Optimal and Heuristic Approaches to Modulo Scheduling with Rational Initiation Intervals in Hardware Synthesis

Patrick Sittel, Nicolai Fiege, John Wickerson, *Senior Member, IEEE,* and Peter Zipf, *Member, IEEE*

***Abstract*—A well-known approach for generating custom hardware with high throughput and low resource usage is *modulo scheduling*, in which the number of clock cycles between successive inputs (the *initiation interval, II*) can be lower than the latency of the computation. The II is traditionally an *integer*, but in this paper, we explore the benefits of allowing it to be a *rational* number. A rational II can be interpreted as the *average* number of clock cycles between successive inputs. Since the minimum rational II can be less than the minimum integer II, higher throughput is possible; moreover, allowing rational IIs gives more options in a design-space exploration. We formulate rational-II modulo scheduling as an integer linear programming (ILP) problem that is able to find latency-optimal schedules for a fixed rational II. We also propose two heuristic approaches that make rational-II scheduling more feasible: one based on identifying strongly connected components in the data-flow graph, and one based on iteratively relaxing the target II until a solution is found. We have applied our methods to a standard benchmark of hardware designs, and our results demonstrate an average speedup w.r.t. II of 1.24× in 35% of the encountered scheduling problems compared to state-of-the-art formulations.**

***Index Terms*—High-level Synthesis, Scheduling**

I. Introduction

Scheduling, the task of mapping operations to clock cycles while respecting resource constraints and maximising throughput, is central in hardware synthesis. High throughput can be achieved by interleaving the schedules of successive samples, as obtained using *modulo scheduling* [1]. The aim is to minimize the number of clock cycles between successive inputs, which is called the *initiation interval* (II).

In traditional modulo scheduling, the II is always an integer [2]. In this work, we explore the consequences of allowing *rational* IIs, such as $\frac{3}{2}$. The idea of a rational II is not new – it has been proposed by Fimmel and Müller in the domain of VLIW architectures [3]. However, our work lifts several restrictions that limit the applicability of the Fimmel–Müller approach (see Section III) and is also the first to explore rational IIs in the context of hardware design. The rough idea is to allow the number of clock cycles between successive inputs to vary, and then to reinterpret the II as the *average* of these numbers. For example, in a situation where the *integer* II is 2 (i.e., a new sample can be inserted every two clock cycles) there might be another solution where the II alternates between 1 and 2. This means that two samples can begin processing every three cycles, which can be interpreted as a *rational* II of $\frac{3}{2}$. A hardware implementation using this smaller, rational II would show significant speedup, throughput being the reciprocal of the II, and better utilisation of functional units (FUs). Moreover, since the rational numbers form a dense set, a further benefit of rational-II scheduling is that it can lead to additional points in the area/throughput trade-off, thus providing more fine-grained control over the design space.

This paper presents several new latency-optimal and heuristic approaches to solve the rational-II modulo scheduling problem, significantly improving on the state-of-the-art in terms of throughput achieved and solving time.

First, in Section IV, we formulate the general problem using integer linear programming (ILP). Our approach outperforms a naïve approach (based on partially unrolling and then using existing integer-II schedulers) in terms of problems solved and latency-optimal solutions found. However, since rational-II modulo scheduling takes longer to solve than traditional integer-II scheduling, heuristics are necessary.

Then, we propose another ILP-based approach, which we call *uniform* scheduling (Section V), that involves constraining each sample to follow the same schedule. In our example above, the two samples that are inserted every three clock cycles could follow completely unrelated schedules. The advantage of a uniform schedule is that the control flow in the resulting hardware may be simpler; the drawback is that parts of the search space for rational-II schedules are pruned.

Following Dai and Zhang [4], our first heuristic approach (Section VI) identifies *strongly connected components* (SCCs) [5] in the flow graph, in order to solve several smaller scheduling problems that are composed afterwards. This reduces the complexity of the scheduling problem but can increase the latency of solutions found.

In our second heuristic approach (Section VII), we propose the first formulation of *iterative modulo scheduling* [2] for rational IIs. The idea is first to attempt scheduling with the minimum II, but to keep trying with successively larger rational IIs if the solver times out.

Finally, in Section VIII, we compare all existing and novel approaches in terms of problems solved, II achieved, and latency achieved. We also show how rational-II scheduling can be useful for design space exploration of automatically generated FPGA designs after place & route.

P. Sittel, N. Fiege and P. Zipf are with the University of Kassel, Germany. J. Wickerson is with Imperial College London, United Kingdom.

```
for (i=2; i<N; i++) {
  A[i] = r(C[i-2]); //o0
  B[i] = r(A[i]);   //o1
  C[i] = r(B[i]);   //o2
  D[i] = r(C[i]);   //o3
  E[i] = r(D[i]);   //o4
}
```

(a) Source code

$$\begin{tikzcd}
o_0 \arrow[d] & \\
o_1 \arrow[r] & o_2 \arrow[ul, "2"'] \arrow[dl] \\
o_3 \arrow[r] & o_4
\end{tikzcd}$$

(b) DFG of the loop body

Fig. 1: Example for rational-II scheduling. Each vertex in the DFG represents one computation from the example code.

Relationship to prior work: This article is a revised and extended version of a conference paper [6]. The non-uniform ILP formulation, the SCC heuristic, and the iterative solver are all new contributions of this article.

Auxiliary material: All the new and existing scheduling algorithms discussed in this article are available in the open-source scheduling library *HatScheT* [7]. All optimisation problems have been formulated using the open-source *ScaLP* library [8], which supports the Gurobi, CPLEX, LPSolve [9] and SCIP [10] solvers. Our benchmark problems can be accessed from the *HatScheT* repository, and can be synthesized after using the open-source tools *FloPoCo* [11] and *Origami HLS* [12] to generate VHDL code.

II. Motivating Example

Consider the example given in Figure 1. Figure 1a shows a for-loop that performs an arbitrary operation r five times per iteration, modifying five different arrays (A–E) each containing at least N elements. The implementation of loops like this can be sped up by building a pipeline. The best performance of this pipeline is achieved when the modulo scheduling problem is solved optimally w.r.t. II and latency [2].

The data-flow graph (DFG) shown in Figure 1b is used to model the scheduling problem. It comprises five vertices of the same resource type r, whose latency is one cycle. The edge from o_3 to o_0 is labelled with a *dependence distance* of two to indicate a *recurrence*: operation o_0 on sample i depends on the result of operation o_3 on sample $i-2$. The other edges implicitly have a dependence distance of zero.

The maximum throughput achieved using modulo scheduling depends both on recurrences and on resource constraints (see Section IV-A). This example has one recurrence whose dependence distance is two and whose latency is three cycles, so the II cannot be less than $\frac{3}{2}$. This is called the *recurrence-constrained minimum II* [13]:

$$\mathrm{II}^{\perp}_{\mathrm{rec}} = \max_{j \in \text{recurrences}} \left(\text{latency}_j / \text{distance}_j\right), \tag{1}$$

where latency$_j$ and distance$_j$ give the latency and dependence distance of the j^{th} recurrence.

Moreover, because there are five r-operations, the II also cannot be less than $5/\mathrm{FUs}(r)$, where $\mathrm{FUs}(r)$ is the number of functional units that can execute operations of type r. This is called the *resource-constrained minimum II*:

$$\mathrm{II}^{\perp}_{\mathrm{res}} = \max_{r \in \text{resources}} \left(\#r / \mathrm{FUs}(r)\right) \tag{2}$$

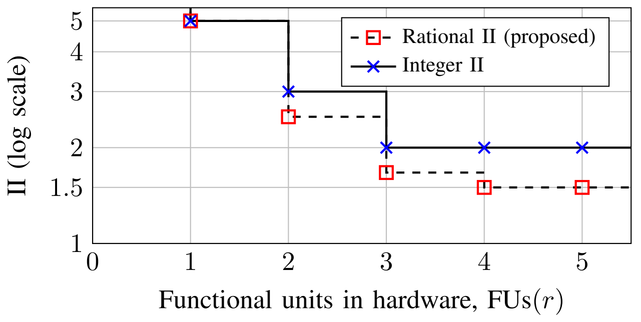


Fig. 2: Comparing the throughput obtained for the DFG of Figure 1b using rational-II and integer-II scheduling. Points nearer the bottom of the graph represent higher throughput.

TABLE I: Comparing integer-II and rational-II schedules for Figure 1b when $\mathrm{FUs}(r) = 3$. The tables assign each operation (first 9 samples) to a clock cycle and a functional unit (FU). We write $n{:}o_i$ for operation o_i on sample n. The thick borders pick out the fourth sample. The ▷ symbol indicates a clock cycle that starts a new sample.

clock cycle		**(a)** Integer II = 2				**(b)** Rational II = $\frac{5}{3}$		
		FU1	FU2	FU3		FU1	FU2	FU3
0	▷	0:o_0			▷	0:o_0		
1		0:o_1				0:o_1		
2	▷	1:o_0	0:o_2		▷	0:o_2	1:o_0	
3		1:o_1	0:o_3			0:o_3	1:o_1	
4	▷	2:o_0	1:o_2	0:o_4	▷	0:o_4	1:o_2	2:o_0
5		2:o_1	1:o_3		▷	3:o_0	1:o_3	2:o_1
6	▷	3:o_0	2:o_2	1:o_4		3:o_1	1:o_4	2:o_2
7		3:o_1	2:o_3		▷	3:o_2	4:o_0	2:o_3
8	▷	4:o_0	3:o_2	2:o_4		3:o_3	4:o_1	2:o_4
9		4:o_1	3:o_3		▷	3:o_4	4:o_2	5:o_0
10	▷	5:o_0	4:o_2	3:o_4	▷	6:o_0	4:o_3	5:o_1
11		5:o_1	4:o_3			6:o_1	4:o_4	5:o_2
12	▷	6:o_0	5:o_2	4:o_4	▷	6:o_2	7:o_0	5:o_3
13		6:o_1	5:o_3			6:o_3	7:o_1	5:o_4
14	▷	7:o_0	6:o_2	5:o_4	▷	6:o_4	7:o_2	8:o_0
15		7:o_1	6:o_3				7:o_3	8:o_1
16	▷	8:o_0	7:o_2	6:o_4			7:o_4	8:o_2
17		8:o_1	7:o_3					8:o_3
18			8:o_2	7:o_4				8:o_4
19			8:o_3					
20				8:o_4				

where $\#r$ is the number of operations of type r in the DFG.

Figure 2 shows the ideal performance of our proposed approach on Figure 1b compared to optimal integer-II solutions, over all possible resource allocations. In all cases except $\mathrm{FUs}(r) = 1$, our approach leads to improved throughput, reaching 33% when $\mathrm{FUs}(r)$ is 4 or 5.

To be more concrete, Table I shows one possible outcome of scheduling our example graph when $\mathrm{FUs}(r) = 3$, using both integer and rational IIs. In the integer-II case, one sample is inserted every two clock cycles, while in the rational-II case, three samples are inserted every five cycles.

There are four observations worth making here.

- The integer-II schedule requires 2 clock cycles more to process 9 samples completely.

TABLE II: Modulo reservation tables (MRTs) of the integer-II (left) and rational-II (right) schedules in Table I

	0	1
FU1	o_0	o_1
FU2	o_2	o_3
FU3	o_4	-

	0	1	2	3	4
FU1	o_0	o_1	o_2	o_3	o_4
FU2	o_3	o_4	o_0	o_1	o_2
FU3	o_1	o_2	o_3	o_4	o_0

- In the integer-II schedule, FU3 is idle half of the time; the rational-II schedule keeps all FUs 100% utilised.
- In the rational-II schedule, all operations on one sample can be performed by the same FU (see the thick borders in Table I); however, it should be noted that this is only one of many possible resource bindings.
- The rational-II schedule is *uniform*, in that all the samples follow the same schedule – they are merely offset by having different start times (and bound to different FUs).

Resource usage can also be visualized using *modulo reservation tables* (MRTs) [14], as shown in Table II. Note that the number of rows is FUs(r) and the number of columns is the number of cycles after which the schedule repeats.

III. Related Work

Determining a modulo schedule under resource constraints is an NP-hard, multi-criteria optimisation problem [15], [16]. Traditionally, the problem is simplified by using the *integer-II* modulo scheduling framework [1] which is a simpler version of the *rational-II* modulo scheduling framework that was published later in [3]. Within these frameworks, search strategies can be classified as either *heuristic* or *optimal*.

A. Integer-II Modulo Scheduling

Although non-iterative approaches have been investigated [17], most state-of-the-art methods in integer-II modulo scheduling utilize constant candidate IIs with the objective of minimizing the sample latency. Here, ILP-based schedulers are the state-of-the-art in latency-optimal scheduling. Heuristic approaches, often based on systems of difference constraints (SDC), drop the ability to determine latency-optimal schedules in order to reduce solving times [18].

A comparison of ILP-based integer-II modulo schedulers [19] suggests that the Eichenberger–Davidson (ED97) [20] and Moovac (MV) [21] formulations represent the state-of-the-art. Early heuristic approaches aimed to reduce solving time [15], [22] or to lower costs for lifetime storage [23], [24]. Using MRTs (see e.g. [13]), genetic algorithms [25] and graph-based approaches [4], [26], [27], even faster solving times have been achieved. The SDC-based modulo scheduling algorithm (MSDC) proposed by Canis et al. [13] can be considered as the state-of-the-art in heuristic integer-II modulo scheduling, since it is implemented in the widely used HLS tool *LegUp* [14] and is competitive with latency-optimal schedulers in terms of finding schedules for the given II [21].

B. Rational-II Modulo Scheduling

Fimmel and Müller first considered the use of rational IIs in their work on modulo scheduling in compilers for VLIW architectures [3]. We bring their ideas to hardware design, and also address several shortcomings of their formulation:

- Their formulation only applies when $\mathrm{II}^{\perp}_{\mathrm{res}} < \mathrm{II}^{\perp}_{\mathrm{rec}}$. In Section VIII-B, we show that actually this assumption is one that rarely holds.
- Their formulation involves finding solutions to a *mixed-ILP* problem. Compared to our proposed approaches, this results in longer solving times and fewer optimal schedules being found, as shown in Section VIII-C.
- Their approach includes no strategy to deal with solver timeouts. Whenever no solution was found, there is no strategy to obtain a schedule at all.

The first two points are addressed in Sections IV and V where we propose two different ILP-based approaches that significantly outperform Fimmel–Müller in terms of problems solved and throughput achieved. As for the third point, we propose the first formulation for heuristic rational-II modulo scheduling in Section VI and the first framework for iterating over rational-IIs in modulo scheduling in Section VII. Analogous to the success of integer-II modulo scheduling in such fields as software pipelining and HLS, we believe that progress on heuristics and fallback strategies is required to enable the potential of rational-II modulo scheduling to optimise throughput in a broad range of applications.

IV. Rational-II Scheduling

In this section, we propose a general ILP formulation to solve the rational-II modulo scheduling problem. Compared to the uniform formulation that will be introduced in Section V, this formulation is a more general approach that is able to solve more scheduling problems optimally w.r.t. II (as shown in Section IV-D). Our approach is a strictly linear program, since all non-linear functions used in the following do not act over decision variables.

First, we present some preliminaries and useful definitions, and highlight the main differences between integer-II and rational-II modulo scheduling (Sections IV-A and IV-B). All constants and variables used are listed in Table III. Our proposed ILP formulation is described in Section IV-C. In contrary to the motivating example, this approach does not guarantee uniform schedules, but more problems can be solved for the optimal II, which we highlight in Section IV-D.

A. Integer and Rational Minimum II

The recurrence-constrained and the resource-constrained minimum II, as defined in (1) and (2), provide lower bounds for the II. The *integer* minimum II is defined as

$$\mathrm{II}^{\perp}_{\mathbb{N}} = \max(\lceil \mathrm{II}^{\perp}_{\mathrm{res}} \rceil, \lceil \mathrm{II}^{\perp}_{\mathrm{rec}} \rceil) \ . \tag{3}$$

The *rational* minimum II, on the other hand, avoids the ceiling functions. It can be defined as

$$\mathrm{II}^{\perp}_{\mathbb{Q}} = \max(\mathrm{II}^{\perp}_{\mathrm{res}}, \mathrm{II}^{\perp}_{\mathrm{rec}}) \ . \tag{4}$$

TABLE III: A glossary of constants (top) and variables (bottom) for resource-constrained rational-II modulo scheduling

Constant/Variable	Meaning
$o_i \in O$	Set of operations in the DFG
$(o_i, o_j) \in E$	Set of edges in the DFG
$d_{ij} \in \mathbb{N}_0$	Dependence distance on edge $o_i \to o_j$
R	Set of resource-constrained operation types
$\check{O} \subseteq O$	Set of resource-constrained operations
$\check{O}_r \subseteq \check{O}$	Set of resource-constrained operations of type $r \in R$, i.e., $\bigcup_{r \in R} \check{O}_r = \check{O}$
$\text{FUs}(r) \in \mathbb{N}$	No. of hardware instances of resource type $r \in R$
$D_i \in \mathbb{N}_0$	Latency of operation $o_i \in O$
$M \in \mathbb{N}$	No. of cycles before the modulo schedule repeats
$S \in \mathbb{N}$	No. of samples inserted every M cycles
$S^\perp$, $M^\perp$	Values for S, M that lead to $\text{II}_\mathbb{Q}^\perp$
$0 \le s \le S-1$	Range of sample indices
$\text{II}_\mathbb{Q} = \frac{M}{S}$	Rational initiation interval
$L \in \mathbb{N}_0$	Maximal latency constraint
$i_{\max}$	Number of iterations steps allowed
$t_{i,s} \in \mathbb{N}_0$	Start time of o_i on sample s
t_v	Virtual node
$b_{i,s,\tau}$	True iff τ is the start time of $o_i \in \check{O}$ on sample s
$\langle \text{II}_0 \ldots \text{II}_{S-1} \rangle$	Latency sequence
$I_s \in \mathbb{N}_0$	Insertion time of sample s

It follows that $\text{II}_\mathbb{N}^\perp = \lceil \text{II}_\mathbb{Q}^\perp \rceil$, and hence that rational-II schedules will always attain a throughput that is at least as good as integer-II schedules. When $\text{II}_\mathbb{Q}^\perp$ is already an integer, we have $\text{II}_\mathbb{Q}^\perp = \text{II}_\mathbb{N}^\perp$, and switching to rational-II scheduling cannot improve throughput (speedup = 1). This situation can be identified quickly before scheduling, and standard integer-II algorithms can be applied. The *maximum* speedup is obtained when $\text{II}_\mathbb{Q}^\perp = 1+\epsilon$ for small, positive ϵ. In this case, the speedup is $\frac{\lceil 1+\epsilon \rceil}{1+\epsilon}$, which tends towards 2. Overall, we have:

$$1 \le \text{speedup} < 2\ . \tag{5}$$

In our experiments (Section VIII), we observe that potential speedups are indeed widely spread from 1 up to 1.99.

B. Problem Specification

We consider the input to be a DFG (O, E) where operations $o_i \in O$ that have a latency in clock cycles (D_i) are connected by directed edges $(o_i, o_j) \in E$. We write $\check{O}_r$ for the set of operations that require resource type r (adder, etc.). The number of available functional units of type r is $\text{FUs}(r)$.

As in most state-of-the-art integer-II modulo scheduling formulations, we consider the II to be a constant input to the ILP problem, calculated using (4). We write II in the form $\frac{M}{S}$, where M is the number of cycles before the insertion sequence repeats, and S is the number of samples inserted every M cycles. Each operation o_i gets assigned S different clock cycles, $t_{i,0}, \ldots, t_{i,S-1}$, where $t_{i,s}$ holds the cycle in which operation o_i is operating on sample s. Therefore, the number of time variables and resource constraints increases linearly with S. This makes ILP-based rational-II modulo scheduling more complex than integer-II modulo scheduling where the number of variables increases quadratically with the number of time variables and resource constraints [18], [21]. In Section VII, we address this problem by picking small S values, aiming to reduce the complexity of the ILP problem while maintaining high throughput.

$$\min(t_v)$$

subject to

$$\textbf{D1}:\ t_{i,\tilde{s}} + D_i - \delta(S, s, d_{i,j}) \le t_{j,s} \qquad \forall s, \forall i,j : (o_i, o_j) \in E$$

$$\text{where } \tilde{s} = (s - d_{i,j}) \bmod S$$

$$\textbf{D2}:\ t_i + D_i \le t_v \qquad \forall i : o_i \in O$$

$$\textbf{D3}:\ t_v \le L$$

$$\textbf{R1}:\ \sum_{0 \le \tau < M} \tau \cdot b_{i,s,\tau} + M \cdot k_{i,s} = t_{i,s} \qquad \forall r, \forall s, \forall i : o_i \in \check{O}_r$$

$$\textbf{R2}:\ \sum_{0 \le \tau < M} b_{i,s,\tau} = 1 \qquad \forall s, \forall r, \forall i : o_i \in \check{O}_r$$

$$\textbf{R3}:\ \sum_{0 \le s < S} \sum_{i : o_i \in \check{O}_r} b_{i,s,\tau} \le \text{FUs}(r) \quad \forall r, \forall \tau < M$$

Fig. 3: An ILP for (non-uniform) rational-II scheduling

C. ILP Formulation

The general problem of rational-II modulo scheduling is formulated in Figure 3.

Constraint **D1** is a variation on the standard causality constraint from integer-II modulo scheduling [20], [21]:

$$t_i + D_i - d_{i,j} \cdot \text{II} \le t_j \qquad \forall i,j : (o_i \to o_j) \in E \tag{6}$$

which states that the start time t_j of operation o_j must not precede the end time of operation o_i from $d_{i,j}$ samples ago. Note that for intra-sample dependency, we have $d_{i,j} = 0$. The dependence distance $d_{i,j}$ is multiplied by II because this is the number of cycles between successive samples. As an example, consider the integer-II schedule from Table I(a), and the edge from o_3 to o_0 in Figure 1b. We have $t_3 = 3$, $t_0 = 0$, $D_3 = 1$, $d_{3,0} = 2$, and II = 2, so (6) holds in this instance.

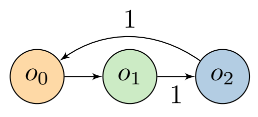
Compared to the integer-II version, our **D1** takes into account the partial unrolling of the DFG, which causes connections and dependence distances to change. The way we model unrolling in our ILP formulation is described in the following. The distance is calculated using:

$$\delta(S, s, d) = \max\left(0, \left\lceil \frac{d-s}{S} \right\rceil\right) \tag{7}$$

which becomes 0 whenever causality can now be modelled within the unrolled DFG ($d - s \le 0$). The origin of edges ($o_{i,\tilde{s}}$) is calculated according to the equation attached to **D1**. A distance of 0 leads to $\tilde{s} = s$, i.e., edges within their respective samples connect the same vertices as in the original version. After unrolling, weighted edges in the original DFG may have a source vertex that represents other samples than the target vertex. Section IV-D gives an example of this procedure.

```
for (i=1; i<=100; i++) {
  A[i] = r(C[i-1]); //o_0
  B[i] = r(A[i]);   //o_1
  C[i] = r(B[i-1]); //o_2
}
```

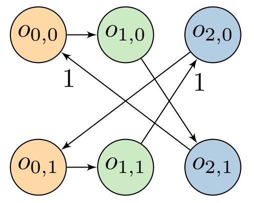
(a) Source code



(b) DFG of the loop body

```
for (i=1; i<=50; i++) {
  A[2*i-1] = r(C[2*i-2]); //o_{0,0}
  B[2*i-1] = r(A[2*i-1]); //o_{1,0}
  C[2*i-1] = r(B[2*i-2]); //o_{2,0}
  A[2*i] = r(C[2*i-1]);   //o_{0,1}
  B[2*i] = r(A[2*i]);     //o_{1,1}
  C[2*i] = r(B[2*i-1]);   //o_{2,1}
}
```

(c) Partially unrolled source code



(d) DFG of the unrolled loop body

Fig. 4: An example to motivate non-uniform scheduling

TABLE IV: Comparing integer-II and rational-II schedules for the example in Figure 4 when FUs$(r) = 2$.

clock cycle		**(a)** Integer II = 2 FU1	**(a)** Integer II = 2 FU2		**(b)** Rational II = $\frac{3}{2}$ FU1	**(b)** Rational II = $\frac{3}{2}$ FU2
0	▷	0:o_0	0:o_2	▷	0:o_0	0:o_2
1		0:o_1		▷	0:o_1	1:o_0
2	▷	1:o_0	1:o_2		1:o_1	1:o_2
3		1:o_1		▷	2:o_0	2:o_2
4	▷	2:o_0	2:o_2	▷	2:o_1	3:o_0
5		2:o_1			3:o_1	3:o_2
6	▷	3:o_0	3:o_2	▷	4:o_0	4:o_2
7		3:o_1			4:o_1	
8		4:o_0	4:o_2			
9		4:o_1				

The objective of the ILP is to minimize the latency of each sample. Following Cong and Zhang [28], we add a 'virtual' node v with a start time of t_v. Constraints **D2** and **D3** ensure that v is scheduled after all nodes have finished processing. Minimising the start time of the virtual node is then the same as minimising the latency of S samples produced.

To enforce that the number of FUs used does not exceed FUs(r) we use binary variables. Following Eichenberger et al. [20], $b_{i,s,\tau}$ in **R1** is true if and only if $t_{i,s} \mod M = \tau$. Constraint **R2** ensures that one (and only one) $b_{i,s,\tau}$ is true for each $t_{i,s}$. This information is then used in **R3** to ensure that the upper limit FUs(r) for each resource type r is respected. The inner sum in **R3** adds up all the uses of resource r in clock cycle $m \bmod M$. This is done for all samples, thus forcing the schedule to never use more than FUs(r) in any clock cycle.

D. The need for non-uniform scheduling

We now give an example that takes advantage of the non-uniform schedules that the formulation given in this section allows. In the next section, we will restrict our attention to uniform schedules, and thus will not be able to find schedules for examples like this one.

The DFG shown in Figure 4b describes the body of the for-loop in Figure 4a. Operations o_0, o_1, and o_2 require the same resource r that has a latency of one cycle. Let FUs$(r) = 2$. From (4), it follows that $\text{II}^{\perp}_{\mathbb{Q}} = \max(\frac{3}{2}, \frac{3}{2}) = \frac{3}{2}$. Figure 4c shows the source after unrolling the loop by a factor of $S = 2$. In Figure 4d, all vertices from Figure 4b are inserted twice ($S = 2$). Edges are inserted using **D1** in combination with (7). The unweighted edge (o_0, o_1) generates two unweighted edges $(o_{0,0}, o_{1,0})$ and $(o_{0,1}, o_{1,1})$. Weighted edges that have a target vertex with $s = 1$ now originate from a source vertex with $s = 0$ and have a distance of 0. Weighted edges that have a vertex with $s = 0$ still carry a weight of 1, but they now originate from a vertex with $s = 1$.

One integer-II solution for $\text{II}^{\perp}_{\mathbb{N}} = 2$ is shown in Table IV, together with the only solution for $\text{II}^{\perp}_{\mathbb{Q}} = \frac{3}{2}$. In the integer-II case, FU2 is only busy 50% of the time, while in the rational-II case all FUs are occupied in every clock cycle. The rational-II schedule is non-uniform, because on the first sample, operations o_0 and o_2 are scheduled together, but on the second sample, o_1 and o_2 coincide.

V. Uniform Rational-II Scheduling

In the previous section, we introduced rational-II modulo scheduling. It aims to find the optimal II by including in its search space *non-uniform* schedules (in which operation on different samples are scheduled independently). We now describe an alternative that is simpler and demands fewer variables, but misses some optimal solutions: *uniform* rational-II modulo scheduling. This section is based on the approach laid out in our conference paper [6].

A. Sequential Sample Insertion

From the motivating example in Section II, we learn that optimal throughput using rational IIs can be achieved using uniformly scheduled samples with alternating insertion times. To model this, we assign every sample $s, s < S$, an insertion time I_s modulo M. In Table I, where II $= \frac{5}{3}$, we have $I_0 = 0$, $I_1 = 2$, and $I_2 = 4$. This means that for all $n \geq 0$, we have sample $3n$ inserted at cycle $5n$, sample $3n+1$ inserted at cycle $5n + 2$, and sample $3n + 2$ inserted at cycle $5n + 4$. We fix the first insertion time to 0.

The repeating sequence of insertions lets us calculate the latency in clock cycles between successive samples. For this, we use *latency sequences* [29], which take the form

$$\langle \text{II}_0 \; \text{II}_1 \; ... \; \text{II}_{S-1} \rangle \tag{8}$$

where

$$\text{II}_s = \begin{cases} I_{s+1} - I_s & \text{if } s < S - 1 \\ M - I_s & \text{if } s = S - 1 \end{cases} . \tag{9}$$

For instance, the sample insertion times from the example in Section II lead to a latency sequence of $\langle 2\ 2\ 1 \rangle$. This yields a modulo-5 schedule where new samples will be inserted in every $0, 2, 4, 5, 7, ...$ cycles. Note that integer IIs correspond to latency sequences of length 1, such as $\langle 3 \rangle$.

B. Causality

The introduction of latency sequences means that the number of cycles between successive samples can vary, depending on the sample index, s. In integer-II scheduling, this can be calculated as $\mathrm{II} \cdot d_{i,j}$ since data insertion is spaced equally. Assuming a latency sequence $\langle \mathrm{II}_0\ \mathrm{II}_1\ ...\ \mathrm{II}_{S-1} \rangle$, the number of cycles between sample s and $s-d$ can be calculated as

$$\Delta_s(d) = \sum_{n=1}^{d} \mathrm{II}_{(s-n) \bmod S} \ . \tag{10}$$

Starting at sample s, the calculation steps backwards through the latency sequence, adding up the last d latencies. Thus, the causality constraint becomes

$$t_{i,s} + D_i - \Delta_s(d_{i,j}) \leq t_{j,s} \quad \forall s, \forall i,j : (o_i, o_j) \in E \ . \tag{11}$$

As an example, consider the rational-II schedule from Table I(b), and the edge from o_3 to o_0. When $s = 0$, we have $t_{3,s} = 3$, $t_{0,s} = 1$, $D_3 = 1$, $d_{3,0} = 2$, and $\Delta_s(2) = 3$, so (11) holds. It also holds for $s = 1$ and $s = 2$. However, with the *different* latency sequence $\langle 1\ 1\ 3 \rangle$ obtained by shifting the FU2 column in Table I(b) up by one cycle and the FU3 column up by two, we would get $\Delta_s(2) = 2$, and hence (11) would be violated – the third sample is being inserted too soon.

The smallest number of clock cycles between successive samples imposes the strongest causality constraints for the scheduler. We define this value as

$$\Delta_{\text{min}}(d) = \min_{s \in S}(\Delta_s(d)) \tag{12}$$

and use it to determine feasible latency sequences before scheduling to speed up solving. This is described next.

C. Determining latency sequences

We now show how feasible latency sequences can be obtained. This significantly reduces the number of variables and constraints required in the ILP formulation. The method presented in this section will enable our novel rational-II modulo scheduling heuristic that is described in Section VI.

The intuition of our approach is to generate latency sequences that are as 'regular' as possible. Consider an example where $\mathrm{II}_{\mathbb{Q}}^{\perp} = \frac{18}{5}$ and $\mathrm{II}_{\mathbb{N}}^{\perp} = 4$. Then intuitively, the latency sequence $a = \langle 1\ 1\ 6\ 1\ 9 \rangle$ imposes stronger causality constraints than our preferred sequence $b = \langle 4\ 4\ 3\ 4\ 3 \rangle$. This is because for a we have $\Delta_{\text{min}}(1) = 1$ and for b we have $\Delta_{\text{min}}(1) = 3$. The integer-II latency sequence $c = \langle 4 \rangle$ is even more relaxed than b, but it has lower throughput. This situation is displayed in Figure 5, which shows the indices of samples inserted on the x-axis and their insertion clock cycle on the y-axis. One can see that, at first, samples get inserted more quickly using a, but b catches up every 5 samples. This has to be the case, as both sequences support an II of $\frac{18}{5}$.

Figure 5 provides the intuition that obtaining regular latency sequences is equivalent to generating a stepped line between $(0,0)$ and (S,M) that is as straight as possible. Hence our approach of generating these sequences, given as pseudocode in Algorithm 1, is inspired by Bresenham's algorithm [30].

To begin generating the desired latency sequence, we calculate the ceiling (II_C) and the floor (II_F) of $\frac{M}{S}$ (line 1). Using II_C, II_F, S and M, we calculate (line 2) how many times II_C and II_F occur ($\#\mathrm{II}_C$ and $\#\mathrm{II}_F$). In our example, we have $S = 5$, $M = 18$, $\mathrm{II}_C = 4$, and $\mathrm{II}_F = 3$. This means

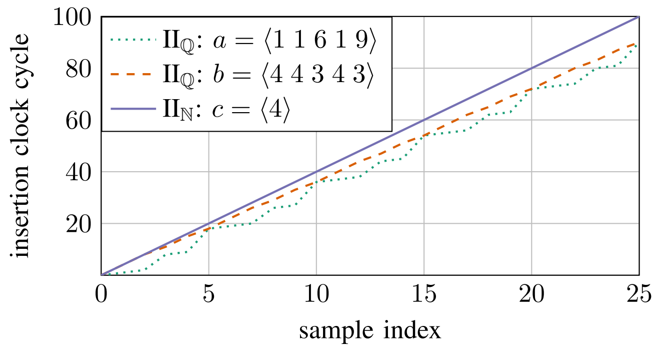


Fig. 5: Two latency sequences for $\mathrm{II}_{\mathbb{Q}} = \frac{18}{5}$. Both require 90 clock cycles to process 25 samples (whereas the integer-II schedule requires 100 clock cycles), but b is more 'regular' and hence the causality constraint is easier to satisfy.

Algorithm 1 Generating regular latency sequences

```
Require: $S$, $M$
Ensure: One $\mathrm{II}_s$ that maximizes $\Delta_{min}(d)\ \forall\ d \in \mathbb{N}_{\geq 0}$
 1: $\mathrm{II}_C \leftarrow \lceil M/S \rceil$; $\mathrm{II}_F \leftarrow \lfloor M/S \rfloor$
 2: $\#\mathrm{II}_F \leftarrow \mathrm{II}_C \cdot S - M$; $\#\mathrm{II}_C \leftarrow M - \mathrm{II}_F \cdot S$
 3: $s_1, s_2 \leftarrow \mathrm{II}_F, \mathrm{II}_C$
 4: $k_{\text{min}}, k_{\text{max}} \leftarrow \#\mathrm{II}_C, \#\mathrm{II}_F$
 5: if $\#\mathrm{II}_F < \#\mathrm{II}_C$ then
 6:     $s_1, s_2 \leftarrow \mathrm{II}_C, \mathrm{II}_F$
 7:     $k_{\text{min}}, k_{\text{max}} \leftarrow \#\mathrm{II}_F, \#\mathrm{II}_C$
 8: $E \leftarrow 0$, $\mathrm{II}_s \leftarrow \{\}$
 9: for $i \leftarrow 1$ to $k_{\text{max}}$ do
10:     $\mathrm{II}_s$.append($s_1$)
11:     $E \leftarrow E + k_{\text{min}}$
12:     if $E \geq k_{\text{max}}$ then
13:         $\mathrm{II}_s$.append($s_2$)
14:         $E \leftarrow E - k_{\text{max}}$
15: return $\mathrm{II}_s$
```

that the latency sequence comprises three occurrences of II_C ($\#\mathrm{II}_C = 3$) and two occurrences of II_F ($\#\mathrm{II}_F = 2$).

We then order the determined values such that $\Delta_{min}(d)$ is maximized for all $d \geq 0$. This keeps the causality constraints as relaxed as possible. The calculated values of $\#\mathrm{II}_C$ and $\#\mathrm{II}_F$ allow us to store the latency that occurs more often in the sequence in s_1, and the other one in s_2. We also need the frequencies of II_C and II_F, which we store in the values k_{min} and k_{max} (lines 3 to 7). In our example, we have $s_1 = 4$, $s_2 = 3$, $k_{\text{min}} = 2$ and $k_{\text{max}} = 3$. In line 8, we instantiate E, which indicates when the latency that appears less often (in our example, $\mathrm{II} = 3$) has to be inserted.

Finally, the for-loop in lines 9–14 appends the value of II_C and II_F that appears more often to II_s. Meanwhile in lines 11, 12 and 14, the threshold value E is used to check whether the less frequent value is being appended to II_s. The determined latency sequence is returned in line 15.

D. ILP Formulation

The problem of uniform rational-II modulo scheduling is formulated in Figure 6. **D1** enforces the causality constraint

$$\min(t_v)$$

subject to

$$\textbf{D1}: \quad t_i + D_i - \Delta_{min}(d_{i,j}) \leq t_j \qquad \forall i,j : (o_i, o_j) \in E$$

$$\textbf{D2}: \quad t_i + D_i \leq t_v \qquad \forall i : o_i \in O$$

$$\textbf{D3}: \quad t_v \leq L$$

$$\textbf{R1}: \sum_{0 \leq \tau < M} \tau \cdot b_{i,s,\tau} + M \cdot k_{i,s} = t_i + I_s \qquad \forall s, \forall r, \forall i : o_i \in \check{O}_r$$

$$\textbf{R2}: \quad \sum_{0 \leq \tau < M} b_{i,s,\tau} = 1 \qquad \forall s, \forall r, \forall i : o_i \in \check{O}_r$$

$$\textbf{R3}: \quad \sum_{0 \leq s < S} \sum_{i : o_i \in \check{O}_r} b_{i,s,\tau} \leq \text{FUs}(r) \qquad \forall r, \forall \tau < M$$

Fig. 6: An ILP for uniform rational-II scheduling

introduced in (10). Analogous to Section IV-C, the latency of each sample is constrained by the user-specified value L, which can be seen in **D2** and **D3**. Sequential IIs and the uniformity of schedules are enabled by **R1**: the variable I_s expresses that each of the S samples follows the same schedule (and hence that every sample is fully processed within L cycles of its insertion), except that each schedule is offset. Constraints **R2** and **R3** are the same as in Figure 3.

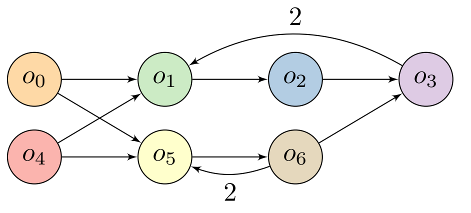
VI. Rational II Scheduling Heuristic

In Section V, we proposed an ILP formulation that can solve the *uniform* rational-II modulo scheduling problem optimally w.r.t. latency. In this section, we drop the ability to optimise latency in order to solve larger problems optimally w.r.t. II. Our heuristic approach is based on three ideas:

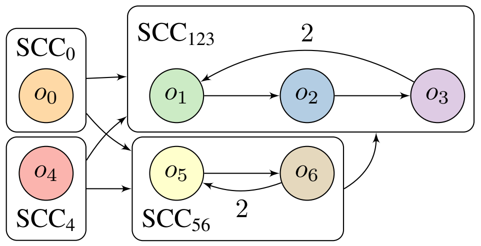
- We obtain valid start times for all operations in the first sample $s = 0$ of the rational-II schedule. Then, start times of operations in other samples directly follow from applying a latency sequence that has been obtained according to Section V-C.
- Analogous to [4], we cluster the DFG into cyclic and acyclic parts by using *strongly connected components* (SCCs) [5]. The cyclic parts of the graph are then scheduled. Note that this is faster than scheduling the complete DFG. To obtain a valid schedule for the complete DFG, the remaining acyclic parts are scheduled using an *as soon as possible* (ASAP) method.
- We introduce a novel method for generating MRT shapes to remove the contribution of S to the ILP's complexity, thus improving solving rates significantly. Although there could be corner cases where this heuristic leads to solutions being missed, we did not encounter any problem in our experiments that could not be solved.

A. Worked Example

As an example of our heuristic, consider the DFG in Figure 7a, where all vertices are of the same resource type r that has a latency of one cycle, and assume that $\text{FUs}(r) = 5$.



(a) An example data-flow graph



(b) Partitioned into SCCs

Fig. 7: Example graph and its partitioning into SCCs

TABLE V: Final MRTs ($M = 3, S = 2, \text{FUs}(r) = 5$)

(a) heuristic

	0	1	2
FU1	$o_{1,0}$	$o_{2,0}$	$o_{3,0}$
FU2	$o_{5,0}$	$o_{6,0}$	$o_{4,0}$
FU3	$o_{0,0}$	-	$o_{1,1}$
FU4	$o_{2,1}$	$o_{3,1}$	$o_{5,1}$
FU5	$o_{6,1}$	$o_{4,1}$	$o_{0,1}$

(b) optimal

	0	1	2
FU1	$o_{0,0}$	$o_{1,0}$	$o_{2,0}$
FU2	$o_{3,0}$	$o_{5,0}$	$o_{6,0}$
FU3	$o_{4,0}$	$o_{2,1}$	$o_{0,1}$
FU4	$o_{1,1}$	$o_{6,1}$	$o_{3,1}$
FU5	$o_{5,1}$	-	$o_{4,1}$

This leads to $\text{II}_{\mathbb{Q}}^{\perp} = \frac{3}{2}$ due to the recurrence among o_1, o_2, and o_3. Algorithm 1 yields the latency sequence $\langle 2\ 1 \rangle$.

In Figure 7b, we see how the original DFG is partitioned into SCCs. Following Dai and Zhang [4], we classify SCCs as *trivial*, *complex* or *basic*. SCCs that contain only one vertex are called *trivial*. SCCs that contain at least one vertex that uses a limited resource are called *complex*. Other SCCs are called *basic*. This partitioning yields two trivial SCCs (SCC$_0$ and SCC$_4$) and two complex SCCs (SCC$_{123}$ and SCC$_{56}$).

To obtain start times for one sample, we partition the MRT of each limited resource into S parts. Such a partitioning is shown in Table Va, where two samples use 14 of the 15 slots. We call an MRT that contains operations of one sample an *intermediate* MRT and one that contains all samples the *final* MRT. Using the concept of intermediate MRTs, we first assign start times for one sample in order to fill the final MRT later. This requires two steps:

1) Obtain a 'relative' schedule for all operations in non-trivial SCCs using the intermediate MRT. (To describe such a schedule, we use $\hat{t_i}$ for operation o_i.)
2) Visit every vertex of the SCC-based DFG (see Figure 7b) using breadth-first search. For trivial SCCs, insert the operation into the intermediate MRT and commit the start time to the final schedule (ASAP). Since modulo slots for operations in non-trivial SCCs are fixed due to step (1), time steps are committed to the final schedule by

TABLE VI: Filling intermediate MRT slots ($M = 3$)

(a)

0	1	2
o_1	o_2	o_3
o_5	o_6	-
-	X	X

(b)

0	1	2
o_1	o_2	o_3
o_5	o_6	-
o_0	X	X

(c)

0	1	2
o_1	o_2	o_3
o_5	o_6	o_4
o_0	X	X

offsetting relative start times by multiples of M.

For our example, a valid solution to step 1 is: $\hat{t}_1 = 0$, $\hat{t}_2 = 1$, $\hat{t}_3 = 2$, $\hat{t}_5 = 0$ and $\hat{t}_6 = 1$. These times lead to the intermediate MRT in Table VIa, in which free ('-') and unavailable ('X') slots are highlighted.

Topologically sorting the SCC graph (step 2) leads to the order $\langle$SCC$_0$, SCC$_4$, SCC$_{56}$, SCC$_{123}\rangle$. SCC$_0$ is trivial and is scheduled ASAP. Vertex o_0 does not have any incoming edges and MRT slot 0 is free, so it is scheduled in time $t_0 = 0$. The intermediate MRT after scheduling o_0 is in Table VIb. SCC$_4$ is also trivial. Vertex o_4 also has no incoming edges but modulo slots 0 and 1 are already full. It follows that o_4 is scheduled at time $t_4 = 2$. The intermediate MRT after committing all operations of one sample is in Table VIc.

SCC$_{56}$ is non-trivial and operation o_5 must start after o_0. This means that SCC$_{56}$ cannot be scheduled with an offset of 0 clock cycles. For this example, the minimum offset is 3 clock cycles, which leads to $t_5 = 3$ and $t_6 = 4$. Note that this does not change the MRT because relative start times of the vertices of SCC$_{56}$ are shifted by an offset that is a multiple of $M = 3$. Finally, SCC$_{123}$ is scheduled in the same way as SCC$_{56}$. The minimum offset is 3 clock cycles, which leads to $t_1 = 3$, $t_2 = 4$ and $t_3 = 5$. The final MRT after committing all operations to the intermediate MRT is shown in Table Va, where $o_{i,j}$ stands for operation i of sample j. Table Vb shows an example MRT for a latency-optimal schedule obtained by the ILP formulation from Section V.

It is interesting that intermediate MRTs can be modelled in many different ways as long as no more than FUs(r) are committed to each time step modulo M. Different intermediate MRT shapes may lead to different outcomes regarding latency, resource usage and maximum frequency. Future work will investigate other heuristic approaches as well as including intermediate MRT shape generation into the ILP formulation.

The final schedule is shown in Table VIIa, which gives the start times of 6 consecutive samples obtained by the heuristic described in this section. Note that our heuristic does not find a latency-optimal solution in this case. For comparison, a latency-optimal schedule is given in Table VIIb (which corresponds to the MRT in Table Vb).

B. General Procedure

We now discuss how our heuristic works in general. Pseudocode for our procedure is given in Algorithm 2. As input, we have the DFG G, the number of available FUs for each resource, the desired values for S and M, and a time value after which scheduling should abort. First, a latency sequence is determined using Algorithm 1 (Line 1). Partitioning of the DFG into SCCs is done in Line 2 via Tarjan's SCC algorithm [5]. In line 3, non-trivial SCCs are combined to obtain a relative schedule that is determined using the ILP formulation given in Figure 8.

TABLE VII: Comparing rational-II schedules for the example graph in Figure 7a when FUs(r) = 5. The thick borders highlight suboptimal (left) and optimal (right) sample latencies that are obtained using the proposed heuristic from Section VI and the proposed ILP from Section V, respectively.

(a) Sample latency = 6

clock cycle		FU1	FU2	FU3	FU4	FU5
0	▷			0:o_0		
1						
2	▷		0:o_4			1:o_0
3	▷	0:o_1	0:o_5	2:o_0		
4		0:o_2	0:o_6		1:o_4	
5	▷	0:o_3	2:o_4	1:o_1	1:o_5	3:o_0
6	▷	2:o_1	2:o_5	4:o_0	1:o_2	1:o_6
7		2:o_2	2:o_6	1:o_3	3:o_4	
8	▷	2:o_3	4:o_4	3:o_1	3:o_5	5:o_0
9		4:o_1	4:o_5		3:o_2	3:o_6
10		4:o_2	4:o_6	3:o_3	5:o_4	
11		4:o_3		5:o_1	5:o_5	
12					5:o_2	5:o_6
13				5:o_3		

(b) Sample latency = 4

clock cycle		FU1	FU2	FU3	FU4	FU5
0	▷	0:o_0		0:o_4		
1		0:o_1	0:o_5			
2	▷	0:o_2	0:o_6	1:o_0		1:o_4
3	▷	2:o_0	0:o_3	2:o_4	1:o_1	1:o_5
4		2:o_1	2:o_5	1:o_2	1:o_6	
5	▷	2:o_2	2:o_6	3:o_0	1:o_3	3:o_4
6	▷	4:o_0	2:o_3	4:o_4	3:o_1	3:o_5
7		4:o_1	4:o_5	3:o_2	3:o_6	
8	▷	4:o_2	4:o_6	5:o_0	3:o_3	5:o_4
9			4:o_3		5:o_1	5:o_5
10				5:o_2	5:o_6	
11					5:o_3	
12						
13						

$$\min(t_v)$$

subject to

$$\begin{aligned}
&\textbf{D1}: && t_i + D_i - \Delta_{min}(d_{i,j}) \le t_j && \forall i,j : (o_i, o_j) \in E \\
&\textbf{D2}: && t_i + D_i \le t_v && \forall i : o_i \in O \\
&\textbf{D3}: && t_v \le L && \\
&\textbf{R1}: && \sum_{0 \le \tau < M} \tau \cdot b_{i,\tau} + M \cdot k_i = t_i && \forall r, \forall i : o_i \in \check{O}_r \\
&\textbf{R2}: && \sum_{0 \le \tau < M} b_{i,\tau} = 1 && \forall r, \forall i : o_i \in \check{O}_r \\
&\textbf{R3}: && \sum_{i : o_i \in \check{O}_r} b_{i,\tau} - \mathtt{h}(|\check{O}_r|, M, \tau) \le 0 && \forall r, \forall \tau < M
\end{aligned}$$

Fig. 8: An ILP problem for scheduling SCCs

We use **D1**–**D3** for dependency and recurrence constraints. Resource constraints are enforced by **R1**–**R3**. Following Eichenberger [20], we use binary variables and MRTs.

However, we propose a heuristic that models intermediate MRTs in order to reduce solving time. To do this, we use non-rectangular intermediate MRTs (see Table VI). Intuitively, the number of operations allowed per modulo slot in these intermediate MRTs can be seen as the *height*. To calculate this height for each $0 \le \tau < M$, we define an $\mathtt{h}$ function:

$$\mathtt{h}(n, M, \tau) = \begin{cases} \left\lceil \frac{n}{M} \right\rceil & \tau = 0 \\ \mathtt{h}(n - \left\lceil \frac{n}{M} \right\rceil, M-1, \tau - 1) & \tau > 0. \end{cases} \quad (13)$$

For our example ($n = |\check{O}_r| = 7, M = 3, 0 \le \tau < 3$), we get $\mathtt{h}(7,3,0) = 3$, $\mathtt{h}(7,3,1) = 2$ and $\mathtt{h}(7,3,2) = 2$ which is the number of operations that are allowed in the respective modulo slots of the intermediate MRT shown in Table VI.

Algorithm 2 Rational II scheduling heuristic

```
Require: $G$, FUs, $S$, $M$, $time$
Ensure: A schedule
 1: $\mathrm{II}_s \leftarrow \texttt{sequence}(S, M)$                //Algorithm 1
 2: partition $G$ into SCCs
 3: get relative schedule for non-trivial SCCs
 4: fill MRT with start times of complex SCCs
 5: SCCs $\leftarrow$ topological sort of SCCs
 6: for each SCC in SCCs do
 7:   if SCC is $trivial$ then
 8:     schedule ASAP respecting intermediate MRT
 9:     insert vertex of SCC into intermediate MRT
10:   else
11:     $scheduled \leftarrow false$
12:     $T \leftarrow 0$
13:     while $scheduled = false$ do
14:       $scheduled \leftarrow true$
15:       if $time \leq 0$ then
16:         return {}
17:       for each vertex $v_i$ in vertices of SCC do
18:         try scheduling $v_i$ in time slot $t_i + T$
19:         if dependency constraint violated then
20:           $scheduled \leftarrow false$
21:           $T \leftarrow T + M$
22:           break
23: return final schedule using $\mathrm{II}_s$
```

In line 4 of Algorithm 2, modulo slots for operations inside non-trivial SCCs are fixed. Relative start times are fixed, but can be delayed by an offset that is an integer multiple of M. Now, the partitioned graph can be scheduled in topological order and relative start times can be offset and committed to the final schedule. This is done in lines 5–22.

Trivial SCCs are handled in lines 7–9 and can be scheduled as soon as possible, given a free MRT slot. Non-trivial SCCs are committed to the final schedule (in lines 13–22) while respecting the MRT slots that are already fixed. In line 15, we check whether the user-specified timeout has been exceeded. If that is the case, we return an empty schedule in line 16. Using the relative schedule, the committing of non-trivial SCCs is done by determining the minimal offset that fulfils dependency constraints. In the first iteration of the while-loop in line 12 ($T \leftarrow 0$), the algorithm tries to commit the relative schedule of the SCC to the final schedule without an offset. If this would violate a dependency constraint (line 19), T is incremented by M and in the next iteration the algorithm tries to commit all operations in the SCC to the final schedule using the updated offset. The final schedule of all samples is then determined using the calculated latency sequence and returned in line 23.

VII. Iterating over Rational-IIs

The approaches proposed in the previous sections can be used to find a schedule with the smallest-possible rational II. However, since scheduling is an NP-hard problem, the ILP solver may timeout before it finds a solution. Therefore, we now present an algorithm for making repeated attempts with increasing IIs, which can be used by any scheduler that accepts rational candidate IIs. The larger the II, the less constrained the problem becomes; the drawback is that the further we deviate from the ideal $\mathrm{II}_{\mathbb{Q}}^{\perp}$, the lower the throughput will be. For the specific case of *rational* IIs, which we write in the form M/S, we have already noted that the number of constraints and variables in the ILP problem increases with S. So the aim of our iterative process is not only to gradually increase M/S, but also to keep S small. To indicate the minimum II, we write $\mathrm{II}_{\mathbb{Q}}^{\perp} = M^{\perp}/S^{\perp}$.

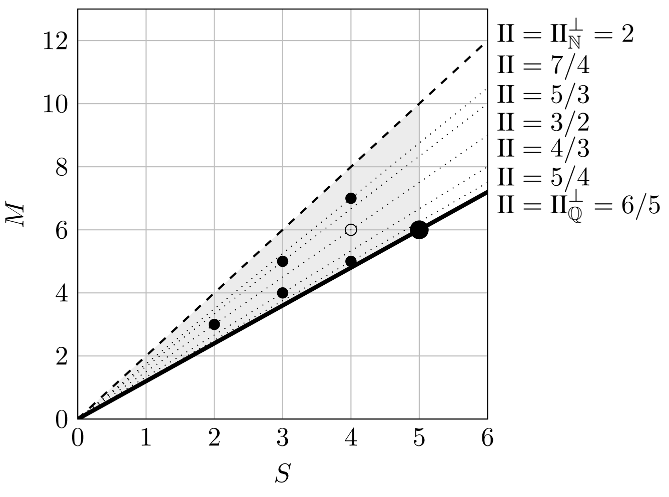


Fig. 9: Possible values for S and M when $\mathrm{II}_{\mathbb{Q}}^{\perp} = \frac{6}{5}$

In Figure 9, we show the design space of a scheduling problem where $\mathrm{II}_{\mathbb{Q}}^{\perp} = \frac{6}{5}$. The large black point at $(5, 6)$ represents the starting point, and the smaller black points represent all the other IIs that we use as fallbacks. In all cases, we pick S and M such that the resulting II lies between $\mathrm{II}_{\mathbb{Q}}^{\perp}$ and $\mathrm{II}_{\mathbb{N}}^{\perp}$. We also only pick IIs where $S \leq 5$, because we judge that if scheduling fails when $S = 5$, then setting $S > 5$ is unlikely to make it more feasible, since increasing S relates to increasing variables and constraints. Thus we restrict our attention to the IIs within the grey triangle. We try these IIs in ascending order of their value (i.e. their slope in Figure 9). Note that IIs with the same value lie on the same line through the origin, as shown by the dotted lines in Figure 9. We omit IIs whose fractions are not fully reduced, such as the hollow dot in Figure 9 which represents an II of $\frac{6}{4}$, because a solution to that scheduling problem would give the same II as if $\frac{3}{2}$ has been solved.

More generally, if the minimum rational II is $\mathrm{II}_{\mathbb{Q}}^{\perp} = M^{\perp}/S^{\perp}$ in its lowest form, then we search for fallback IIs of the form M/S that satisfy:

$$\mathrm{II}_{\mathbb{Q}}^{\perp} \leq \frac{M}{S} < \mathrm{II}_{\mathbb{N}}^{\perp} \qquad \text{and} \qquad S \leq S^{\perp} \tag{14}$$

and are also in their lowest form.

Our algorithm builds on Rau's *iterative modulo scheduling* algorithm [2], but has one crucial complication: Rau's algorithm uses integer IIs, so it can simply increment that integer for each successive scheduling attempt. In the following, we propose a straightforward method of 'incrementing' a rational number for rational-II modulo scheduling.

Algorithm 3 Generating rational-II sequences

```
Require: $S^\perp, M^\perp, S_{\max}$
Ensure: A sequence of rational IIs
1:  $\mathrm{II}_{\mathbb{N}}^\perp \leftarrow \lceil M^\perp / S^\perp \rceil$
2:  queue $\leftarrow \{\}$
3:  if $S^\perp \leq S_{\max}$ then
4:      queue.insert($S^\perp, M^\perp$)
5:  for $S \leftarrow 2$ to $S_{\max}$ do
6:      for $M \leftarrow \lceil \mathrm{II}_{\mathbb{Q}}^\perp \cdot S \rceil$ to $\mathrm{II}_{\mathbb{N}}^\perp \cdot S$ do
7:          if irreducible($S, M$) then
8:              queue.insert($S, M$)
9:  ratIISort(queue)
10: return queue
```

Algorithm 4 Iterative Rational-II scheduling

```
Require: $G$, FUs, $i_{\max}$, $S_{\max}$, $time$
Ensure: A schedule $\mathbb{S}$
1: $S^\perp, M^\perp \leftarrow$ minQII($G$, FUs)                    //Equ. 4
2: q $\leftarrow$ getQueue($S^\perp, M^\perp, S_{\max}$)               //Alg. 3
3: $\mathbb{S} \leftarrow \{\}$                          //schedule times container
4: $i \leftarrow 0$
5: while $i < i_{\max}$ and $\mathbb{S}$.empty and !q.empty do
6:     $\mathbb{S} \leftarrow$sched($G$, FUs, q.front, $time$)   //e.g. Alg. 2
7:     queue.pop_front()
8:     $i \leftarrow i + 1$
9: return $\mathbb{S}$
```

We describe how rational numbers for iterative rational-II modulo scheduling are determined in Algorithm 3. The required inputs are $S^\perp, M^\perp$ and $S_{\max}$. By default, $S_{\max}$ is set to $S^\perp$, but by making it a distinct parameter, we allow the possibility of capping S at smaller values. As we shall show in our experiments, better results can often be obtained by keeping S small. Using $S^\perp$ and $M^\perp$, $\mathrm{II}_{\mathbb{N}}^\perp$ is calculated in line 1. A container that collects all pairs of M and S that combine to candidate rational IIs is initiated in line 2. Then, whenever $S^\perp \leq S_{\max}$, our iterative algorithm considers $\mathrm{II}_{\mathbb{Q}}^\perp$ as a starting point by inserting it into the queue of rational IIs in lines 3–4.

Our proposal for enumerating rational numbers can be seen in lines 5–8. We enumerate the rational numbers that satisfy (14) using the for-loops in lines 5 and 6. Reducible fractions are skipped in line 7; the others are inserted into the queue in line 8. Finally, the queue is sorted to prepare the iteration over rational-IIs in line 9 and returned in line 10.

Our procedure that performs iterative rational-II modulo scheduling is given in Algorithm 4. The main idea is to find a valid schedule for the problem description (G, FUs) in $i_{\max}$ iteration steps using an upper bound $S_{\max}$ within a specified time frame. At first, $S^\perp, M^\perp$ are calculated in line 1. Then, the queue of rational IIs is generated in line 2.

The iteration is controlled by the while-loop in lines 5–8. The loop terminates if either the maximum number of allowed iteration steps is reached, a schedule has been found or no more candidate IIs are in the queue. An attempt to solve the scheduling problem using the first element of the candidate II queue is made in line 6. In lines 7–8, the first element is removed from the queue in order to proceed to the next candidate II and the iteration counter is incremented. A valid schedule or an empty container is returned in line 9.

Algorithms 3 and 4 can be controlled by the user through the $S_{\max}$ and $i_{\max}$ values. The impact of varying $S_{\max}$ and $i_{\max}$ on scheduling results regarding throughput achieved is examined on large problems in Section VIII-D.

VIII. Experiments

We now evaluate the proposed methods. First, we analyze how much speedup w.r.t. II can be obtained using rational-II modulo scheduling. Then, we show that our non-uniform scheduler performs the best in terms of optimal rational-II scheduling and that our SCC-based heuristic increases solving rate significantly. We do this by examining how many of the encountered problems can be solved within a fixed time limit by (1) the proposed approaches, (2) the Fimmel–Müller (FM) rational-II formulation, and (3) the unrolling approach using three different state-of-the-art integer-II schedulers: MV [21], MSDC [13] and ED97 [20]. Also, sample latency achieved is discussed. Then, the proposed iterative method for rational II modulo scheduling is evaluated and discussed. Finally, we show how rational-II scheduling can improve Pareto-frontiers regarding throughput and area after place & route.

A. Experimental Setup

We have evaluated the various scheduling approaches on a set of 16 test instances from digital signal processing and embedded computing. The vanDongen benchmark was used by Fimmel and Müller [3]; we include it because it is the only example we could find where their assumption of $\mathrm{II}_{\mathrm{rec}}^\perp > \mathrm{II}_{\mathrm{res}}^\perp$ can actually be met. We have used 13 of the remaining benchmarks before [6]. The remaining two, iir [37] and r-2 FFT [40], which are larger problems in terms of number of operations and $\mathrm{II}_{\mathrm{rec}}^\perp$, are new. The source code of all our benchmarks is available online [7], [12]. Gurobi 8.1 (in single-threaded mode) was used as the solver.

All problems were solved on a server system with an Intel Xeon E5-2650v3 2.3 GHz CPU with 128 GB RAM. The hardware description after scheduling was generated using *Origami HLS* [12] which itself uses *FloPoCo* [11] for VHDL generation. The examined hardware implementations were synthesized, placed, and routed using Vivado v2018.1 for a Xilinx Virtex7 xc7v2000t g1925-2G targeting 250 MHz.

B. Analysing Potential Speedups

First, we analyse the potential speedup for rational-II scheduling by evaluating $\mathrm{II}_{\mathrm{rec}}^\perp$ and $\mathrm{II}_{\mathrm{res}}^\perp$ for all possible resource allocations (#FUs) for each problem. The results of this experiment are displayed in Table VIII. To provide a sense of the complexity, the number of operations that have to be performed per loop iteration (#ops) and the $\mathrm{II}_{\mathrm{rec}}^\perp$ are given. Resources are shared only within loops. Every operation of the same type is implemented using homogeneous FUs. For each benchmark, we enumerate all possible resource allocations (#allocs). The 'avg. $\mathrm{II}_{\mathrm{res}}^\perp$' column reports the average value

TABLE VIII: Analysing the speedup that can be potentially obtained by using rational IIs rather than integer IIs.

	DFG properties		Allocation info (sweep over all possible resource allocations)				Potential speedup	
instance	#ops	$\mathrm{II}^{\perp}_{\mathrm{rec}}$	#allocs	avg. $\mathrm{II}^{\perp}_{\mathrm{res}}$	$\#(\mathrm{II}^{\perp}_{\mathrm{res}} > \mathrm{II}^{\perp}_{\mathrm{rec}})$	rational II	avg.	max.
vanDongen [31]	10	5.33	10	2.93	1	9 (90%)	1.13×	1.13×
dlms [32]	16	4	15	2.71	3	0 (0%)	–	–
gen [32]	15	1	15	2.71	14	7 (47%)	1.3×	1.6×
gm [33]	16	1	24	3.04	23	5 (21%)	1.47×	1.67×
hilbert [34]	14	1	18	2.42	17	3 (17%)	1.33×	1.33×
lms [32]	15	18	15	2.71	0	0 (0%)	–	–
linear phase [35]	29	1	91	4.11	90	71 (78%)	1.25×	1.87×
srg [32]	17	1	8	2.29	7	1 (13%)	1.5×	1.5×
sam [36]	121	1	1770	6.77	1769	1403 (79%)	1.21×	1.97×
biquad [37]	14	10	16	2.69	0	0 (0%)	–	–
rgb [38]	24	1	64	3.07	63	7 (11%)	1.5×	1.5×
spline [38]	26	1	64	3.78	63	26 (41%)	1.3×	1.75×
ycbcr [38]	22	1	32	2.78	31	3 (9%)	1.5×	1.5×
iir [37]	194	14	4096	7.16	496	123 (3%)	1.03×	1.03×
cholesky [39]	266	1	113386	9.31	113385	100235 (88%)	1.15×	1.98×
r-2 FFT [40]	576	1	2408448	12.07	2408447	1978795 (82%)	1.15×	1.99×
average	85.94	3.9	–	4.41	–	– (**36**%)	**1.24**×	**1.49**×

of $\mathrm{II}^{\perp}_{\mathrm{res}}$ over these allocations. We then report how many of the possible resource allocations lead to $\mathrm{II}^{\perp}_{\mathrm{res}} > \mathrm{II}^{\perp}_{\mathrm{rec}}$. In test instances biquad and lms, we find that $\mathrm{II}^{\perp}_{\mathrm{rec}}$ always dominates $\mathrm{II}^{\perp}_{\mathrm{res}}$, and since $\mathrm{II}^{\perp}_{\mathrm{rec}}$ is an integer in both cases, no speedup can be obtained using rational-II scheduling.

We then report how many of the remaining resource allocations have a minimum II that is not an integer (column 'rational II'). For example, test instance dlms has $\mathrm{II}^{\perp}_{\mathrm{res}} > \mathrm{II}^{\perp}_{\mathrm{rec}}$ in three out of its 15 possible resource allocations, but still the minimum II in each case is an integer. This can be explained by the fact that the resource type with the largest number of operations is `mult`, with five instances. No allocation can lead to a rational II between 4 and 5 and, thus, no speedup can be obtained using rational-II scheduling. Note that this can always be determined quickly before attempting scheduling (see Section IV-A) and an integer-II scheduler can be used instead. In all other cases, there exist resource allocations where the minimum II is not an integer.

On average, 36% of all resource allocations show speedup potential for rational-II scheduling (see bottom row of Table VIII). Of those, the average potential speedup is 1.24×. In the larger models (r-2 FFT, cholesky), the maximum speedup potential reaches 1.99× which is consistent with the range we derived in Section IV-A.

C. Measuring Actual Speedups

Now, we analyze the performance of uniform and non-uniform rational-II modulo scheduling formulations in terms of II and latency achieved. To reduce the number of scheduling experiments for the large sam, iir, cholesky, and r-2 FFT benchmarks, only resource allocations with a potential speedup of $\mathrm{II}^{\perp}_{\mathbb{N}}/\mathrm{II}^{\perp}_{\mathbb{Q}} \geq 1.05$ were considered.

We solved all problems using the proposed SCC-based, the proposed uniform, and the proposed non-uniform approach, and compared the performance achieved to four state-of-the-art approaches: the FM formulation [3] and, after partially unrolling the problem, three integer-II formulations: MV [21], MSDC [13] and ED97 [20]. For each experiment, a solver

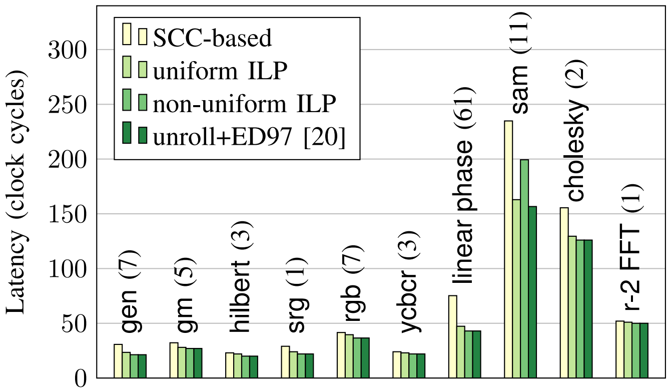


Fig. 10: Average latency achieved by four different schedulers. For each benchmark, we give in parentheses the number of resource allocations where all four schedulers could find a solution; the latency is averaged over these allocations. The absent benchmarks are those where no allocations led to all four schedulers finding a solution.

timeout of 300 seconds, no iteration ($i_{\max} = 1$) and no variation of $S_{\max}$ was used. Benchmarks dlms, lms and biquad do not appear because there were no allocations with a non-integer minimum II.

Results of all scheduling experiments are in Table IX. The first and seventh schedulers use heuristics so we cannot tell which of their solutions are optimal. The most solutions (83%) were found by the heuristic SCC-based uniform rational-II scheduling approach. Although the overall solving rate of the proposed ILP-based formulation (10%) is relatively low, almost all problems (111/120) among the smaller benchmarks (fewer than 100 vertices) were solved optimally w.r.t. latency. The only problems where the solver reported infeasible were encountered using the vanDongen instance for uniform schedulers. In fact, no uniform schedule can exist (see Section IV-D). All other missing solutions are due to timeouts. In Section VIII-D, we investigate how the proposed iterative

TABLE IX: Comparing the performance of rational-II schedulers. Each instance (first column) gives rise to several scheduling problems (second column), one per resource allocation. For each scheduler we give the number of problems solved (with a 5-minute timeout), and how many of those solutions are latency-optimal.

		prop. SCC-based		prop. uniform		prop. non-uniform		FM [3]		unroll+MV [21]		unroll+MSDC [13]		unroll+ED97 [20]	
instance	#prob.	#sol.	opt.	#sol.	opt.	#sol.	opt.	#sol.	opt.	#sol.	opt.	#sol.	opt.	#sol.	opt.
vanDongen	9	0	-	0	0	9	9	9	9	9	9	9	-	9	9
gen	7	7	-	7	7	7	7	7	7	7	7	7	-	7	7
gm	5	5	-	5	5	5	5	5	5	5	4	5	-	5	5
hilbert	3	3	-	3	3	3	3	3	3	3	3	3	-	3	3
srg	1	1	-	1	1	1	1	1	1	1	1	1	-	1	1
rgb	7	7	-	7	7	7	7	7	7	7	4	7	-	7	7
spline	26	26	-	26	24	26	26	26	7	22	20	26	-	26	26
ycbcr	3	3	-	3	3	3	3	3	3	3	3	3	-	3	3
linear phase	71	71	-	61	61	71	71	71	9	39	34	71	-	71	71
sam	500	500	-	9	4	500	106	80	0	1	0	29	-	140	136
iir	123	122	-	0	0	84	82	0	0	0	0	0	-	18	17
cholesky	197	135	-	2	2	18	9	0	0	0	0	13	-	13	12
r-2 FFT	232	108	-	1	1	5	1	0	0	0	0	5	-	6	2
avg.	-	83%	-	10%	9%	62%	27%	17%	4%	8%	7%	15%	-	26%	25%
total	1184	988	-	125	118	739	330	212	51	97	85	179	-	309	299
total in $\leq$ 1 min	1184	983	-	109	109	242	242	46	46	75	75	164	-	191	191
total time	-	80.41 min		5098.95 min		4359.35 min		3948.54 min		3578.10 min		2521.13 min		3417.21 min	
avg. time per sol.	-	0.07 min		0.53 min		3.19 min		3.89 min		0.94 min		0.26 min		1.27 min	

method can increase solving rate.

Regarding non-uniform rational-II scheduling, our proposed formulation performed the best and solved 62% of all problems. The FM formulation solved 17%, but only 51 of the 212 solutions are optimal. For all the other approaches, the optimal II was always achieved whenever a solution was found, since the II is an input not the objective. For the unrolling approaches, ED97 performs the best with 26% of problems solved. The proposed heuristic that achieved the best solving rate does not minimise latency. The last two rows of Table IX show the total time taken (including timeouts) and the average time per *solved* instance (i.e., excluding timeouts). We see that the SCC-based heuristic finds the most solutions and also does so in the shortest time.

We now analyse average latency achieved in Figure 10. Since it solved the most problems and almost all solutions were optimal w.r.t. latency, we compare our approaches to ED97. For all benchmarks, ED97 achieved the shortest and the SCC-based heuristic the longest latency on average. Latency was increased by a factor of 1.33× on average and 2.21× at most. Since it is not obvious how latency impacts the final hardware costs in the context of rational-II modulo scheduling, we plan to investigate this in future work.

D. Evaluating our Iterative Scheduler

The experiments in the previous subsection were carried out without applying iterative rational-II modulo scheduling and adapting $S_{\max}$. At best only 18 out of 197 and 6 out of 232 problems from the cholesky and r-2 FFT benchmark were solved, respectively. We investigate how the solving rate can be improved using the above-mentioned methods. By using Algorithm 3, the candidate II deviates from $\mathrm{II}_{\mathbb{Q}}^{\perp}$. Therefore, we use the quotient ($\mathrm{II}_{\mathbb{Q}}^{\perp}/\mathrm{II}_{\mathrm{a}}$) of II achieved ($\mathrm{II}_{\mathrm{a}}$) and the theoretical minimum II for comparison. A schedule that achieves $\mathrm{II}_{\mathrm{a}} = \mathrm{II}_{\mathbb{Q}}^{\perp}$ is rated with a quality of 1, decreasing when II_{a} becomes larger. If no solution in the given time limit was found, we logged an II quality of 0.

To examine our iterative approach, we now focus on the cholesky benchmark. In our experiments, the r-2 FFT benchmark showed the same characteristics. Figure 11 shows the average II quality achieved for all 197 rational-II scheduling problems using different settings for $i_{\max}$ and $S_{\max}$. The average II quality of integer-II scheduling is shown as a dashed line. For $i_{\max} = 10$, we can see that $3 \leq S_{\max} \leq 20$ outperforms the integer-II baseline in terms of II quality. The results show stability in the range of $6 \leq S_{\max} \leq 14$ and peak at 13 (II quality of 0.93). It is interesting to observe that for smaller values of $i_{\max} = 2$ and $i_{\max} = 4$, we observe that the II quality follows the trend of $i_{\max} = 10$ at first and gets worse earlier the smaller the value of $i_{\max}$.

Close to optimal II quality results that outperform integer-II scheduling for the cholesky benchmark using the proposed iterative rational-II modulo scheduling approach with the non-uniform ILP formulation were achieved using $i_{\max} = 10$ and $S_{\max} = 13$. Since a new solving process is started for each iteration, this does not necessarily mean that more solving time is required. Moreover, it indicates that the identification of small values for $S_{\max}$ before solving the ILP may speed up the solving process. Larger values for $i_{\max}$ led to better results, which is expected as a general behaviour, since the iteration process attempts to solve the scheduling problem under increasingly relaxed conditions.

E. Design-Space Exploration

To understand hardware overhead after place & route, we studied all 71 resource allocations of the linear phase

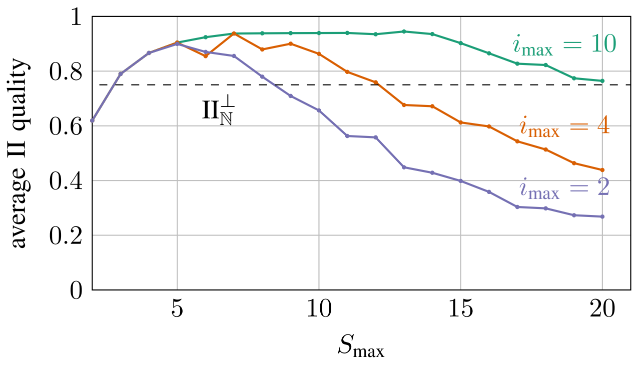


Fig. 11: Average II quality achieved for cholesky using the proposed non-uniform rational-II scheduler from Section IV and the proposed iteration from Section VII. Each mark represents the average over 197 scheduling problems.

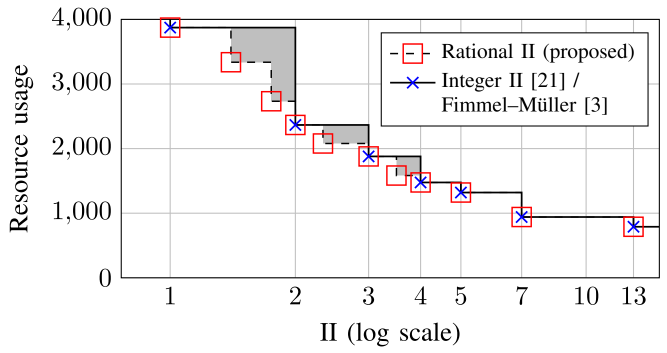


Fig. 12: Exploring Pareto-optimal implementations of the linear phase benchmark after place & route [6].

benchmark model. We focus on linear phase as it is large enough to show interesting tradeoffs, but small enough to be solved for all Pareto-optimal resource allocations; other benchmarks showed similar patterns. We define resource usage as

$$RU = slicesUsed + N \cdot DSPsUsed,$$

where N denotes the slice-to-DSP ratio of the given FPGA device; in our experiments, $N = 142$. The Pareto frontier for II and resource usage is shown in Figure 12. The Pareto-optimal implementations that were found using integer-IIs are displayed as crosses. Four new Pareto-optimal designs are revealed using our proposed rational-II formulation. These can be achieved because rational IIs fill the gaps between integers.

Note that all implementations are able to support the demanded 250 MHz, and implementing rational-II scheduling does not affect the operating frequency of the final hardware. Note also that, at least on this benchmark, rational-II scheduling does not change the fact that the best possible II is still 1 (top-left point), and the best possible resource usage is still provided by the bottom-right point; the value here is the finer-grained control over the design-space exploration. This 'fine-grained control' is highlighted by the area (highlighted in grey) between the two Pareto frontiers.

IX. Conclusion & Future Work

We show that in 35% of the encountered scheduling problems, speedups w.r.t. II of 1.24× on average and up to 1.99× are possible compared to integer-II modulo scheduling. To take advantage of this potential, we have presented novel ILP formulations for uniform and non-uniform rational-II modulo scheduling that are able to determine optimal rational IIs whenever the number of operations in the DFG does not exceed about 150. We have proposed the first heuristic for rational-II modulo scheduling that is able to solve 83% of the encountered problems with an average II quality of 0.86.

We have proposed the first framework for iterative rational-II scheduling. By doing this we achieve two things; first, a fallback strategy that iterates to easier-to-solve problems in case of solver timeout is introduced. Optimality w.r.t. II is sacrificed, but this is better than no solution at all. Second, we show that using the proposed iteration procedure with 'good' values for $i_{\max}$ and S_{max} enables us to solve rational-II benchmark problems with up to 266 vertices with an II quality of 0.93. Tuning of $i_{\max}$ and S_{max} will be investigated in future work. Finally, Pareto frontiers after place & route can be improved using our approach, thus enabling a more fine-grained control over the design-space. Complete enumeration of the design-space is not feasible; identification of resource allocations that actually contribute to the Pareto frontier will be addressed in future work. In addition, the theoretical analysis of the minimum II in combination with synthesis results from Section VIII-E indicate that it is possible to identify resource allocations that lead to the Pareto frontier before scheduling and synthesis. We envision reducing the overall design time for multi-objective optimisation in custom hardware design by our approach significantly.

Acknowledgements

This work was carried out while the first author visited Imperial College, supported by the UK EPSRC (grant EP/P010040/1). We also acknowledge the financial support of grant ZI 762/5-1 from the German Research Foundation (DFG) and grant EP/R006865/1 from the EPSRC.

References

[1] B. R. Rau and C. D. Glaeser, "Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High-performance Scientific Computing," *ACM SIGMICRO*, pp. 183–198, 1981.

[2] B. R. Rau, "Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops," in *Proc. of the 27th Int. Symposium on Microarchitecture*, 1994.

[3] D. Fimmel and J. Müller, "Optimal Software Pipelining under Resource Constraints," *Int. Journal of Foundations of Computer Science*, 2001.

[4] S. Dai and Z. Zhang, "Improving Scalability of Exact Modulo Scheduling with Specialized Conflict-Driven Learning," in *Proc. of the 56th Annual Design Automation Conference*, 2019.

[5] R. Tarjan, "Depth-First Search and Linear Graph Algorithms," in *SIAM Journal on Computing*, 1972, pp. 146–160.

[6] P. Sittel, J. Wickerson, M. Kumm, and P. Zipf, "Modulo Scheduling with Rational Initiation Intervals in Custom Hardware Design," in *25th Asia and South Pacific Design Automation Conference*, 2020.

[7] P. Sittel, J. Oppermann, M. Kumm, A. Koch, and P. Zipf, "HatScheT: A Contribution to Agile HLS," in *Int. Workshop on FPGAs for Software Programmers*, 2018.

[8] P. Sittel, T. Schönwälder, M. Kumm, and P. Zipf, "ScaLP: A Light-Weighted (MI)LP Library," in *Workshop on 'Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen'*, 2018.

[9] M. Berkelaar, K. Eikland, and P. Notebaert, "LPSolve: Open Source (Mixed-Integer) Linear Programming System," *Eindhoven U. of Technology*, 2004.

[10] T. Achterberg, "SCIP: Solving Constraint Integer Programs," *Springer Mathematical Programming Computation*, pp. 1–41, 2009.

[11] F. de Dinechin and B. Pasca, "Designing Custom Arithmetic Data Paths with FloPoCo," *IEEE Design & Test of Computers*, 2011.

[12] Origami HLS. http://www.uni-kassel.de/go/origami.

[13] A. Canis, S. D. Brown, and J. H. Anderson, "Modulo SDC Scheduling with Recurrence Minimization in High-level Synthesis," in *24th Int. Conf.on Field Programmable Logic and Applications*, 2014.

[14] 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. on Embedded Computing Systems*, vol. 13, no. 2, pp. 1–27, 2013.

[15] M. Lam, "Software Pipelining: An Effective Scheduling Technique for VLIW Machines," in *ACM SIGPLAN Conf.on Programming Language design and Implementation*, 1988.

[16] J. Oppermann, P. Sittel, M. Kumm, M. Reuter-Oppermann, A. Koch, and O. Sinnen, "Design-Space Exploration with Multi-Objective Resource-Aware Modulo Scheduling," in *25th European Conf. on Parallel Processing*, 2019.

[17] J. Oppermann, M. Reuter-Oppermann, L. Sommer, A. Koch, and O. Sinnen, "Exact and Practical Modulo Scheduling for High-level synthesis," *ACM Trans. on Reconfigurable Technology and Systems*, vol. 12, no. 2, pp. 1–26, 2019.

[18] Z. Zhang and B. Liu, "SDC-based Modulo Scheduling for Pipeline Synthesis," in *IEEE/ACM Int. Conf. on Computer-aided Design*, 2013.

[19] J. Oppermann, "Advances in ILP-based Modulo Scheduling for High-Level Synthesis," Ph.D. dissertation, TU Darmstadt, 2019.

[20] A. E. Eichenberger and E. S. Davidson, "Efficient Formulation for Optimal Modulo Schedulers," *Proc. of the ACM SIGPLAN'97 Conf. on Programming Language Design and Implementation*, 1997.

[21] J. Oppermann, A. Koch, M. Reuter-Oppermann, and O. Sinnen, "ILP-based Modulo Scheduling for High-level Synthesis," in *IEEE Int. Conf. on Compilers, Architectures, and Synthesis of Embedded Systems*, 2016, pp. 1–10.

[22] K. Fan, M. Kudlur, H. Park, and S. Mahlke, "Cost Sensitive Modulo Scheduling in a Loop Accelerator Synthesis System," in *Proc. of the 38th Annual IEEE/ACM Inter. Symposium on Microarchitecture*, 2005.

[23] R. A. Huff, "Lifetime-sensitive modulo scheduling," *ACM SIGPLAN Notices*, vol. 28, no. 6, pp. 258–267, 1993.

[24] J. Llosa, A. González, E. Ayguadé, and M. Valero, "Swing Modulo Scheduling: A Lifetime-sensitive Approach," in *Proc. of the IEEE Conf.on Parallel Architectures and Compilation Technique*, 1996.

[25] L. de Souza Rosa, C.-S. Bouganis, and V. Bonato, "Scaling Up Modulo Scheduling For High-Level Synthesis," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 5, pp. 912–925, 2018.

[26] L. de Souza Rosa, V. Bonato, and C.-S. Bouganis, "Scaling Up Loop Pipelining for High-Level Synthesis: A Non-iterative Approach," in *Proc. of the IEEE Int. Conf.on Field-Programmable Technology*, 2018, pp. 62–69.

[27] P. Sittel, N. Fiege, M. Kumm, and P. Zipf, "Isomorphic Subgraph-based Problem Reduction for Resource Minimal Modulo Scheduling," in *Int. Conf.on Reconfigurable Computing and FPGAs*, 2019.

[28] J. Cong and Z. Zhang, "An Efficient and Versatile Scheduling Algorithm Based on SDC Formulation," in *43rd ACM/IEEE Design Automation Conference*, 2006.

[29] J. H. Patel and E. S. Davidson, "Improving the Throughput of a Pipeline by Insertion of Delays," in *ACM Computer Architecture News*, 1976.

[30] J. E. Bresenham, "Algorithm for Computer Control of a Digital Plotter," *IBM Systems Journal*, vol. 4, no. 1, pp. 25–30, 1965.

[31] V. H. Van Dongen, G. R. Gao, and Q. Ning, "A Polynomial Time Method for Optimal Software Pipelining," in *Parallel Processing: CONPAR 92—VAPP V*. Springer, 1992.

[32] U. Meyer-Baese, A. Vera, A. Meyer-Baese, M. Pattichis, and R. Perry, "Discrete Wavelet Transform FPGA Design using MatLab/Simulink," in *Independent Component Analyses, Wavelets, Unsupervised Smart Sensors, and Neural Networks IV*, 2006.

[33] D. Goodman and M. Carey, "Nine Digital Filters for Decimation and Interpolation," *IEEE Trans. on Acoustics, Speech, and Signal Processing*, vol. 25, no. 2, pp. 121–126, 1977.

[34] M. Kumm and M. S. Sanjari, "Digital Hilbert Transformers for FPGA-based Phase-locked Loops," in *IEEE Int. Conf.on Field Programmable Logic and Applications*, 2008.

[35] D. Shi and Y. J. Yu, "Design of Linear Phase FIR Filters with High Probability of Achieving Minimum Number of Adders," *IEEE Trans. on Circuits and Systems*, vol. 58, no. 1, pp. 126–136, 2011.

[36] H. Samueli, "An Improved Search Algorithm for the Design of Multiplierless FIR Filters with Powers-of-two Coefficients," *IEEE Trans. on Circuits and Systems*, vol. 36, no. 7, pp. 1044–1047, 1989.

[37] K. K. Parhi, *VLSI Digital Signal Processing Systems: Design and Implementation*. John Wiley & Sons, 2007.

[38] D. G. Bailey, *Design for Embedded Image Processing on FPGAs*. John Wiley & Sons, 2011.

[39] (2011) Cholesky Decomposition. http://www.alterawiki.com/wiki/Floating-point_Matrix_Inversion_Example.

[40] M. Garrido, J. Grajal, M. Sánchez, and O. Gustafsson, "Pipelined Radix-2^k Feedforward FFT Architectures," *IEEE Trans. on Very Large Scale Integration Systems*, vol. 21, no. 1, pp. 23–32, 2013.

Patrick Sittel received a M.Sc. degree in Electrical and Communications Engineering from the University of Kassel in 2016. He is a Ph.D. candidate at the Department of Digital Technology at the University of Kassel. His research interests include high-level synthesis, computer-aided design of embedded systems and FPGAs.

Nicolai Fiege received a B.Sc. degree in Electrical Engineering from the University of Kassel in 2018, where he is currently working on his M.Sc. in Electrical Engineering. His research interests include computer-aided hardware design and hardware implementation of artificial neural networks.

John Wickerson (M'17, SM'19) received a Ph.D. in Computer Science from the University of Cambridge in 2013. He is a Lecturer in the Department of Electrical and Electronic Engineering at Imperial College London. His research interests include high-level synthesis, the design and implementation of programming languages, and software verification. He is a Senior Member of the IEEE and a Member of the ACM.

Peter Zipf (M'05) received the Ph.D. (Dr.-Ing.) degree from the University of Siegen, Germany, in 2002. He was a Postdoctoral Researcher at the Department of Electrical Engineering and Information Technology, Darmstadt University of Technology, Darmstadt, Germany, until 2009. He is currently the chair of Digital Technology at the University of Kassel, Germany. His current research interests include reconfigurable computing, embedded systems and CAD algorithms for circuit optimization.