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 $1.24 \times$ 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 maximizing 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 naive 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 VII) 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.
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
}

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 1(a) 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 1(b) 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]:

\[
\Pi^r_{\text{rec}} = \max_{j \in \text{recurrences}} \left( \frac{\text{latency}_j}{\text{distance}_j} \right),
\]

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 \( \frac{5}{\text{FUs}(r)} \), where \( \text{FUs}(r) \) is the number of functional units that can execute operations of type \( r \). This is called the resource-constrained minimum II:

\[
\Pi^r_{\text{res}} = \max_{r \in \text{resources}} \left( \frac{\#r}{\text{FUs}(r)} \right)
\]

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 1(b) compared to optimal integer-II solutions, over all possible resource allocations. In all cases except \( \text{FUs}(r) = 1 \), our approach leads to improved throughput, reaching 33% when \( \text{FUs}(r) = 4 \) or 5.

To be more concrete, Table I shows one possible outcome of scheduling our example graph when \( \text{FUs}(r) = 3 \), 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.
Traditionally, the problem is simplified by using the integer-II modulo scheduling, since it is implemented in the widely used HLS tool LegUp [14] and is competitive with 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 $\Pi_{\text{res}}^{\perp} < \Pi_{\text{rec}}^{\perp}$. 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 VII and the first framework for iterating over rational-IIs in modulo scheduling in Section VIII. 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.

### 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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>FU1</td>
<td>0</td>
<td>0</td>
<td>FU1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>FU2</td>
<td>0</td>
<td>0</td>
<td>FU2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>FU3</td>
<td>0</td>
<td>0</td>
<td>FU3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ \Pi_{\text{res}}^{\perp} = \max(\Pi_{\text{res}}^{\perp}, \Pi_{\text{rec}}^{\perp}) \]  
\[ \Pi_{\text{rec}}^{\perp} = \max(\Pi_{\text{res}}^{\perp}, \Pi_{\text{rec}}^{\perp}) \]
TABLE III: A glossary of constants (top) and variables (bottom) for resource-constrained rational-II modulo scheduling

<table>
<thead>
<tr>
<th>Constant/Variable</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$o_i \in O$</td>
<td>Set of operations in the DFG</td>
</tr>
<tr>
<td>$(o_i, o_j) \in E$</td>
<td>Set of edges in the DFG</td>
</tr>
<tr>
<td>$d_{ij} \in \mathbb{N}_0$</td>
<td>Dependence on edge $o_i \rightarrow o_j$</td>
</tr>
<tr>
<td>$R$</td>
<td>Set of resource-constrained operation types</td>
</tr>
<tr>
<td>$O \subseteq R$</td>
<td>Set of resource-constrained operations</td>
</tr>
<tr>
<td>$O_r \subseteq \hat{O}$</td>
<td>Set of resource-constrained operations of type $r \in R$</td>
</tr>
<tr>
<td>$\text{FU}(r) \in \mathbb{N}$</td>
<td>No. of hardware instances of resource type $r \in R$</td>
</tr>
<tr>
<td>$D_i \in \mathbb{N}_0$</td>
<td>Latency of operation $o_i \in O$</td>
</tr>
<tr>
<td>$M \in \mathbb{N}$</td>
<td>No. of cycles before the modulo schedule repeats</td>
</tr>
<tr>
<td>$S \in \mathbb{N}$</td>
<td>No. of samples inserted every $M$ cycles</td>
</tr>
<tr>
<td>$S^\perp, M^\perp$</td>
<td>Values for $S, M$ that lead to $\Pi_2^*$</td>
</tr>
<tr>
<td>$0 \leq s \leq S - 1$</td>
<td>Range of sample indices</td>
</tr>
<tr>
<td>$\Pi_2 = \frac{M}{\sqrt{S}}$</td>
<td>Rational initiation interval</td>
</tr>
<tr>
<td>$L \in \mathbb{N}_0$</td>
<td>Maximal latency constraint</td>
</tr>
<tr>
<td>$i_{\text{max}}$</td>
<td>Number of iterations steps allowed</td>
</tr>
</tbody>
</table>

$t_{i,s} \in \mathbb{N}_0$ | Start time of $o_i$ on sample $s$ |
$t_v$ | Virtual node |
$b_{i,s,\tau}$ | True iff $\tau$ is the start time of $o_i \in O$ on sample $s$ |
$\langle \Pi_0 \ldots \Pi_{S-1} \rangle$ | Latency sequence |
$I_s \in \mathbb{N}_0$ | Insertion time of sample $s$ |

It follows that $\Pi_2^* = [\Pi_2^*, \Pi_2^*]$, and hence that rational-II schedules will always attain a throughput that is at least as good as integer-II schedules. When $\Pi_2^*$ is already an integer, we have $\Pi_2^* = \Pi_2^*$, 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 $\Pi_2^* = 1 + \epsilon$ for small, positive $\epsilon$. In this case, the speedup is $\left[\frac{1+\epsilon}{1+\epsilon}\right]$, which tends towards 2. Overall, we have:

$$1 \leq \text{speedup} < 2.$$ (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 $\hat{O}$ for the set of operations that require resource type $r$ (adder, etc.). The number of available functional units of type $r$ is $\text{FU}(r)$.

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 $\frac{M}{\sqrt{S}}$, 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.

![Fig. 3: An ILP for (non-uniform) rational-II scheduling](image)

where the number of variables increases quadratically with the number of time variables and resource constraints [18], [21]. In Section VIII we address this problem by picking small $S$ values, aiming to reduce the complexity of the ILP problem while maintaining high throughput.

C. ILP Formulation

The general problem of rational-II modulo scheduling is formulated in Figure 3. Constraint $D_1$ is a variation on the standard causality constraint from integer-II modulo scheduling [20], [21]:

$$t_i + D_i - d_{i,j} \cdot \Pi \leq t_j \quad \forall i, j : (o_i \rightarrow o_j) \in E \quad (6)$$

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 II(a), and the edge from $o_3$ to $o_0$ in Figure 15. 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 $D_1$ 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:

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

which becomes 0 whenever causality can now be modelled within the unrolled DFG $(d-s \leq 0)$. The origin of edges $(o_i, o_j)$ is calculated according to the equation attached to $D_1$. A distance of 0 leads to $\hat{s} = s$, 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.
D. The need for non-uniform scheduling

We now give an example that takes advantage of the non-uniform nature of the resource utilization: Let's consider the loop in Figure 4. Operations $o_0$, $o_1$, and $o_2$ require the same resource $r$ that has a latency of one cycle. Let $FU_3 = 2$. From Figure 4, it follows that $II^{1}_{0} = \max(\frac{2}{5}, \frac{2}{5}) = \frac{3}{5}$. Figure 4c shows the source after unrolling the loop by a factor of $S = 2$. In Figure 4c, all vertices from Figure 4b are inserted twice ($S = 2$). Edges are inserted using $D_1$ in combination with $(7)$. The unweighted edge $(o_0, o_1)$ generates two unweighted edges $(o_0, o_1)$ and $(o_1, o_1)$. 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 $II^{1}_{0} = 2$ is shown in Table IV together with the optimal solution for $II^{1}_{1} = \frac{3}{2}$. In the integer-II case, $FU_2$ is only busy 50% of the time, while in the rational-II case all $FU$s 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 $II = \frac{3}{2}$, 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

$$ \langle II_{0}, II_{1}, \ldots, II_{S-1} \rangle $$

where

$$ II_{s} = \begin{cases} I_{s+1} - I_i & \text{if } s < S - 1 \\ M - I_i & \text{if } s = S - 1 \end{cases} $$

For instance, the sample insertion times from the example in Section I lead to a latency sequence of (2 2 1). 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 (3).

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 \( II \cdot d_{i,j} \) since data insertion is spaced equally. Assuming a latency sequence \( (II_0 \ II_1 \ldots \ II_{s-1}) \), the number of cycles between sample \( s \) and \( s - d \) can be calculated as

\[
\Delta_s(d) = \sum_{n=1}^{d} II_{(s-n) \text{ mod } s} \cdot d_{i,j}.
\] (10)

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

\[
t_{i,s} + D_i - \Delta_s(d_{i,j}) \leq t_{j,s} \ \forall s, \forall i, j : (a_i, o_j) \in E.
\] (11)

As an example, consider the rational-II schedule from Table I(b), and the edge from \( a_3 \) 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 \( \Pi \) 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 \( \Pi \) 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

\[
\Delta_{\text{min}}(d) = \min_{s \in S}(\Delta_s(d))
\] (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 \( \Pi_Q^c = \frac{18}{5} \) and \( \Pi_R^c = 4 \). 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 \( \Pi \) 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 (\( \Pi_C \)) and the floor (\( \Pi_F \)) of \( \frac{M}{S} \) (line 1). Using \( \Pi_C \), \( \Pi_F \), \( S \), and \( M \), we calculate (line 2) how many times \( \Pi_C \) and \( \Pi_F \) occur (\#\( \Pi_C \) and \#\( \Pi_F \)). In our example, we have \( S = 5, M = 18, \Pi_C = 4, \) and \( \Pi_F = 3 \). This means that the latency sequence comprises three occurrences of \( \Pi_C \) (\#\( \Pi_C = 3 \)) and two occurrences of \( \Pi_F \) (\#\( \Pi_F = 2 \)).

We then order the determined values such that \( \Pi_C \) (\#\( \Pi_C = 3 \)) has to be inserted. We also need the frequencies of \( \Pi_C \) and \( \Pi_F \), which we store in the values \( k_{\text{min}} \) and \( k_{\text{max}} \) (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, \( \Pi = 3 \)) has to be inserted.

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

Algorithm 1 Generating regular latency sequences

\[
\text{Require: } S, M
\]

\[
\text{Ensure: One } \Pi_s \text{ that maximizes } \Delta_{\text{min}}(d) \forall d \in \mathbb{N}_{\geq 0}
\]

1: \( \Pi_C \leftarrow \lceil \frac{M}{S} \rceil; \Pi_F \leftarrow \lfloor \frac{M}{S} \rfloor 
\]
2: \( \#\Pi_C \leftarrow \Pi_C \cdot S - M; \#\Pi_C \leftarrow M - \Pi_F \cdot S 
\]
3: \( s_1, s_2 \leftarrow \Pi_F, \Pi_C 
\]
4: \( k_{\text{min}}, k_{\text{max}} \leftarrow \#\Pi_C, \#\Pi_F 
\]
5: if \( \#\Pi_F < \#\Pi_C \) then
6: \( s_1, s_2 \leftarrow \Pi_C, \Pi_F 
\]
7: \( k_{\text{min}}, k_{\text{max}} \leftarrow \#\Pi_F, \#\Pi_C 
\]
8: \( E \leftarrow 0, \Pi_s \leftarrow \{ \}
\]
9: for \( i \leftarrow 1 \) to \( k_{\text{max}} \) do
10: \( \Pi_s.\text{append}(s_1) 
\]
11: \( E \leftarrow E + k_{\text{min}} 
\]
12: if \( E \geq k_{\text{min}} \) then
13: \( \Pi_s.\text{append}(s_2) 
\]
14: \( E \leftarrow E - k_{\text{max}} 
\]
15: return \( \Pi_s 
\]

D. ILP Formulation

The problem of uniform rational-II modulo scheduling is formulated in Figure 6. D1 enforces the causality constraint
introduced in [10]. Analogous to Section [V-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) [3]. 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 $FUs(r) = 5$.

This leads to $\Pi_{II}^t = \frac{3}{2}$ due to the recurrence among $o_1$, $o_2$, and $o_3$. Algorithm [1] yields the latency sequence $\langle 2, 1 \rangle$.

In Figure [7a] 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$^0_0$ and SCC$^4_4$) and two complex SCCs (SCC$^{123}_C$ and SCC$^{56}_C$).

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 [V], 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 $t_i$ 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>
<thead>
<tr>
<th>FU1</th>
<th>FU2</th>
<th>FU3</th>
<th>FU4</th>
<th>FU5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$o_{1,0}$</td>
<td>$o_{2,0}$</td>
<td>$o_{3,0}$</td>
<td>$o_{2,1}$</td>
<td>$o_{4,1}$</td>
</tr>
<tr>
<td>$o_{5,0}$</td>
<td>$o_{0,0}$</td>
<td>$o_{0,0}$</td>
<td>$o_{0,0}$</td>
<td>$o_{0,0}$</td>
</tr>
<tr>
<td>$o_{6,1}$</td>
<td>$o_{4,1}$</td>
<td>$o_{4,1}$</td>
<td>$o_{4,1}$</td>
<td>$o_{4,1}$</td>
</tr>
</tbody>
</table>

Table V: Final MRTs ($M = 3, S = 2, FUs(r) = 5$)

(a) heuristic

(b) optimal

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>
<thead>
<tr>
<th>FU1</th>
<th>FU2</th>
<th>FU3</th>
<th>FU4</th>
<th>FU5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$o_{0,0}$</td>
<td>$o_{1,0}$</td>
<td>$o_{2,0}$</td>
<td>$o_{3,0}$</td>
<td>$o_{4,0}$</td>
</tr>
<tr>
<td>$o_{5,0}$</td>
<td>$o_{6,0}$</td>
<td>$o_{7,0}$</td>
<td>$o_{8,0}$</td>
<td>$o_{9,0}$</td>
</tr>
</tbody>
</table>

Fig. 7: Example graph and its partitioning into SCCs
TABLE VI: Filling intermediate MRT slots (M = 3)

(a) 0 | 1 | 2 (b) 0 | 1 | 2 (c) 0 | 1 | 2
o1 | o2 | o3 | o1 | o2 | o3 | o1 | o2 | o3
o5 | o6 | X  | o5 | o6 | X  | o5 | o6 | o4
-   | X  | X  | -   | X  | X  | -   | X  | X 

offsetting relative start times by multiples of M.

For our example, a valid solution to step 1 is: \( t_1 = 0 \), \( t_2 = 1 \), \( t_3 = 2 \), \( t_4 = 0 \) and \( t_5 = 0 \). These times lead to
the intermediate MRT in Table VIIa, in which free (‘-’) and unavailable (‘X’) slots are highlighted.

Topologically sorting the SCC graph (step 2) leads to the order \( \langle \text{SCC}_0, \text{SCC}_4, \text{SCC}_{56}, \text{SCC}_{123} \rangle \). SCC_0 is trivial and is
scheduled ASAP. Vertex o0 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 o0 is in Table VIIb. SCC_4 is also trivial. Vertex o4 also has no incoming edges but modulo
slots 0 and 1 are already full. It follows that o4 is scheduled at time \( t_4 = 2 \). The intermediate MRT after committing all
operations of one sample is in Table VIIc.

SCC_{56} is non-trivial and operation o5 must start after o0. This means that SCC_{56} cannot be scheduled with an offset
of 0 clock cycles. For this example, the minimum offset is 3


clock cycle
FU1 FU2 FU3 FU4 FU5
0 0.00 0.00 0.00 0.00 0.00
1 0.00 0.00 1.00 0.00 1.00
2 2.00 2.00 0.00 1.00 0.00
3 2.00 2.00 0.00 1.00 0.00
4 2.00 2.00 0.00 1.00 0.00
5 4.00 4.00 3.00 5.00 5.00
6 4.00 4.00 3.00 5.00 5.00
7 4.00 4.00 3.00 5.00 5.00
8 4.00 4.00 3.00 5.00 5.00
9 5.00 5.00 5.00
10 5.00 5.00
11 5.00 5.00
12 5.00 5.00
13 5.00 5.00

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 VIII respectively.

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 VIII respectively.

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
obtain a relative schedule that is determined using the ILP formulation given in Figure 8.

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 \( 0 \leq \tau < M \), we define an \( h \) function:

\[
h(n, M, \tau) = \begin{cases} \left\lfloor \frac{n}{M} \right\rfloor, & \tau = 0 \\ h(n - \left\lfloor \frac{n}{M} \right\rfloor, M - 1, \tau - 1), & \tau > 0. \end{cases}
\]

For our example \( n = |\tilde{O}_1| = 7, M = 3, 0 \leq \tau < 3 \), we get \( h(7, 3, 0) = 3, h(7, 3, 1) = 2 \) and \( h(7, 3, 2) = 2 \) which
is the number of operations that are allowed in the respective modulo slots of the intermediate MRT shown in Table VI.

Fig. 8: An ILP problem for scheduling SCCs
Algorithm 2 Rational II scheduling heuristic

Require: \( G, \) FUs, \( S, M, \) time
Ensure: A schedule
1: \( \Pi_s \leftarrow \text{sequence}(S, M) \) //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: \hspace{1em} if SCC is trivial then
8: \hspace{2em} schedule ASAP respecting intermediate MRT
9: \hspace{2em} insert vertex of SCC into intermediate MRT
10: \hspace{1em} else
11: \hspace{2em} scheduled \( \leftarrow \) false
12: \hspace{2em} \( T \leftarrow 0 \)
13: \hspace{2em} while scheduled = false do
14: \hspace{3em} scheduled \( \leftarrow \) true
15: \hspace{3em} if time \( \leq 0 \) then
16: \hspace{4em} return \{ \}
17: \hspace{3em} for each vertex \( v_i \) in vertices of SCC do
18: \hspace{4em} try scheduling \( v_i \) in time slot \( t_i + T \)
19: \hspace{4em} if dependency constraint violated then
20: \hspace{5em} scheduled \( \leftarrow \) false
21: \hspace{5em} \( T \leftarrow T + M \)
22: \hspace{4em} break
23: return final schedule using \( \Pi_s \)

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 fulfills 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-II S

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 II, 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 \( \Pi_{Q}^\bot = \frac{M}{S} = \frac{6}{5} \).

In Figure 9, we show the design space of a scheduling problem where \( \Pi_{Q}^\bot = \frac{6}{5} \). 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 \( \Pi_{Q}^\bot \) and \( \Pi_{N}^\bot \). 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 \( 5/6 \), the origin, as shown by the dotted lines in Figure 9.

In Figure 9, we show the design space of a scheduling problem where \( \Pi_{Q}^\bot = \frac{6}{5} \). 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 \( \Pi_{Q}^\bot \) and \( \Pi_{N}^\bot \). 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 \( 5/6 \), the origin, as shown by the dotted lines in Figure 9.

Fig. 9: Possible values for \( S \) and \( M \) when \( \Pi_{Q}^\bot = \frac{6}{5} \)

More generally, if the minimum rational II is \( \Pi_{Q}^\bot = M/S^\bot \) in its lowest form, then we search for fallback IIs of the form \( M/S \) that satisfy:

\[
\Pi_{Q}^\bot \leq \frac{M}{S} < \Pi_{N}^\bot \quad \text{and} \quad S \leq S^\bot \tag{14}
\]

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: $S^\perp, M^\perp, S_{\text{max}}$
Ensure: A sequence of rational IIs
1: $\Pi_{S}^\perp \leftarrow [M^\perp/S^\perp]$
2: queue $\leftarrow \{\}$
3: if $S^\perp \leq S_{\text{max}}$ then
4: queue.insert($S^\perp, M^\perp$)
5: for $S \leftarrow 2$ to $S_{\text{max}}$ do
6: for $M \leftarrow \lceil S^\perp \cdot S \rceil$ to $S^\perp$ do
7: if irreducible$(S, M)$ then
8: queue.insert($S, M$)
9: return queue

Algorithm 4 Iterative Rational-II scheduling

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

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 $S_{\text{max}}$. By default, $S_{\text{max}}$ 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 $M^\perp$, $\Pi_{\text{II}}^\perp$ 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 $S^\perp \leq S_{\text{max}}$, our iterative algorithm considers $\Pi_{\text{II}}^\perp$ 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 $(G, \text{FU’s})$ in $i_{\text{max}}$ iteration steps using an upper bound $S_{\text{max}}$ 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 $S_{\text{max}}$ and $i_{\text{max}}$ values. The impact of varying $S_{\text{max}}$ and $i_{\text{max}}$ 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 II rec > II res 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 II rec, 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 II rec and II res 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 II rec 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. II res’ column reports the average value
TABLE VIII: Analysing the speedup that can be potentially obtained by using rational IIs rather than integer IIs.

<table>
<thead>
<tr>
<th>instance</th>
<th>DFG properties</th>
<th>Allocation info (sweep over all possible resource allocations)</th>
<th>Potential speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#ops</td>
<td>$\Pi_{\text{rec}}^\perp$</td>
<td>#allocs</td>
</tr>
<tr>
<td>vanDongen</td>
<td>10</td>
<td>5.33</td>
<td>10</td>
</tr>
<tr>
<td>dlms</td>
<td>16</td>
<td>4</td>
<td>15</td>
</tr>
<tr>
<td>gen</td>
<td>15</td>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>gm</td>
<td>16</td>
<td>1</td>
<td>24</td>
</tr>
<tr>
<td>hilbert</td>
<td>14</td>
<td>1</td>
<td>18</td>
</tr>
<tr>
<td>lms</td>
<td>15</td>
<td>18</td>
<td>15</td>
</tr>
<tr>
<td>linear phase</td>
<td>16</td>
<td>1</td>
<td>91</td>
</tr>
<tr>
<td>srg</td>
<td>17</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>sam</td>
<td>121</td>
<td>1</td>
<td>1770</td>
</tr>
<tr>
<td>biquad</td>
<td>14</td>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>rgb</td>
<td>17</td>
<td>1</td>
<td>64</td>
</tr>
<tr>
<td>spline</td>
<td>11</td>
<td>1</td>
<td>64</td>
</tr>
<tr>
<td>ycbcr</td>
<td>22</td>
<td>1</td>
<td>32</td>
</tr>
<tr>
<td>iir</td>
<td>194</td>
<td>14</td>
<td>4096</td>
</tr>
<tr>
<td>cholesky</td>
<td>266</td>
<td>1</td>
<td>113386</td>
</tr>
<tr>
<td>r-2 FFT</td>
<td>576</td>
<td>1</td>
<td>2408448</td>
</tr>
<tr>
<td>average</td>
<td>85.94</td>
<td>3.9</td>
<td>–</td>
</tr>
</tbody>
</table>

of $\Pi_{\text{res}}^\perp$ over these allocations. We then report how many of the possible resource allocations lead to $\Pi_{\text{res}}^\perp > \Pi_{\text{rec}}^\perp$. In test instances biquad and lms, we find that $\Pi_{\text{rec}}^\perp$ always dominates $\Pi_{\text{res}}^\perp$, and since $\Pi_{\text{rec}}^\perp$ 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 $\Pi_{\text{res}}^\perp > \Pi_{\text{rec}}^\perp$ 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 $\times$. In the larger models (r-2 FFT, cholesky), the maximum speedup potential reaches 1.99 $\times$ 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 $\Pi_{\text{rec}}^\perp / \Pi_{\text{res}}^\perp > 1.05$ 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 timeout of 300 seconds, no iteration ($i_{\text{max}} = 1$) and no variation of $S_{\text{max}}$ 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.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>vanDongen</td>
<td>9</td>
<td>0</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>gen</td>
<td>7</td>
<td>5</td>
<td>-</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>gm</td>
<td>5</td>
<td>5</td>
<td>-</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>hilbert</td>
<td>3</td>
<td>3</td>
<td>-</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>srg</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>rgb</td>
<td>7</td>
<td>7</td>
<td>-</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>spline</td>
<td>26</td>
<td>26</td>
<td>-</td>
<td>26</td>
<td>26</td>
<td>26</td>
<td>26</td>
<td>26</td>
<td>26</td>
</tr>
<tr>
<td>ycbcr</td>
<td>3</td>
<td>3</td>
<td>-</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>linear phase</td>
<td>71</td>
<td>71</td>
<td>-</td>
<td>61</td>
<td>61</td>
<td>61</td>
<td>61</td>
<td>61</td>
<td>61</td>
</tr>
<tr>
<td>sam</td>
<td>500</td>
<td>500</td>
<td>-</td>
<td>9</td>
<td>4</td>
<td>500</td>
<td>106</td>
<td>80</td>
<td>1</td>
</tr>
<tr>
<td>iir</td>
<td>123</td>
<td>122</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>84</td>
<td>82</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>cholesky</td>
<td>197</td>
<td>135</td>
<td>-</td>
<td>2</td>
<td>2</td>
<td>18</td>
<td>9</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>r-2 FFT</td>
<td>232</td>
<td>108</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>avg.</td>
<td>83%</td>
<td>10%</td>
<td>9%</td>
<td>62%</td>
<td>27%</td>
<td>17%</td>
<td>4%</td>
<td>8%</td>
<td>7%</td>
</tr>
<tr>
<td>total</td>
<td>1184</td>
<td>988</td>
<td>109</td>
<td>118</td>
<td>739</td>
<td>330</td>
<td>212</td>
<td>97</td>
<td>85</td>
</tr>
<tr>
<td>total in ≤ 1 min</td>
<td>1184</td>
<td>983</td>
<td>109</td>
<td>109</td>
<td>242</td>
<td>242</td>
<td>46</td>
<td>75</td>
<td>75</td>
</tr>
<tr>
<td>avg. time per sol.</td>
<td>0.71 min</td>
<td>0.53 min</td>
<td>3.19 min</td>
<td>3.89 min</td>
<td>0.94 min</td>
<td>0.26 min</td>
<td>1.27 min</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| avg. time per sol. | 0.71 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 \( S_{\text{max}} \). 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 \( II_q \). Therefore, we use the quotient (II_q/II_a) of II achieved (II_a) and the theoretical minimum II for comparison. A schedule that achieves \( II_a = II_q \) 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 \( i_{\text{max}} \) and \( S_{\text{max}} \). The average II quality of integer-II scheduling is shown as a dashed line. For \( i_{\text{max}} = 10 \), we can see that 3 ≤ \( S_{\text{max}} \) ≤ 20 outperforms the integer-II baseline in terms of II quality. The results show stability in the range of 6 ≤ \( S_{\text{max}} \) ≤ 14 and peak at 13 (II quality of 0.93). It is interesting to observe that for smaller values of \( i_{\text{max}} = 2 \) and \( i_{\text{max}} = 4 \), we observe that the II quality follows the trend of \( i_{\text{max}} = 10 \) at first and gets worse earlier the smaller the value of \( i_{\text{max}} \).

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 \( i_{\text{max}} = 10 \) and \( S_{\text{max}} = 13 \). 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 \( S_{\text{max}} \) before solving the ILP may speed up the solving process. Larger values for \( i_{\text{max}} \) 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...
grained control over the design-space. Complete enumeration
be improved using our approach, thus enabling a more fine-
gained 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.

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 $i_{\text{max}}$ and $S_{\text{max}}$ enables us to solve rational-II benchmark problems with up to 266 vertices with an II
quality of 0.93. Tuning of $i_{\text{max}}$ and $S_{\text{max}}$ will be investigated in future work. Finally, Pareto frontiers after place & route can
be improved using our approach, thus enabling a more fine-
gained 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.

References
an Easily Schedulable Horizontal Architecture for High-performance
Pipelining Loops,” in Proc. of the 27th Int. Symposium on Microarchi-
[4] S. Dai and Z. Zhang, “Improving Scalability of Exact Modulo Schedul-
ing with Specialized Conflict-Driven Learning,” in Proc. of the 56th
with Rational Initiation Intervals in Custom Hardware Design,” in 25th
Contribution to Agile HLS,” in Int. Workshop on FPGAs for Software
Programmers, 2018.


[47] 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.

[48] John Vickerson (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.

[49] 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.