- (4) $x_i - x_j \leq -D_i + b_{ij}II - (m_i - m_j)$

In Formulation 3, there is no guarantee that x_i values are multiples of II , and as such, y_i are not guaranteed to be integer. Thus, there is no guarantee that $t_i \% II = (x_i + m_i) \% II = m_i$. In other words, the MRT that corresponds to the solution (calculated as $t_i \% II$) is not guaranteed to be the same valid MRT used to construct the problem in Formulation 3, thus, it is not guaranteed to be valid. This has no consequences for the flow proposed in Figure 3, since the schedule values (t_i) are only used for comparison.

In the following paragraphs, it is proved that if a feasible MRT exists for Formulation 2, then Formulation 3 is always feasible (regardless of the MRT). As such, by employing Formulation 3, the proposed methodology ensures that a schedule can be derived that can be used to guide the DSE process.

Notation: let δx_{ab} represent the $x_a - x_b$.

For the following proof, the following bounds need to be introduced:

- 1) $-(II - 1) \leq \delta m_{ij} \leq (II - 1)$
- 2) $0 \leq D_i \leq (II - 1)$
- 3) $0 \leq (II - 1) - D_i \leq (II - 1)$

Bound 1 is a consequence that $0 \leq m_i \leq (II - 1)$ for any instruction I_i by definition. Bound 2 implies that instruction I_i delay is smaller than II , which is true for functional units are not pipelined themselves. Thus, it is imperative that every instruction has to finish its execution before being executed again. Bound 3 is a consequence of bound 2.

The first consideration to make is that the dependencies created by the forward edges ($I_i \rightarrow I_j$) can always be satisfied regardless of the MRT. This is because the RHS in Formulation 2 can be only 0 or -1 , since $b_{ij} = 0$, and bounds 1 and 2 make $-(II - 1) - (II - 1) \leq -D_i - \delta m_{ij} \leq 0 + (II - 1)$. The forward edge constraints are also always valid for Formulation 3 since it is a relaxation of Formulation 2. Thus, we will only consider the back-edges constraints from now on.

Using bound 1, we can expand the RHS of Formulation 2 to enclose all possible values of the MRT, including the ones that would make it infeasible, resulting in Inequality inq.1.

$$\delta y_{ij} \leq \left\lfloor \frac{b_{ij}II - D_i - \delta m_{ij}}{II} \right\rfloor \leq \left\lfloor \frac{b_{ij}II - D_i + (II - 1)}{II} \right\rfloor \quad (\text{inq.1})$$

In the same fashion, for Formulation 3 to be feasible for any MRT, is equivalent to rewrite the RHS on Formulation 3 as Inequality inq.2. That is, δx_{ij} has to be smaller than the lower bound of $b_{ij}II - D_i - \delta m_{ij}$.

$$\delta x_{ij} \leq b_{ij}II - D_i - (II - 1) \leq b_{ij}II - D_i - \delta m_{ij} \quad (\text{inq.2})$$

Thus, the condition for the feasible space of Formulation 3 to encompass all values of MRT is if Inequality inq.1 is encompassed by Inequality inq.2, resulting in Inequality inq.3.

$$\left\lfloor \frac{b_{ij}II - D_i + (II - 1)}{II} \right\rfloor \leq b_{ij}II - D_i - (II - 1) \quad (\text{inq.3})$$

If $II = 1$, Inequality inq.3 always holds. Thus, we only need to analyse the case when $II > 1$.

Using bound 3, we can see that the Left-Hand-Side (LHS) of Inequality inq.3 is equal to b_{ij} , implying that the condition in Inequality inq.3 can be rewritten as Inequality inq.4.

$$\begin{aligned}
b_{ij} &\leq b_{ij}II - D_i - (II - 1) \\
\Rightarrow b_{ij} &\leq b_{ij} + b_{ij}(II - 1) - D_i - (II - 1) \\
\Rightarrow b_{ij} &\geq \left\lceil \frac{D_i + (II - 1)}{(II - 1)} \right\rceil \quad (\text{inq.4})
\end{aligned}$$

Finally, using bound **3**, and that $b_{ij} \geq 1$ by definition, we conclude that Inequality inq.4 always holds, meaning also that Inequality inq.3 always holds.

As such, if Formulation 2 is feasible for an MRT, then Formulation 3 is always feasible regardless the MRT.

V. SDC BASED GENETIC ALGORITHM

As noted in Section IV-A, any method of choice can explore the allocation space as long valid and feasible MRTs are produced by the method where Formulation 2 is used to solve the scheduling problem. In this work a Genetic Algorithm (GA) is selected as it is known to be applicable for problems without information about the solution structure [13].

The first step in the creation of a GA is the design of the GA chromosomes. Formulations 2 and 3 allow us to select the MRTs as chromosomes (individuals). In this way, we reduce the number of problem variables (described in lines 5 to 11 on Formulation 1) by capturing these constraints in the GA chromosome encoding. As will be explained later, the chromosome evolution guarantees a valid MRT, and as such, lines 5 to 11 are removed (with the overlap variables).

A. The Genetic Algorithm

The GA purpose is to evolve MRTs (individuals) according to the final schedule latency (fitness). That is, since II is given, and tc is constant, the GA evolves individuals to reduce the solution latency. MRTs are implemented as lists of (m_i, r_i, I_i) triplet, where (m_i, r_i) correspond instruction I_i MRT slot, which is a common way to represent sparse matrices such as the MRT.

1) *Main Algorithm*: The proposed GA algorithm is described in Algorithm 3, to which we will refer as GAS. The parameters impacting GAS are the population size $nPop$, the insemination rate i_r , the number of new individuals created at each generation $offspringSize$, the minimum number of generation $minGen$, and the mutation probability $mutationProb$.

The first step is the creation of a random population (lines 1 to 4). The initial population is inseminated with individuals created by randomly legalizing the NRCASAP schedule MRT (lines 5 and 11) using the legalization process (described in details in Algorithm 5). Since the NRCASAP schedule is the unreachable ideal, we expect that its MRT values contain “good genes” and the optimality will be close to a slight modification of its MRT .

The population insemination introduces some knowledge about the solution structure into the GA, which is shown to improve GA results as shown in [14]. Figure 4 shows the number of generations a typical GAS execution takes to find a feasible MRT in function of the population size, with and without the insemination. For Figure 4, we used

the benchmark **complex** (described in details in Section VI-C).

Fig. 4: Average number of generations needed to find a feasible MRT with and without the insemination process.

Consider the example in Figure 1a, with $a_0 = 2$ resource instances and $MII = 3$. Figure 1b presents the NRCASAP MRT, which had three instruction with congruence class $m = 1$. The legalization process can generate the MRTs presented in Figures 1c, 1d, or 1e, which will be used to inseminate the initial population.

The population is then evolved for $minGen$ generations (lines 13 to 23). The evolution process consists of creating $offspringSize$ new individuals, called cubs, (lines 14 to 22), from two different individuals randomly selected from the population (line 15) through a cross-over process (described in details in Algorithm 6). The mutation occurs by attributing a new random empty MRT slot to the instructions in the new created MRTs (lines 18 to 20). Then, the cubs are added to the population (lines 21 and 22), what results in a $nPop + offspringSize$ individuals population. Selection is made by ordering and selecting the $offspringSize$ best population individuals (lines 23 and 24).

Finally, the algorithm does not stop until it reaches the minimum number of generations $minGen$, or until it finds a feasible MRT (line 27). Furthermore, we define $budget$ as a maximum number of generations in case a feasible MRT is not found (lines 25 and 26), which cause GAS to return a failure for the given II meaning that the candidate II should be increased.

2) *Evaluation Algorithm*: Algorithm 4 describes the evaluation function, which considers two cases. The first case is when the MRT is feasible, and thus we have a solution for the problem. The second case is when the MRT is infeasible, and thus Formulation 3 is used to calculate the fitness with a penalty given by how many conflicts there were in the solution of Formulation 3.

First, the algorithm tries to solve Formulation 2 (line 1), and the resulting schedule latency is returned if it is

Algorithm 3: Genetic Algorithm to calculate a schedule for a given II .

```

input :  $II$  and GA parameters
output : A schedule for  $II$ 
/* Initialize population step */
1 for  $i \leftarrow 0$  to  $nPop - 1$  do
2   create a valid  $MRT_i$  by assigning each
   instruction to a random empty slot;
3    $fitness_i \leftarrow \text{evaluateIndividual}(MRT_i)$ ;
4   add  $(MRT_i, fitness_i)$  to population;

/* Insemination Process */
5 Calculate the non-resource constrained ASAP
  schedule;
6 Calculate the non-valid  $MRT_{ASAP}$ ;
7 for  $i \leftarrow 0$  to  $i_r * nPop$  do
8    $MRT_i \leftarrow MRT_{ASAP}$ ;
9   validateIndividual( $MRT_i$ );
10   $fitness_i \leftarrow \text{evaluateIndividual}(MRT_i)$ ;
11  add  $(MRT_i, fitness_i)$  to population;

/* Population evolution */
12  $gen \leftarrow 0$ ;
13 do
14   /* Offspring creation */
15   for  $i \leftarrow 0$  to  $\frac{offspringSize-1}{2}$  do
16     ( $p1, p2$ )  $\leftarrow$  different random individuals in
     the population;
17     ( $cub1, cub2$ )  $\leftarrow$  crossover( $p1, p2$ );
18     for  $cub \in (cub1, cub2)$  do
19       /* mutation */
20       for instruction  $I \in MRT_{cub}$  do
21         if random probability
           > mutationProb then
22           reallocate  $I$  in a random
           available slot in the  $MRT_{cub}$ ;
23        $fitness_i \leftarrow$ 
         evaluateIndividual( $MRT_{cub}$ );
24       add  $(MRT_{cub}, fitness_{cub})$  to
         population;

/* selection */
25 sort population according to increasing fitness
  order;
26  $population \leftarrow nPop$  first individuals of
  population;
27 if  $gen > budget \times minGen$  then
  fail to schedule with the given  $II$ ;
28 while  $!(gen \geq minGen \ \&\& \ foundFeasible)$ ;

```

feasible (lines 2 to 5). If Formulation 2 is infeasible for the given MRT , Formulation 3 (line 7) is solved, which is guaranteed to be feasible (Section IV-A). The MRT corresponding to the relaxed problem solution $MRT_{relaxed}$ is calculated using $m_i^{relaxed} = t_i \% II$ (line 9), where $t_i = (x_i + m_i)$. The latency is calculated using t_i values (line 9). Since $MRT_{relaxed}$ is not guaranteed to be valid, we

modify it using the validation function (line 11), which return how many resource conflicts the non-valid MRT had and randomly fixes the MRT (described in Algorithm 5). Finally, the function returns the latency l for the relaxed schedule plus a penalty according to the number of resource conflicts the MRT had times II (line 12). This penalty captures that the relaxed schedule can be a solution for the non-relaxed one, which happens when no resource conflicts occur, and also punishes relaxed schedules with conflicts, making them less likeable to survive through generations.

Algorithm 4: Function for individual evaluation for the GA presented in Algorithm 3.

```

input :  $MRT$ 
output : Fitness for the  $MRT$ 
1  $success \leftarrow$  solve Formulation 2 for  $MRT$ ;
2 if success then
3   Calculate the solution latency  $l$ ;
4   return  $l + II * tc$ ;
5    $foundFeasible \leftarrow true$ 
6 else
7   solve Formulation 3 for  $MRT$ ;
8   Calculate the non valid  $MRT_{relaxed}$ ;
9   Calculate the solution latency  $l$ ;
10   $MRT \leftarrow MRT_{relaxed}$ ;
11   $n\_outs \leftarrow \text{validateIndividual}(MRT)$ ;
12  return  $l + II * n\_outs$ ;

```

As example, consider Figure 1c, which is feasible and Formulation 2 returns the schedule $t_{1c} = \{0, 1, 1, 2, 3\}$, with latency $l_{1c} = 4$, resulting in $fitness_{1c} = 4$.

Now, consider Figure 1d, which is infeasible, and solving Formulation 3 results in $x_{1d} = \{0, 1, 0, 0, 0\}$, thus $t_{1d} = \{0, 1, 1, 1, 2\}$. Note that $MRT_{relaxed}$ has instructions $\{I_2, I_3, I_4\}$ in the same congruence class, thus it is not valid, and the validation function will return the latency $l_{1d} = 3$ and the number of conflicts $n_outs = 1$, resulting in $fitness_{1d} = 3 + 1 * 3 = 6$. Furthermore, $MRT_{relaxed}$ need to be validated, before being added back to the population.

3) *Validation Algorithm:* Algorithm 5 describes the function used to modify non-valid MRT s into valid ones, randomly handling conflicts, and also counting how many conflicts there were in the MRT . This function is necessary for the insemination and individual evaluation processes since invalid MRT s can be generated in both procedures.

Each resource constrained instruction I is attempted to be allocated in its current MRT slot (lines 2 and 3), and if there is already another instruction allocated to that slot, I is attempted to be scheduled in another resource instance, with the same congruence class (lines 4 to 8). This solves conflicts where two instructions are assigned for the same resource instance, but there are still other available instances in the same congruence class.

If all resources are busy, a randomly allocated instruction I_r in the congruence class is selected (line 10), and the number of conflicts is incremented (line 16). Finally, both instruction dispute a coin toss to define which one will be

allocated to a random *MRT* empty slot (lines 11 to 15). This random selection is made as there is no information on which allocation is the best to be performed.

Algorithm 5: Function for counting and fixing MRT conflicts for the GA presented in Algorithm 3.

```

input : MRT
output : New valid MRT and n_outs
1 n_outs  $\leftarrow$  0;
2 for each instruction I  $\in$  MRT do
3   if MRT slot is not available then
4     m  $\rightarrow$  I congruence class;
5     r  $\rightarrow$  I resource instance;
6     rtry  $\leftarrow$  any available resource instance in
       the congruence class m;
7     if  $\exists$  rtry then
8       allocate I in slot (m, rtry);
9     else
10      Ir  $\leftarrow$  a random allocated instruction of
        congruence class m;
        /* 50 % chance for each case */
11      if random(0, 1)  $\geq$  0.5 then
12        reallocate I to an available random
          MRT slot (new random (m, r)
            values);
13      else
14        allocate I in Ir slot in the MRT;
15        reallocate Ir to an available random
          MRT slot;
16      n_outs = n_outs + 1;
17 return n_outs;

```

To illustrate the validation process, consider Figure 1b. First *I*₁, *I*₂, and *I*₃ will be successfully allocated in their congruence class. Then *I*₄ will fail to be allocated with *m*₄ = 1. A random allocated instruction with *m* = 1 is chosen, which can be *I*₂ or *I*₃. Let's assume that *I*₂ is chosen, then a random selection between *I*₂ and *I*₄ will be performed. If *I*₂ is selected, it remains as it is, and *I*₄ is randomly allocated into an empty slot, which can result to MRTs depicted in Figure 1c or Figure 1e. If *I*₄ is selected, *I*₂ is randomly allocated which leads to the MRT depicted in Figure 1d.

4) *Cross-Over Algorithm*: Algorithm 6 describes the single point cross-over function used. The cross-over point is a randomly chosen number of instructions (line 1), which are copied to *cub*₁ and *cub*₂ respectively from parents *p*₁ and *p*₂ (lines 2 to 4). The rest of the instructions are copied from the opposite parent (lines 5 to 7). This simple process might cause MRT conflicts since an instruction of the first part of *p*₁ can occupy the same slot as an instruction in the second part of *p*₂. Possible conflicts are solved using the validation process described in Algorithm 5.

To illustrate the cross-over, consider Figures 1c and 1d, which are represented by the lists *p*₁ = *MRT*_{1c} =

Algorithm 6: Cross-over function for the GA presented in Algorithm 3.

```

input : Parents p1 and p2
output : Offspring cub1 and cub2
1 crossPoint  $\leftarrow$  rand(1, nInsts - 1);
   /* copy first part of chromosome from
     corresponding parent */
2 for Ii  $\leftarrow$  0 to crossPoint - 1 do
3   cub1 :: (mi, ri, Ii)  $\leftarrow$  p1 :: (mi, ri, Ii);
4   cub2 :: (mi, ri, Ii)  $\leftarrow$  p2 :: (mi, ri, Ii);
   /* copy second part of chromosome from the
     other parent */
5 for Ii  $\leftarrow$  crossPoint to nInsts - 1 do
6   cub1 :: (mi, ri, Ii)  $\leftarrow$  p2 :: (mi, ri, Ii);
7   cub2 :: (mi, ri, Ii)  $\leftarrow$  p1 :: (mi, ri, Ii);
   /* fix possible conflicts in the cubs */
8 validateIndividual(MRTcub1);
9 validateIndividual(MRTcub2);

```

{(0, 0, *I*₁), (1, 0, *I*₂), (1, 1, *I*₃), (0, 1, *I*₄), (2, 0, *I*₅)}

 and *p*₂ = *MRT*_{1d} = {(0, 0, *I*₁), (2, 0, *I*₂), (1, 1, *I*₃), (1, 0, *I*₄), (0, 1, *I*₅)} respectively. The cross-over point is a random number between 1 and 4. Supposing the cross-over point is 2, the resulting cubs would be *cub*₁ = {(0, 0, *I*₁), (1, 0, *I*₂), (1, 1, *I*₃), (1, 0, *I*₄), (0, 1, *I*₅)} and *cub*₂ = {(0, 0, *I*₁), (2, 0, *I*₂), (1, 1, *I*₃), (0, 1, *I*₄), (2, 0, *I*₄)}.

VI. COMPARISON WITH SDC AND ILP MODULO SCHEDULERS

A. Performance Models

The total number of individuals evaluated by the GA is: *totalIndividuals* = *nPop* × (1 + *i_r*) + *maxGen* × *offspringSize*. Under the proposed formulation, a maximum of two SDC problems are solved for each individual. As such, for GAS to be faster than SDCS, the *totalIndividuals* need to be less than the number of problems solved by SDCS.

Let *nPop* = *offspringSize* = $\alpha \times n$ and *maxGen* = β , where α and β are parameters defined by the user, and *n* is the loop size. Thus, *totalIndividuals* = $n \times (\alpha(1 + \text{inseminationRate}) + \beta)$. Since in the worst-case two SDC problems are solved per individual (Formulations 2 and 3), GAS is set to solve $\mathcal{O}(n)$ SDC problems.

The conditions for GAS to be faster than the SDCS worst-case is:

$$\begin{aligned}
 & 2 \times (\alpha n(1 + i_r) + \beta(\alpha n)) < \text{budgetRatio} \times n^2 \\
 \Rightarrow & 2\alpha(\beta + 1 + i_r)1 < \text{budgetRatio} \times n \\
 \Rightarrow & \frac{2\alpha(\beta + 1 + i_r)}{\text{budgetRatio}} < n \quad (\text{inq.5})
 \end{aligned}$$

Inequality inq.5 provides a sufficient condition for GAS to be faster than SDCS worst-case, and indicates that there exists a large enough *n* to which GAS will be faster than SDCS worst-case. The impact of the above condition will

be revisited in the performance evaluation section where the parameters α and β are tuned.

B. Characterisation

A high-level comparison between SDCS, ILPS and GAS is presented in Table III. The table shows that ILPS has the worst solver complexity, which also has the worst scaling with the loop size n .

In the case of GAS, the problem size scales better than in the case of SDCS, as GAS creates only one variable for each instruction I_i , instead of D_i (as proposed by [10]), as well as GAS does not add constraints related to the allocation as SDCS does (Section III-A). As such, the GAS's main sources of speed-up compared to SDCS is a) the smaller number of problems solved, which is achieved due Formulation 3 usage and the efficient MRT exploration by the GA, and b) the fact that the generated problems exhibit smaller sizes.

TABLE III: Comparison between the schedulers.

	SDCS	ILPS	GAS
Problem type	SDC	ILP	SDC
Solver complexity	Polynomial	Exponential	Polynomial
Problem size	grows during execution	$\mathcal{O}(n^2)$	0
# Problems solved	$\mathcal{O}(n^2)$	1	$\mathcal{O}(n)$
II	Not optimal	Optimal	Not optimal
Latency	Not optimal	Optimal	Not optimal

In case of SDCS, its main source of inefficiency lies in its iterative heuristic search for finding a solution with a valid MRT. A secondary source of inefficiency is its incapability to prove infeasibility for a given II , requiring the allocated time budget expiration. Such infeasible II s are explored by [10] using a different *RecMII* estimation than [6], which results in smaller and infeasible MII .

The ILPS's main source of inefficiency lies on the quadratic problem size scaling, and on the exponential complexity of its solver. In contrary, GAS search solves a number of problems that scale linear with n , as opposed to the quadratic scaling of number of problems in SDCS. Furthermore, GAS problems are in the SDC form, and thus its solver exhibits polynomial complexity.

Concerning the quality of the achieved solution, SDCS does not guarantee latency or II optimality due to its heuristic nature. If a solution cannot be found within a pre-allocated time budget, the II is increased, and a new search is performed. On the contrary, ILPS guarantees both latency and II optimality, since the solver guarantees to find the optimal solution for the problem, for a given II , or declare infeasibility. GAS does not guarantee latency or II optimality since the GA may fail to find a feasible MRT before it reaches its stop criteria.

C. Benchmark Selection

The proposed approach is evaluated using 13 HLS benchmarks, and their details and characteristics are

summarised in Table IV. All suitable loops of the above benchmarks were submitted for scheduling using the SDCS, ILPS, and GAS approaches. Benchmarks **mt**, **dv**, **fat**, **cp**, and **ac** are the same ones used in [6]. Please note that benchmarks **mt**, **fat**, and **ac** are synthetic benchmarks that have been designed specifically to test SDCS's scalability and are originally used in [6]. These benchmarks use more than 50% resource-constrained nodes.

The rest of the benchmarks exhibit various data access pattern, loop dependencies, logic operations, nested loops, and induction variables to validate the approaches under different code constructions.

Moreover, as benchmarks **rs**, **cv**, **gp**, **ft**, and **sh** contain more than one loop in their source code, the entries in Tables IV, V, and V reflect the combined loop characteristics.

All schedulers were implemented as part of LegUP [21] infrastructure, where the SDCS is natively implemented. As LegUP imposes restrictions on which loops can be pipelined, the same restrictions are also applied for ILPS and GAS. As such, all loops have bounds and array sizes modified to statically determined values, and loops with unknown bounds and more than one induction variable were set not to be pipelined. Since only innermost loops, which are in the main C file, can be pipelined, all benchmarks were adapted to a single C file, and only the innermost loops were selected to be pipelined. Furthermore, a full inline is applied in the whole code, since LegUP implements local memories only for arrays that are used exclusively within the context of one, and only one, function.

It should be noted that the above code modifications are applied due to the LegUP's restrictions. The discussed schedulers can be implemented on other compilers without these restrictions, as has been demonstrated by [10].

D. Performance Evaluation

The results presented in this section were obtained on an Ubuntu 14.04 machine, with 16 GB of RAM, and an Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz. All schedulers are implemented inside LegUP 4.0 structure as LLVM 3.5 opt passes. **Gurobi** 7.5 [22] is chosen as solver for all schedulers.

The SDCS budget is set as *budgetRatio* = 6 (LegUP default) and *INCREMENTAL_SDC* = 1 (speeds up SDCS [23]). The ILPS time budget is set to 5 minutes.

The parameters of GAS were empirically set as follows: $\alpha = 0.10$, $\beta = 10$, and $i_r = 0.05$, for all benchmarks, satisfying $\alpha(\beta + 1 + i_r) = 1.105 < 6 = \text{budgetRatio}$, with the expectation that GAS will, in most cases, solve fewer problems than the SDCS worst case. The mutation rate is empirically defined as *mutationProb* = 1%.

The three schedulers are tested with respect to the obtained quality of solution and the time required to reach that solution on the above benchmarks. The obtained results are presented in Tables V and VI, corresponding to the average of 50 repetitions. Overall, all schedulers achieved similar results (II and loop latency), but with a clear time improvement for the proposed GAS scheduler.

TABLE IV: Benchmark selection.

Name	Short Name	Benchmark	Loop size	Ar- rays Size	Operations +/*/%/+/+/*/%/[]	Constraints +/*/%/+/+/*/%/[]	Dis- tance	recMII/ resMII/ MII
rs decoder [15]	rs	none	7	256	0/1/0/0/0/0/18	X/1/X/X/X/X/2	1	8/13/13
covariance [16]	cv	Polybench	38	500	0/0/0/1/1/0/6	X/X/X/1/1/X/2	1	47/4/47
gauss pivoting [17]	gp	MIT [18]	45	100	0/0/0/2/1/0/8	X/X/X/1/1/X/2	1	23/5/23
row col [17]	rc	Xilinx	18	100	0/0/0/0/0/0/6	X/X/X/X/X/X/2	200	1/3/3
fft [19]	ft	LegUp	55	1024	0/0/1/0/0/0/6	X/X/1/X/X/X/2	1	3/5/5
sha [20], [10]	sh	CHStone	111	8192	0/0/0/0/0/0/4	X/X/X/X/X/X/2	1	5/9/9
jacobi 2d [17]	j2	Polybench	28	100	0/0/0/4/1/0/6	X/X/X/3/1/X/2	1	75/3/75
multipliers [6]	mt	LegUp	28	100	7/2/0/0/0/0/7	3/1/X/X/X/X/1	1	3/4/4
dividers [6]	dv	LegUp	72	100	12/0/4/0/0/0/11	3/X/3/X/X/X/2	1	3/6/6
faddtree [6]	fat	LegUp	81	100	0/0/0/21/0/0/22	X/X/X/3/X/X/2	1	27/11/27
add int [17]	ai	LivemoreC	82	100	0/0/0/21/9/0/12	X/X/X/3/3/X/2	1	1/12/12
complex [6]	cp	LegUp	98	100	0/7/2/0/0/0/25	X/3/1/X/X/X/2	9	20/13/20
adderchain [6]	ac	LegUp	92	100	48/0/0/0/0/0/24	3/X/X/X/X/X/2	1	3/12/12

TABLE V: Number and sizes of problems solved.

Name	# Solves			# Variables			# Constraints		
	SDCS	ILPS	GAS	SDCS	ILPS	GAS	SDCS	ILPS	GAS
rs	50	8	48	119	344	84	341	816	82
cv	22	3	57.80	107	131	38	324	264	43
gp	24	3	50.93	103	177	45	326	418	66
rc	13	1	6	28	117	18	105	332	35
ft	25	3	18	75	201	55	217	480	68
sh	57	5	85.23	125	425	111	347	1071	137
j2	25	1	21.70	112	229	28	365	638	39
mt	23	1	19.80	78	208	28	262	584	42
dv	67	1	66.13	351	534	73	1143	1622	157
fat	175	1	84.67	402	2113	81	1333	6929	103
ai	94	1	62	530	1114	82	1694	3387	105
cp	373	1	120.17	399	1488	98	1291	4837	159
ac	470	1	96.60	145	1276	100	569	4133	149
geomean	56.09	1.71	43.90	145.43	409.35	56.43	467.98	1119.88	79.42
ratio	1	0.03	0.78	1	2.81	0.39	1	2.39	0.17

Table V presents the number of problems solved, and the number of variables and constraints for all schedulers. As the results demonstrate, GAS solves significantly simpler problems (i.e. fewer variables and constraints) than SDCS. Furthermore, the larger (i.e. more instructions) the code is, the fewer problems are solved by GAS when compared to SDCS.

Table VI presents the obtained II , the solution's latency, the total number of cycles, and the scheduling time to reach the solution, for all loops of all benchmarks. Results show that all schedulers reached the minimum II .

With respect to the obtained solution latency, SDCS produce solutions with 12% longer latency when compared to ILPS, while GAS produces solutions with 13% longer latency when compared with SDCS. However, the impact of the latency is amortized in the total number of cycles in the equation $total_{cycles} = l + II \times (tc - 1)$, as tc usually is a larger number than l . As illustrated in Table VI, the solutions produced by GAS exhibit on average 1% penalty in the total cycles.

Finally, Table VI shows the computation time taken by SDCS, ILPS and GAS. ILPS is not comparable to the other approaches due to its exponential scaling with the problem size. GAS presents 55% improvement over SDCS in our

tests (geomean).

To exemplify scalability, Figures 5 and 6 present the computation time and achieved II as a function of the loop size, for the benchmarks in Table IV when **loop unrolling** is applied. The unrolling factor is presented in the number above the GAS marks. Benchmarks **gp**, **ft**, **sh**, and **j2** are excluded from this evaluation due to small unrolling factors that they can support (i.e. their loops can be unrolled only by factors 1 and 2).

Figure 5 illustrates that the proposed GAS approach scales better with the loop size than SDCS. Moreover, the results show that GAS can provide solutions for long codes, where for the same cases SDCS fails (missing points).

Moreover, Figure 6 shows that GAS can find smaller II s than SDCS in the case of long codes, which indicates an improved performance on the total number of cycles for large codes, contrasting the results obtained for smaller codes presented on Table VI.

Figure 7 shows the GAS solution latency for the benchmark **cp** (the most challenging benchmark as indicated by Tables V and VI) as a function of the number of generations and parameterized according to the population size. Figure 7 shows how the population size and number of generations can be increased to obtain a smaller latency, showing the

TABLE VI: Performance and computation time results.

Name	II			Latency			Total Cycles			Scheduling Time (s)		
	SDCS	ILPS	GAS	SDCS	ILPS	GAS	SDCS	ILPS	GAS	SDCS	ILPS	GAS
rs	13	13	13	23	18	20.63 ± 0.74	897	892	894.63 ± 0.74	1.04e-02 $\pm 1.02e-03$	3.61e-01 $\pm 2.21e-02$	8.47e-03 $\pm 8.30e-04$
cv	47	47	47	65	62	62 ± 0	2368	2365	2365 ± 0	5.70e-03 $\pm 5.56e-04$	1.05e-02 $\pm 1.32e-03$	8.51e-03 $\pm 1.34e-03$
gp	23	23	23	53	49	49 ± 0	30	26	26 ± 0	6.94e-03 $\pm 7.24e-04$	1.77e-02 $\pm 1.41e-03$	8.96e-03 $\pm 9.38e-04$
rc	3	3	3	6	6	6.43 ± 0.34	303	303	303.43 ± 0.34	3.70e-03 $\pm 4.60e-04$	6.95e-02 $\pm 4.29e-03$	1.45e-03 $\pm 1.40e-04$
ft	5	5	5	13	13	14.50 ± 0.61	648	648	649.50 ± 0.61	5.18e-03 $\pm 2.34e-04$	1.49e-01 $\pm 1.09e-02$	3.50e-03 $\pm 2.52e-04$
sh	9	9	9	13	9	12.37 ± 0.88	177	173	176.37 ± 0.88	1.39e-02 $\pm 1.11e-03$	3.47e-01 $\pm 1.18e-02$	1.84e-02 $\pm 7.50e-04$
j2	75	75	75	76	75	139.50 ± 7.80	7351	7350	7414.50 ± 7.80	1.12e-02 $\pm 5.32e-04$	8.25e-01 $\pm 2.43e-02$	5.23e-03 $\pm 6.95e-04$
mt	4	4	4	24	23	27.60 ± 1.31	380	379	383.60 ± 1.31	8.54e-03 $\pm 5.39e-04$	1.66e+00 $\pm 4.59e-02$	3.88e-03 $\pm 4.36e-04$
dv	6	6	6	72	71	78.20 ± 1.67	306	305	312.20 ± 1.67	6.04e-02 $\pm 8.51e-05$	3.32e+02 $\pm 5.28e-01$	2.54e-02 $\pm 3.19e-03$
fat	27	27	27	101	101	140.23 ± 7.63	1154	1154	1193.23 ± 7.63	1.67e-01 $\pm 2.34e-03$	1.19e+03 $\pm 5.30e-01$	2.79e-02 $\pm 2.46e-03$
ai	12	12	12	95	91	136.80 ± 3.34	191	187	232.80 ± 3.34	1.31e-01 $\pm 2e-03$	1.15e+02 $\pm 2.19e-01$	2.01e-02 $\pm 2.17e-03$
cp	20	20	20	127	122	141.37 ± 4.20	1807	1802	1821.37 ± 4.20	3.54e-01 $\pm 3.57e-03$	1.19e+03 $\pm 3.23e-01$	5.61e-02 $\pm 5.69e-03$
ac	12	12	12	25	12	27.33 ± 2.10	1093	1080	1095.33 ± 2.10	2.36e-01 $\pm 5.24e-03$	1.19e+03 $\pm 2.76e-01$	3.57e-02 $\pm 2.20e-03$
geomean	12.87	12.87	12.87	37.80	33.36	42.79	579.12	569.63	585.05	2.45e-02	3.66e+00	1.10e-02
ratio	1	1	1	1	0.88	1.13	1	0.98	1.01	1	149	0.45

Fig. 5: Scheduling time scaling as a function of the loop unrolling factor.

Fig. 6: Final II scaling as a function of the loop unrolling factor. Fig. 7: Latency evolution according to the number of generations for benchmark **cp**.

trade-off between computation time and solution quality.

It should be noted that the results reported are obtained by applying the above module scheduling approach on a small number of loops. However, in the case of a design space exploration in a large application, multiple kernels need to be considered for loop pipelining, resulting in a combinatorial number of possible configurations. As such the obtained gains from the proposed approach will reflect to a significant reduction of the required absolute design

space exploration time. For example, the DSE method presented by [5] evaluates up to 250 designs per benchmark. Some of these evaluations vary the loops unrolling factor (which is equivalent to our scaling tests presented on Figures 5 and 6).

Finally, tests were performed in order to evaluate the performance of the methods in cases where the initial minimum II , MII , is underestimated, leading to cases of infeasible II s. As such, benchmarks with $RecMII > ResMII$ are selected and the MII is set to $ResMII$ in order to force the schedulers to handle cases where the candidate II is infeasible. The obtained results, presented in Table VII, show that all schedulers spend more time to provide a solution as compared to Table VI, since now they also handle cases where II is infeasible, however, the schedulers relative performance has not changed.

TABLE VII: Computation time (s) for $MII = ResII$.

Name	cv	gp	j2	fat	cp	geom.	ratio
SDCS	0.02	0.02	0.04	0.18	0.36	0.06	1
ILPS	0.14	0.15	2.18	1.12e3	1.17e3	9.04	142.17
GAS	0.01	0.01	0.02	0.03	0.05	0.02	0.31

VII. CONCLUSION

The work in this paper presents a novel formulation for the Modulo Scheduling problem, as well as a proposed solution that is based partially on the use of a Genetic Algorithm. Results have shown that a significant speedup can be achieved compared to the current state-of-the-art

approaches, where the proposed formulation also enjoys better scalability properties paving the way for attacking larger source codes.

In summary, the results show that both ILPS and GAS can produce valid schedules, which are not optimal if a limited time budget is enforced. The ILPS produces non-optimal solutions during the branch and bound process in the solver, and GAS produces a valid non-“proven to be”-optimal solution for each individual. Furthermore, the resulting non-minimum latency contributes weakly to the total number of cycles for the complete loop execution, what minimizes the GAS impact on the overall loop performance.

The obtained results illustrate that the proposed scheduler achieves the number of SDC problems solved $\mathcal{O}(n)$ scalability, where n is the number of LLVM IR instructions in the loop code. This is a significant improvement compared to the worst-case scalability $\mathcal{O}(n^2)$ provided by the current state-of-the-art SDSCS approach.

Furthermore, the scaling test results indicate that GAS is capable of finding solutions for large codes where SDSCS fails to do so, and also suggest that GAS is capable to find smaller II than SDSCS, indicating a more efficient exploration of the solution space.

ACKNOWLEDGMENT

The authors would like to thank the São Paulo Research Foundation (FAPESP) for the financial support given to this project (grant number 2016/13327-6).

REFERENCES

- [1] B. C. Schafer and K. Wakabayashi, “Divide and conquer high-level synthesis design space exploration,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 17, no. 3, pp. 29:1–29:19, Jul. 2012. [Online]. Available: <http://doi.acm.org/10.1145/2209291.2209302>
- [2] Z. Wang, B. He, W. Zhang, and S. Jiang, “A performance analysis framework for optimizing OpenCL applications on FPGAs,” in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, March 2016, pp. 114–125.
- [3] B. C. Schafer, T. Takenaka, and K. Wakabayashi, “Adaptive simulated annealer for high-level synthesis design space exploration,” in *2009 International Symposium on VLSI Design, Automation and Test*, April 2009, pp. 106–109.
- [4] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong, “Polyhedral-based data reuse optimization for configurable computing,” in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA ’13. New York, NY, USA: ACM, 2013, pp. 29–38. [Online]. Available: <http://doi.acm.org/10.1145/2435264.2435273>
- [5] B. C. Schafer, “Probabilistic multiknob high-level synthesis design space exploration acceleration,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 3, pp. 394–406, March 2016.
- [6] A. Canis, S. D. Brown, and J. H. Anderson, “Modulo SDC scheduling with recurrence minimization in high-level synthesis,” in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, Sept 2014, pp. 1–8.
- [7] Z. Zhang and B. Liu, “SDC-based modulo scheduling for pipeline synthesis,” in *Proceedings of the International Conference on Computer-Aided Design*, ser. ICCAD ’13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 211–218. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2561828.2561872>
- [8] J. Cong and Z. Zhang, “An efficient and versatile scheduling algorithm based on SDC formulation,” in *2006 43rd ACM/IEEE Design Automation Conference*, July 2006, pp. 433–438.
- [9] J. Cong, B. Liu, and Z. Zhang, “Scheduling with soft constraints,” in *Proceedings of the 2009 International Conference on Computer-Aided Design*, ser. ICCAD ’09. New York, NY, USA: ACM, 2009, pp. 47–54. [Online]. Available: <http://doi.acm.org/10.1145/1687399.1687410>
- [10] J. Oppermann, A. Koch, M. Reuter-Oppermann, and O. Sinnen, “ILP-based modulo scheduling for high-level synthesis,” in *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, ser. CASES ’16. New York, NY, USA: ACM, 2016, pp. 1:1–1:10. [Online]. Available: <http://doi.acm.org/10.1145/2968455.2968512>
- [11] S. Venugopalan and O. Sinnen, “ILP formulations for optimal task scheduling with communication delays on parallel systems,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 1, pp. 142–151, Jan 2015.
- [12] J. M. Codina, J. Llosa, and A. González, “A comparative study of modulo scheduling techniques,” in *Proceedings of the 16th International Conference on Supercomputing*, ser. ICS ’02. New York, NY, USA: ACM, 2002, pp. 97–106. [Online]. Available: <http://doi.acm.org/10.1145/514191.514208>
- [13] M. Mitchell, “An introduction to genetic algorithms,” 1996.
- [14] L. Rosa, A. Delbem, C. Toledo, and V. Bonato, “Design and analysis of evolutionary bit-length optimization algorithms for floating to fixed-point conversion,” *Applied Soft Computing*, vol. 49, pp. 447 – 461, 2016. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S156849461630429X>
- [15] A. Agarwal, M. C. Ng, and Arvind, “A comparative evaluation of high-Level hardware synthesis using reed solomon decoder,” *IEEE Embedded Systems Letters*, vol. 2, no. 3, pp. 72–76, Sept 2010.
- [16] W. Zuo, W. Kemmerer, J. B. Lim, L.-N. Pouchet, A. Ayupov, T. Kim, K. Han, and D. Chen, “A polyhedral-based systemic modeling and generation framework for effective low-power design space exploration,” in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, ser. ICCAD ’15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 357–364. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2840819.2840870>
- [17] J. Liu, S. Bayliss, and 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*, May 2015, pp. 159–162.
- [18] “Gaussian elimination with pivoting,” [Online]. Available: <http://web.mit.edu/10.001/Web/CourseNotes/GaussPivoting.c>
- [19] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, “LegUp: high-level synthesis for FPGA-based processor/accelerator systems,” in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA ’11. New York, NY, USA: ACM, 2011, pp. 33–36. [Online]. Available: <http://doi.acm.org/10.1145/1950413.1950423>
- [20] A. Prost-Boucle, O. Muller, and F. Rousseau, “Fast and standalone design space exploration for high-level synthesis under resource constraints,” *Journal of Systems Architecture*, vol. 60, no. 1, pp. 79 – 93, 2014. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S1383762113001938>
- [21] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, “LegUp: an open-source high-level synthesis tool for FPGA-based processor/accelerator systems,” *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 2, pp. 24:1–24:27, Sep. 2013. [Online]. Available: <http://doi.acm.org/10.1145/2514740>
- [22] G. Optimization, “Inc., “gurobi optimizer reference manual,” 2017,” URL: <http://www.gurobi.com>, 2017.
- [23] A. Canis, J. Choi, B. Fort, R. Lian, Q. Huang, N. Calagar, M. Gort, J. J. Qin, M. Aldham, T. Czajkowski, S. Brown, and J. Anderson, “From software to accelerators with legUp high-level synthesis,” in *Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, ser. CASES ’13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 18:1–18:9. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2555729.2555747>

The authors would like to thank the reviewers for their valuable comments and feedback that let to the improvement of this paper. In the newly submitted version, we have addressed all requests, and the details are given below.

REVIEWER # 1

The author defines the loop size is the number of instructions, but the problem formulation for ILP in TABLE II, the n represents the operations number. Because the instructions usually have more than one operations in loops. So they are not the same.

We are sorry for the nomenclature conflict. While [10] uses operations, we use IR instructions, but both refer to the same thing, an operation or instruction executed on a functional unit. We corrected the nomenclature in the text.

Though the problem size ($O(n)$) can scale with the loop size as announced in the proposed method, we think the author should give a detailed complexity analysis when adding the GA method, rather than just simply give the conclusion of polynomial complexity (TABLE III). Only the algorithm complexity doesn't grow too large or can hold a similar polynomial complexity after adding the GA method, then the reduction in problem size can bring benefits.

Section VI-A presents an analysis on how many individuals (thus MRTs and SDCs problems solved) the GAS is set to solve. We updated the second paragraph to reflect the idea more clearly.

REVIEWER # 2

We thank you for the valuable comments and observations.