BLOOP: Boolean Satisifiability-based Optimized Loop Pipelining

NICOLAI FIEGE, University of Kassel, Germany
PETER ZIPF, University of Kassel, Germany

Modulo scheduling is the premier technique for throughput maximization of loops in high-level synthesis by interleaving consecutive loop iterations. The number of clock cycles between data insertions is called initiation interval (II). For throughput maximization, this value should be as low as possible; therefore its minimization is the main optimization goal.

Despite its long historical existence, modulo scheduling always remained a relevant research topic over the last years with many exact and heuristic algorithms available in literature.

Nevertheless, we are able to leverage the scalability of modern Boolean Satisfiability (SAT) solvers to outperform state-of-the-art ILP-based algorithms for latency-optimal modulo scheduling for both integer and rational IIs. Our algorithm is able to compute valid modulo schedules for the whole CHStone and MachSuite benchmark suites, with 99% of the solutions being proven to be throughput-optimal for a timeout of only 10 min per candidate II. For various time limits, not a single tested scheduler from the state-of-the-art is able to compute more verified optimal solutions or even a single schedule with a higher throughput than our proposed approach. Using an HLS toolflow we show that our algorithm can be effectively used to generate Pareto-optimal FPGA implementations regarding throughput and resource usage.

CCS Concepts: • **Hardware → High-level and register-transfer level synthesis**; **Operations scheduling**.

Additional Key Words and Phrases: Modulo scheduling, Loop pipelining, Boolean Satisfiability

ACM Reference Format:
Nicolai Fiege and Peter Zipf. 2023. BLOOP: Boolean Satisifiability-based Optimized Loop Pipelining. *ACM Trans. Reconfig. Technol. Syst.* 16, 3, Article 49 (July 2023), 33 pages. https://doi.org/10.1145/3599972

1 INTRODUCTION

Field-programmable gate arrays (FPGA) received increased interest in many computationally demanding fields over the last decades, such as image and video processing [31, 56], machine learning [33, 37, 53, 55], control engineering [25, 44] or cryptography [35]. This requires productive means of obtaining high-quality circuits that meet all throughput specifications while at the same time being resource and energy efficient. One way of obtaining such circuits is high-level synthesis (HLS), the process of generating digital hardware implementations at register transfer level (RTL) from behavioral descriptions such as C/C++ code [17, 24].

Loop computations take up a significant portion of runtime during program execution [5]. Modulo scheduling[1]—interleaving consecutive loop iterations to exploit instruction-level parallelism—is the premier technique for loop acceleration [46], and is therefore used in many HLS frameworks [10, 36]. Despite its long historical existence and many exact and heuristic modulo scheduling algorithms being published over the last decades [2, 4, 5, 14, 15, 18–20, 34, 41, 45, 48, 57], it still remains one of the most computationally expensive steps in contemporary HLS. The optimization goal is usually

[1] also referred to as "loop pipelining"

Authors' addresses: Nicolai Fiege, nfiege@uni-kassel.de, University of Kassel, Kassel, Germany; Peter Zipf, zipf@uni-kassel.de, University of Kassel, Kassel, Germany.

to minimize the time between consecutive data sample insertions, called the initiation interval (II), under tight resource constraints.

In scheduling, we differentiate between the following cases: (1) *static scheduling* [46], where the execution times of all operations are computed during circuit compilation; (2) *dynamic scheduling* [43], which aims at circumventing conservative data dependencies for static scheduling by determining execution times at runtime, potentially causing an area-overhead for increased performance [30]; (3) a combination thereof [7] aiming at statically scheduling the parts of the input description, where dynamic scheduling cannot bring a performance gain.

In this paper we propose BLOOP[2], an optimal static modulo scheduling algorithm based on Boolean satisfiability (SAT). Our proposed algorithm quickly determines an initial schedule for the optimal II using a problem reduction technique [14] within the framework of iterative modulo scheduling [45] and afterwards performs an iterative search for the optimal schedule length leveraging the scalability of modern SAT solvers. This process combines the main advantage of heuristics (quickly determining a feasible solution) with the main advantage of exact algorithms (determining the optimal solution given enough computation time).

Our contributions are the following:

(1) We present a SAT-based optimal modulo scheduling algorithm under resource constraints (Section 2).
(2) We generalize our algorithm to also support rational IIs (Section 3).
(3) We evaluate our algorithm on several benchmark suites (Section 5) consisting of a diverse selection of modulo scheduling problems and show superiority over the existing state-of-the-art regarding the number of (optimally proven) solutions and algorithm runtime (Sections 5.1–5.2).
(4) We show that FPGA implementations generated with schedules obtained from our proposed scheduler lead to better tradeoffs in the design space regarding resource usage and throughput achieved compared to solutions obtained by state-of-the-art approaches (Section 5.3).

This ensures that our algorithm can prove useful for the evaluation of new heuristics and for the generation of highly optimized digital circuits.

1.1 The modulo scheduling problem

Before giving a motivational example for the importance of optimal modulo scheduling, we formally define the modulo scheduling problem (MSP). Table 1 shows an overview of all inputs to the MSP. One input is the data flow graph (DFG) $G = \{O, E, D, \Delta\}$ which consists of vertices representing the operations $o_i \in O$; weighted, directed edges representing data dependencies $(o_i \rightarrow o_j) \in E$; edge weights representing algorithmic distances $d_{i,j} \in D$; and minimum time differences between consecutive operations due to data dependencies $\delta_{i,j} \in \Delta$. In the default case, such a time difference is the source operation's latency L_i in clock cycles. In order to prohibit excessive chaining of operations with zero latency (e.g., bit-wise or logical operations), dedicated chaining edges are added to the DFG before scheduling [41]. Those additional edges have $\delta_{i,j} = 1$ to force two operations into two different clock cycles whenever the critical path would exceed the maximum clock period. Therefore,

$$\delta_{i,j} = \begin{cases} 0, & \text{if } o_i \text{ and } o_j \text{ are chainable and } L_i = 0, \\ 1, & \text{if } o_i \text{ and } o_j \text{ are not chainable and } L_i = 0, \\ L_i, & \text{else.} \end{cases} \tag{1}$$

[2]Boolean satisfiability-based optimized loop pipelining

Table 1. Parameters of the MSP

Parameter	Definition	Explanation
$o_i \in O$	$\forall i : 0 \leq i < \lvert O \rvert$	operation
$L_i \in \mathbb{N}_0$	$\forall i : 0 \leq i < \lvert O \rvert$	latency of operation i in clock cycles
$(o_i \rightarrow o_j) \in E$	$E \subseteq O \times O$	edge
$d_{i,j} \in D$	$D \subset \mathbb{N}_0$	distance on the edge $(o_i \rightarrow o_j)$
$\delta_{i,j} \in \Delta$	$\Delta \subset \mathbb{N}_0$	minimum number of clock cycles between the execution of o_i and o_j
$\omega \in \Omega$	$\{\text{add}, \text{mult}, \ldots\}$	operator type
$\omega \in \check{\Omega}$	$\check{\Omega} \subseteq \Omega$	resource-limited operator type
$\text{FUs}(\omega) \in \mathbb{N}$	$\forall \omega : \omega \in \check{\Omega}$	number of allocated functional units of resource-limited type ω
$o_i \in \check{O}_\omega$	$\check{O}_\omega \subseteq O$	operation of resource-limited type ω
$\text{II} \in \mathbb{N}$	$\text{II}^\bot \leq \text{II} \leq \text{II}^\top$	initiation interval

The second input to the MSP are the operator limits due to resource constraints. Each operation can be executed by one unique operator type ω. Usually, operators[3] are resource-limited if they are either sparsely available (e.g., memory read/write accesses) or consume significant resources (e.g., floating point operations). Other operations are usually modeled without an operator limit if they can be implemented in hardware in a very resource-efficient way (e.g., bit-wise or logical operations).

The output of the MSP is (i) an II and (ii) a set of non-negative integer start times t_i that, together with the II, satisfy all data dependencies and operator constraints, i.e.,

$$t_j + d_{i,j} \cdot \text{II} \geq t_i + \delta_{i,j} \qquad \forall i, j : (o_i \rightarrow o_j) \in E \tag{2}$$

and

$$\lvert \{ i : o_i \in \check{O}_\omega, t_i \bmod \text{II} = m \} \rvert \leq \text{FUs}(\omega) \qquad \forall \omega, m : \omega \in \check{\Omega}, 0 \leq m < \text{II}, \tag{3}$$

where $\check{O}_\omega$ contains all operations that can be executed by the resource-limited operator ω.

The minimum achievable II is limited by two factors: (i) recurrences in the DFG, and (ii) operator constraints [46]. We assume that all operators can accept new input data in each clock cycle (i.e., they are fully pipelined with L_i pipeline stages). Therefore, n operators can execute at most $n \cdot \text{II}$ operations within II clock cycles, which results in

$$\text{II} \geq \frac{\lvert \check{O}_\omega \rvert}{\text{FUs}(\omega)} \qquad \forall \omega : \omega \in \check{\Omega}. \tag{4}$$

Here, we denote the number of operations that can be executed by an operator of type ω as $\lvert \check{O}_\omega \rvert$. We follow previous work [5, 19, 41, 45, 48] and define the resource-limited minimum II as

$$\text{II}^\bot_{\text{res}} = \max_{\omega \in \check{\Omega}} \left(\frac{\lvert \check{O}_\omega \rvert}{\text{FUs}(\omega)} \right). \tag{5}$$

Recurrences limit the minimum achievable II via (2). The recurrence-limited minimum II, $\text{II}^\bot_{\text{rec}}$, is therefore solely limited by the set of all cycles, C, in G. We follow Tarjan's definition of a cycle [52],

[3] also referred to as "functional units" (FU)

$c_x \in C$, and denote it as

$$c_x = \{(o_a \rightarrow o_b),\ (o_b \rightarrow o_c),\ (o_c \rightarrow\ \ldots\ \rightarrow o_a)\}. \tag{6}$$

This means that a cycle is a sequence of edges, for which (i) the source vertex of an edge is the sink vertex of the previous edge; (ii) the source vertex of the first edge is the sink vertex of the last edge; (iii) all vertices in c_x appear exactly once as a source and once as a sink vertex; and (iv) the first source vertex is the last sink vertex. The subscript x just serves as a unique numbering of all cycles and does not depend on the edges included in c_x. The cyclic dependencies of all resulting dependency constraints lead to the condition

$$\mathrm{II} \geq \frac{\sum\limits_{i,j:(o_i \rightarrow o_j) \in c_x} \delta_{i,j}}{\sum\limits_{i,j:(o_i \rightarrow o_j) \in c_x} d_{i,j}} \qquad \forall x : c_x \in C. \tag{7}$$

$\mathrm{II}^{\perp}_{\mathrm{rec}}$ directly follows [5] as

$$\mathrm{II}^{\perp}_{\mathrm{rec}} = \max_{c_x \in C} \left(\frac{\sum\limits_{i,j:(o_i \rightarrow o_j) \in c_x} \delta_{i,j}}{\sum\limits_{i,j:(o_i \rightarrow o_j) \in c_x} d_{i,j}} \right). \tag{8}$$

For graphs without cycles (i.e., $C = \emptyset$), $\mathrm{II}^{\perp}_{\mathrm{rec}}$ is usually defined as $\mathrm{II}^{\perp}_{\mathrm{rec}} = 1$. We follow previous work [45] and define the minimum II, $\mathrm{II}^{\perp}$, as

$$\mathrm{II}^{\perp} = \max\{\mathrm{II}^{\perp}_{\mathrm{rec}}, \mathrm{II}^{\perp}_{\mathrm{res}}\}. \tag{9}$$

Note that the optimum II is merely lower-bounded by $\mathrm{II}^{\perp}$, because the interaction of resource and precedence constraints can lead to infeasible IIs that are larger than or equal to $\mathrm{II}^{\perp}$.

1.2 Motivational Example

Fig. 1 shows an example to motivate the need for loop pipelining in HLS. Fig. 1(a) is a loop written in C++. We assume that all array elements with indices $i = 0 \ldots 3$ are initialized to 1. The loop iterates 996 times and in each iteration it reads data from three input arrays, $x_0 \ldots x_2$, computes three multiplications and stores the results in the arrays $y_0 \ldots y_2$.

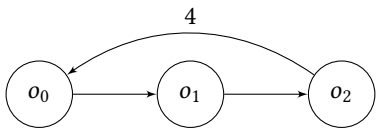
In this example we assume that inputs (i.e., x_0, x_1, x_2) are available to the circuit via a parallel streaming interface and all outputs (i.e., y_0, y_1, y_2) are made available to the surrounding circuit via another parallel streaming interface. This allows us to omit modeling (possibly limited) load/store operations, which—due to their latency—might even change $\mathrm{II}^{\perp}_{\mathrm{rec}}$. Our experimental evaluation in Section 5 considers both memory models. Fig. 1(b) shows the corresponding DFG. Note the edge $o_2 \rightarrow o_0$ with a distance of $d_{2,0} = 4$. It represents the data dependency from y_0 on y_2 from four iterations before. Edge weights of intra-loop dependencies ($d_{i,j} = 0$) are not explicitly shown in the DFG to improve clarity. Multiplications on an FPGA can be efficiently implemented using embedded DSPs. We assume that they are pipelined with two pipeline stages leading to a latency of two clock cycles per operation.

The graph contains exactly one cycle, $c_0 = \{(o_0 \rightarrow o_1), (o_1 \rightarrow o_2), (o_2 \rightarrow o_0)\}$, which results in $\mathrm{II}^{\perp}_{\mathrm{rec}} = \frac{2+2+2}{0+0+4} = \frac{3}{2}$. For modulo scheduling with integer IIs, the minimum integer II, $\mathrm{II}^{\perp}_{\mathbb{N}}$, is defined as $\mathrm{II}^{\perp}_{\mathbb{N}} = \lceil \mathrm{II}^{\perp} \rceil$. Without additional resource constraints, $\mathrm{II}^{\perp}_{\mathbb{N}} = 2$. For rational IIs, the II does not need to be an integral number [48]. Therefore, $\mathrm{II}^{\perp}_{\mathbb{Q}} = \mathrm{II}^{\perp}$ and in this example $\mathrm{II}^{\perp}_{\mathbb{Q}} = \frac{3}{2}$.

Table 2(a) shows a non-modulo schedule for the given scheduling problem. It uses one FU and has a schedule length of six. Therefore, a new data sample can be introduced every six clock cycles—denoted by the small triangles in time steps 0, 6 and 12.

```
for (int i=4; i<1000; i++) {
    y_0[i] = x_0[i] * y_2[i-4]; // o_0
    y_1[i] = x_1[i] * y_0[i];   // o_1
    y_2[i] = x_2[i] * y_1[i];   // o_2
}
```

(a) Example for-loop



(b) Example DFG

Fig. 1. Motivational example

Table 2. Comparison of different schedules for the DFG shown in Figure 1(b). The notation $s : o_i$ stands for "*operation represented by vertex o_i of sample with index s*".

(a) non-modulo

↓ t		FU1
0	▷	$0 : o_0$
1		
2		$0 : o_1$
3		
4		$0 : o_2$
5		
6	▷	$1 : o_0$
7		
8		$1 : o_1$
9		
10		$1 : o_2$
11		
12	▷	$2 : o_0$
13		
14		$2 : o_1$

(b) II = 3

	FU1
▷	$0 : o_0$
	$0 : o_1$
▷	$1 : o_0$
	$0 : o_2$
	$1 : o_1$
▷	$2 : o_0$
	$1 : o_2$
	$2 : o_1$
▷	$3 : o_0$
	$2 : o_2$
	$3 : o_1$
▷	$4 : o_0$
	$3 : o_2$
	$4 : o_1$

(c) II = 2

	FU1	FU2
▷	$0 : o_0$	
▷	$0 : o_1$	$1 : o_0$
▷	$2 : o_0$	$1 : o_1$
	$0 : o_2$	
▷	$2 : o_1$	$3 : o_0$
		$1 : o_2$
▷	$4 : o_0$	$3 : o_1$
	$2 : o_2$	
▷	$4 : o_1$	$5 : o_0$
		$3 : o_2$
▷	$6 : o_0$	$5 : o_1$
	$4 : o_2$	
▷	$6 : o_1$	$7 : o_0$

(d) II = $^3/_2$

	FU1	FU2
▷	$0 : o_0$	
▷	$0 : o_1$	$1 : o_0$
▷	$2 : o_0$	
	$0 : o_2$	$1 : o_1$
▷	$2 : o_1$	$3 : o_0$
▷	$4 : o_0$	$1 : o_2$
	$2 : o_2$	$3 : o_1$
▷	$4 : o_1$	$5 : o_0$
▷	$6 : o_0$	$3 : o_2$
	$4 : o_2$	$5 : o_1$
▷	$6 : o_1$	$7 : o_0$
▷	$8 : o_0$	$5 : o_2$
	$6 : o_2$	$7 : o_1$
▷	$8 : o_1$	$9 : o_0$

By allowing modulo scheduling, the throughput can be increased by a factor of two without allocating more FUs, as seen in Table 2(b). The FU is fully utilized which means that throughput cannot be increased anymore, without allocating more resources.

With two FUs, an integer II = 2 is possible (see Table 2(c)). Even when the pipeline is full (beginning at $t = 4$), the two FUs are only utilized 75% of the time. Allowing the II to vary allows for a further increase in throughput as seen in Table 2(d). Here, the II alternates between one and two clock cycles, resulting in an *average* II of $\frac{3}{2}$. Starting at $t = 4$, the pipeline is full and both FUs are fully utilized. Now, an increase in available FUs cannot lead to an increase in throughput anymore, because the recurrence limits the minimum II to $\text{II}^{\perp} = \frac{3}{2}$.

Two conclusions can be drawn from this example:

(1) Modulo scheduling is a particularly effective method for throughput enhancement.
(2) Allowing the II to be a rational number can lead to even higher throughput for some resource allocations than constraining $\text{II} \in \mathbb{N}$.

2 THE ALGORITHM

Proving satisfiability (or unsatisfiability) of a Boolean function is an NP-complete problem [12]. For efficient contemporary SAT solving algorithms, the function of interest has to be in the conjunctive normal form (CNF). This means that it consists of a conjunction of clauses; each clause is a disjunction of literals; and each literal is a Boolean variable or its negation.

Before describing the SAT formulation, we define the notation used in the following: $x \vee y$ is the disjunction of x and y, $x \wedge y$ is the conjunction of x and y, and $\neg x$ is the negation of x. Using this notation, the CNF of, e.g., an XOR relation between x and y is given as

$$f(x, y) = (x \vee y) \wedge (\neg x \vee \neg y). \tag{10}$$

2.1 SAT formulation

We start by introducing a SAT formulation for the following question:

GIVEN. *A graph $G = \{O, E, D, \Delta\}$, an initiation interval II, a maximum schedule length $\hat{L}$, a set of operators Ω.*

QUESTION. *Does a set of time steps $\{t_i : o_i \in O\}$ exist such that precedence constraint (2) is fulfilled for each edge $(o_i \rightarrow o_j) \in E$ and resource constraint (3) is fulfilled for each resource-limited operator $\omega \in \check{\Omega} \subseteq \Omega$?*

For each operation $o_i \in O$ we firstly compute the earliest and latest start times, $t_i^{\perp}$ and $t_i^{\top}$, respectively. We obtain these values using an ASAP and an ALAP schedule without operator constraints. While the ASAP scheduler automatically determines the minimum possible schedule length, we constrain the ALAP schedule to $\hat{L}$.

2.1.1 Boolean variables. Knowledge about earliest and latest possible execution times allows us to allocate one variable

$$t_{i,\tau} \qquad \forall i, \tau \ : \ o_i \in O, t_i^{\perp} \leq \tau \leq t_i^{\top} \tag{11}$$

per operation o_i and possible time slot τ in which this operation can be scheduled.

To ensure that we are not oversubscribing any resource-constrained operator in any congruence class modulo II, we also allocate one variable

$$b_{i,\beta} \qquad \forall i, \beta \ : \ o_i \in \check{O}_{\omega}, 0 \leq \beta < \mathrm{FUs}(\omega), \omega \in \check{\Omega} \tag{12}$$

per resource-constrained operation o_i and possible operator that this vertex can be assigned to.

Additionally, we declare

$$T_{i,j} \qquad \forall i, j \ : \ o_i \in \check{O}_{\omega}, o_j \in \check{O}_{\omega}, i > j, \omega \in \check{\Omega} \tag{13}$$

and

$$B_{i,j} \qquad \forall i, j \ : \ o_i \in \check{O}_{\omega}, o_j \in \check{O}_{\omega}, i > j, \omega \in \check{\Omega} \tag{14}$$

for all pairs of operations of all resource-constrained operators. $T_{i,j}$ denotes whether o_i and o_j are scheduled in different congruence classes and $B_{i,j}$ states whether o_i and o_j are bound to different operators. Even though the introduction of these variables is not strictly necessary to model the MSP under a schedule length constraint for a given II, DFG and operator allocation, we propose to include them to greatly reduce the necessary number of clauses.

2.1.2 Time and resource assignment constraints. We ensure that each operation is assigned to at least one time step by adding the clauses

$$\bigvee_{\tau=t_i^{\perp}}^{t^{\top}} t_{i,\tau} \qquad \forall i \ : \ o_i \in O \tag{15}$$

for all operations. Note that we are not forcing the assignment to *exactly* one time step, as this would need more clauses to model. Furthermore, the assignment to *at least* one time step suffices, since the SAT solver will make sure that all dependency and resource constraints are obeyed for *each* assigned time step. In the spirit of schedule length minimization we always choose the smallest one.

Similarly, we ensure the assignment of each operation to at least one operator with clauses

$$\bigvee_{\beta=0}^{\mathrm{FUs}(\omega)-1} b_{i,\beta} \qquad \forall i, \omega \ : \ o_i \in \check{O}_{\omega}, \omega \in \check{\Omega}. \tag{16}$$

2.1.3 Dependency constraints. For a valid modulo schedule, each data dependency, imposed by (2), must hold for each edge $(o_i \rightarrow o_j) \in E$. Therefore, we pre-compute all execution times for all pairs of operations connected via an edge and add the clause

$$\neg t_{i,\tau_i} \vee \neg t_{j,\tau_j} \qquad \forall \ i, j, \tau_i, \tau_j \ :$$

$$(o_i \rightarrow o_j) \in E, t_i^{\perp} \leq \tau_i \leq t_i^{\top}, t_j^{\perp} \leq \tau_j \leq t_j^{\top}, \tau_j + d_{i,j} \cdot \mathrm{II} < \tau_i + \delta_{i,j} \tag{17}$$

whenever (2) is violated.

2.1.4 Resource constraints. Aside from data dependencies, we must also ensure that each operator executes at most one operation per clock cycle. For this, we assume that each operator is fully pipelined (i.e., it has a blocking time of one clock cycle) and can thus accept new input data in each clock cycle.

Since an operator can only execute at maximum one operation per clock cycle, we must ensure that all operations of an operator type are either scheduled in different congruence classes or are executed by different operators. This can, for example, be ensured by adding the clause

$$\neg t_{i,\tau_i} \vee \neg t_{j,\tau_j} \vee \neg b_{i,\beta} \vee \neg b_{j,\beta} \qquad \forall \ i, j, \tau_i, \tau_j, \beta \ :$$

$$0 \leq \beta < \mathrm{FUs}(\omega), o_i \in \check{O}_{\omega}, o_j \in \check{O}_{\omega}, \omega \in \check{\Omega}, i < j, t_i^{\perp} \leq \tau_i \leq t_i^{\top}, t_j^{\perp} \leq \tau_j \leq t_j^{\top}, \tau_i \bmod \mathrm{II} = \tau_j \bmod \mathrm{II} \tag{18}$$

for each resource-constrained operator, each pair of operations that can be executed by that operator type and each congruence class. The number of these clauses rapidly grow for a large number of allocated operators. Thus, we propose to instead use the clause

$$T_{i,j} \vee B_{i,j} \qquad \forall i, j \ : \ o_i \in \check{O}_{\omega}, o_j \in \check{O}_{\omega}, i > j, \omega \in \check{\Omega} \tag{19}$$

with $T_{i,j}$ implying that o_i and o_j are scheduled in different congruence classes and $B_{i,j}$ implying that o_i and o_j are bound to different operators. This means that we have to make sure that $T_{i,j}$ and $B_{i,j}$ are correctly set to zero if two operations are scheduled in the same congruence classes or are bound to the same operator. We ensure this by adding the clauses

$$\neg T_{i,j} \vee \neg t_{i,\tau_i} \vee \neg t_{j,\tau_j} \qquad \forall i, j, \tau_i, \tau_j \ :$$

$$o_i \in \check{O}_{\omega}, o_j \in \check{O}_{\omega}, i > j, \omega \in \check{\Omega}, t_i^{\perp} \leq \tau_i \leq t_i^{\top}, t_j^{\perp} \leq \tau_j \leq t_j^{\top}, \tau_i \bmod \mathrm{II} = \tau_j \bmod \mathrm{II} \tag{20}$$

and

$$\neg B_{i,j} \vee \neg b_{i,\beta} \vee \neg b_{j,\beta} \qquad \forall i, j, \beta \,:\, o_i \in \check{O}_\omega, o_j \in \check{O}_\omega, i > j, \omega \in \check{\Omega}, 0 \leq \beta < \mathrm{FUs}(\omega). \tag{21}$$

In case an operator of type ω with blocking time $\lambda_\omega > 1$ starts an operation in time step $\hat{t}$, it cannot start a new operation until time step $\hat{t} + \lambda_\omega$. Although we do not consider and evaluate this case in this paper, one could easily encode this property in our SAT formulation by adding Clause (20) to the solver whenever $(\tau_i + \lambda_i) \bmod \mathrm{II} = (\tau_j + \lambda_j) \bmod \mathrm{II}\ \forall \lambda_i, \lambda_j \in [0,\ \lambda_\omega - 1]$.

2.2 Schedule length minimization

We use Algorithm 1 for schedule length minimization given an II, a graph G, a set of allocated operators Ω and a timeout. We start by constructing an empty schedule and setting a *success* variable to false, indicating that we have not found a valid solution, yet. Then, in Line 3, we enter a loop which breaks by a return statement when we either proved that the II is infeasible, found the optimum, or encountered a timeout.

The first task in each loop iteration is to compute the next candidate schedule length. This can be done by, e.g., a binary search between a lower and an upper bound.

The function `computeNextScheduleLength` returns an invalid number in case the whole solution space for $\hat{\mathrm{L}}$ was searched in previous loop iterations. We use this information to stop the search in Line 5. A lower bound for $\hat{\mathrm{L}}$ ($\mathrm{L}^\perp$) can be trivially constructed by performing a resource-unconstrained ASAP schedule. Obviously, this schedule length is not always feasible under resource constraints. In these cases, we rely on the SAT solver to quickly prove infeasibility. There are several possibilities to construct an upper bound on $\hat{\mathrm{L}}$ ($\mathrm{L}^\top$). Examples are (i) using a heuristic modulo scheduler like [5, 14, 34] or (ii) computing a mathematical upper bound like the one proposed in [41]. We choose option (i) but want to emphasize that the chosen heuristic modulo scheduler plays an important role regarding algorithm runtime and quality of results. First of all, not all heuristic schedulers guarantee optimality regarding the II. Secondly, we will show in Section 5.2 that heuristic modulo schedulers are not even guaranteed to be faster than our proposed algorithm. In Section 2.4 we will show how to use the problem reduction technique proposed in [14] to preserve optimality regarding the II while still quickly generating a feasible modulo schedule (if it exists) to upper bound $\hat{\mathrm{L}}$.

Dai & Zhang [14] proposed to use the length of a heuristic non-modulo schedule to derive $\mathrm{L}^\top$. Even though this approach can work in practice, it is trivial to construct an example for which the schedule length of a non-modulo schedule is shorter than the optimal schedule length for a modulo schedule (e.g., in our motivating example from Section 1.2, the non-modulo schedule is shorter than the optimal modulo schedule for $\mathrm{II} = 2$). We therefore choose to not follow their approach to preserve exactness of our proposed algorithm.

Finally, in Lines 8–11, we use the SAT-based scheduler described in Section 2.1 to either compute a valid schedule for the II, DFG, operator allocation, maximum schedule length or to prove that it does not exist for these input parameters.

2.2.1 Computing the next candidate schedule length. The most simple way to find the optimum schedule length is to start with $\hat{\mathrm{L}} = \mathrm{L}^\perp$ and increment it any time the SAT solver reports infeasibility for that $\hat{\mathrm{L}}$. The first feasible $\hat{\mathrm{L}}$ is then guaranteed to be the optimal schedule length, as all lower values were previously proven to be infeasible. A scheduling problem is proven to be infeasible for the given II if the SAT solver reports unsatisfiability for $\hat{\mathrm{L}} = \mathrm{L}^\top$. This approach has the downside that the number of SAT problems to solve grows with $O(\mathrm{L}^\top - \mathrm{L}^\perp)$.

A more sophisticated approach would be a binary search. In this case, the number of SAT problems grows with $O(\log_2(\mathrm{L}^\top - \mathrm{L}^\perp))$. For conservative $\mathrm{L}^\top$, the disadvantage of this approach is that $\hat{\mathrm{L}}$

Algorithm 1 Latency-optimal modüulo scheduling

Require: II, G, Ω, timeout
Ensure: A valid schedule $\mathbb{S}$ satisfying all resource and precedence constraints

```
1:  𝕊 ← ∅                                                    ▷ empty schedule
2:  success ← false
3:  while true do                                            ▷ break by return
4:      L̂ ← computeNextScheduleLength(success)
5:      if L̂ is invalid or time() > timeout then
6:          return 𝕊                                         ▷ II infeasible/found optimum/timeout
7:      end if
8:      𝕊̂, success ← schedule(II, G, Ω, L̂, timeout)          ▷ using the SAT formulation in Section 2.1
9:      if success then
10:         𝕊 ← 𝕊̂
11:     end if
12: end while
```

already becomes very large on the second try (i.e., $\frac{L^{\perp}+L^{\top}}{2}$)—in practice it is usually much higher than the optimal schedule length if the candidate II is feasible. Since the number of variables in the SAT formulation grows quadratically[4] and the number of clauses to model resource and precedence constraints even grows quartically[5], a binary search does not scale well to larger problem sizes for conservative estimations of $L^{\top}$. In Section 4.2 we give a more detailed explanation how our SAT formulation scales w.r.t. the problem size (i.e., the input parameters given in Table 1).

Therefore, we propose to combine the two approaches in a search for which the number of SAT problems to solve scales with $O(\sqrt{L^{\top} - L^{\perp}})$. We start by doing a linear search, and increment $\hat{L}$ by $k \cdot \left\lceil \sqrt{L^{\top} - L^{\perp}} \right\rceil$ each time $\hat{L}$ is proven infeasible. Once we found a valid solution, we might have bypassed the optimum schedule length. Therefore, we start iteratively decreasing $\hat{L}$ by one until we prove an $\hat{L}$ infeasible, again. Then, the last feasible $\hat{L}$ is the optimum for the given II. By empirical studies we found that choosing $k = 0.5$ leads to a low number of necessary iterations for practical scheduling problems.

2.3 Initiation interval minimization

We follow previous work [5, 19, 41, 45] and use iterative modulo scheduling via Algorithm 2 to compute a schedule for the optimum II. This algorithm works by first computing lower ($II^{\perp}$) and upper ($II^{\top}$) bounds for the II and then iteratively tries to solve the scheduling problem for a given (constant) II. Lower bounds on the II are due to operator constraints and recurrences in the DFG as described in Section 1.2. An upper bound can be computed by using a non-modulo scheduler. We use the ASAP scheduler with heuristic resource constraints from the HatScheT library [49]. To limit the computation time per scheduling problem, the user can provide a time budget per candidate II and a maximum number of allowed scheduling attempts, maxRuns.

2.4 Fast initial schedule computation

There are two critical downsides of the previously outlined approach to solve the MSP:

[4]The number of all variables grows quadratically w.r.t. the problem size.
[5]The number of clauses (20) and (21) grow with the fourth power w.r.t. the problem size.

Algorithm 2 Iterative modulo scheduling

Require: G, Ω, timeout, maxRuns ▷ default: maxRuns = ∞
Ensure: A valid schedule $\mathbb{S}$
1: $\text{II}^{\perp} \leftarrow$ `computeMinII()`
2: $\text{II}^{\top} \leftarrow$ `computeMaxII()`
3: **for** $i \leftarrow 0 \ldots \text{maxRuns} - 1$ **do**
4: $\quad \text{II} \leftarrow \text{II}^{\perp} + i$
5: $\quad$ **if** $\text{II} = \text{II}^{\top}$ **then**
6: $\quad\quad$ **return** $\emptyset$ ▷ failed to find schedule
7: $\quad$ **end if**
8: $\quad \mathbb{S} \leftarrow$ `schedule`$(\text{II}, G, \Omega, \text{timeout})$ ▷ Alg. 1
9: $\quad$ **if** $\mathbb{S} \neq \emptyset$ **then** ▷ a schedule for the II exists
10: $\quad\quad$ **return** $\mathbb{S}$ ▷ found schedule
11: $\quad$ **end if**
12: **end for**
13: **return** $\emptyset$ ▷ failed to find schedule (timeout or no modulo schedule exists)

(1) The upper bound for $\hat{\text{L}}$ proposed in [41] usually overestimates the actual schedule length by an order of magnitude which means that many large SAT problems must be solved before classifying an II as *infeasible*.
(2) As $\text{II}^{\perp}_{\text{rec}}$ and $\text{II}^{\perp}_{\text{res}}$ are merely lower bounds on the achievable II, it is possible that multiple IIs must be proven as *infeasible* before the first feasible II is encountered. Using Algorithm 1, this would require solving numerous SAT programs before the scheduler reaches a feasible II.

To overcome these two shortcomings, we adopt Dai & Zhang's approach [14] of partitioning the graph into Strongly Connected Components (SCC) [52] to simplify the scheduling problem. The goal is to create *one* SAT problem that must be solved to either prove infeasibility for the given II or to quickly generate an initial schedule that results in a better (i.e., lower) upper bound on the schedule length than the one proposed by Oppermann et al. [41]. Additionally, the resulting SAT problem should, in general, be easier to solve than the overall scheduling problem.

Partitioning the graph into SCCs yields a set of subgraphs for which each subgraph is an SCC. An SCC is defined to contain paths between each pair of vertices that only include edges inside the SCC [52]. Therefore, each vertex inside an SCC is, by definition, part of at least one cycle.

We use Algorithm 3 to compute an initial schedule. We start by partitioning the graph into SCCs using Tarjan's algorithm [52]. Each SCC can be classified into *trivial*, *basic* or *complex*. Trivial SCCs only contain one vertex that is either part of no recurrence or only consists of a self-loop. Basic SCCs are non-trivial SCCs without any resource limits and complex SCCs are non-trivial SCCs that contain at least one resource-limited vertex. Therefore, when deciding whether an II is feasible, it suffices to only consider the complex SCCs.

After partitioning the graph, we initialize an empty graph and sets to store the earliest and latest start times for all vertices of complex SCCs. Since we omit inter-SCC connections for the decision whether an II is feasible, the for-loop in Line 5 can consider each SCC individually regarding its maximum schedule length and the earliest and latest start times of its vertices.

A possibility to decide the maximum schedule length for an SCC would be using the upper bound by Oppermann et al. [41]. Due to its overestimation of the actual schedule length, we would severely overcomplicate the SAT problem. To the best knowledge of the authors, the upper bound by Oppermann et al. is the best upper bound on the schedule length currently available in literature.

Algorithm 3 SCC-based modulo scheduling

Require: G, Ω, II, timeout
Ensure: A valid schedule $\mathbb{S}$

```
SCCs ← partition(G)                                   ▷ compute SCCs
G_S ← {}                                              ▷ init. empty graph
T^⊥ ← {}                                              ▷ init empty set of earliest start times
T^⊤ ← {}                                              ▷ init empty set of latest start times
for each scc in SCCs do
    if scc is complex then
        L^⊤_scc ← maxScheduleLength(scc, Ω, II)
        T^⊥_scc ← earliestTimes(scc, Ω, II, L^⊤_scc)
        T^⊤_scc ← latestTimes(scc, Ω, II, L^⊤_scc)
        G_S ← G_S ∪ scc                               ▷ insert SCC into graph
        T^⊥ ← T^⊥ ∪ T^⊥_scc                           ▷ insert earliest start times
        T^⊤ ← T^⊤ ∪ T^⊤_scc                           ▷ insert latest start times
    end if
end for
S_scc ← schedule(II, G_S, Ω, timeout, T^⊥, T^⊤)
if S_scc = ∅ then
    return ∅                                          ▷ timeout or II is infeasible
end if
for each scc in SCCs do
    if scc is basic then
        S_B ← scheduleSDC(II, scc, timeout)           ▷ schedule basic SCC
        S_scc ← S_scc ∪ S_B
    end if
end for
S ← assembleSchedule(S_scc, SCCs, G, Ω, II)           ▷ compute final schedule from relative schedules
return S
```

Therefore, we propose to use the ILP formulation shown in Fig. 2 to compute the maximum schedule length of a given SCC. As the ILP does not consider any operator constraints and we consider each SCC separately, the ILP's runtime is negligible compared to the time to solve the overall scheduling problem. Constraint **A** is the standard dependency constraint given in (2). To maximize the schedule length, we create two additional variables: t^- and t^+. If we ensure that t^- is equal to the earliest start time and t^+ is equal to the latest end time of any operation, we can compute the maximum schedule length by maximizing the difference between these two values.

For $x_i = 1$, Constraint **B** simplifies to $t^- - t_i \geq 0$. This would ensure t^- is at least as large as t_i. If $x_i = 0$, the constraint is always satisfied for a sufficiently large value of C.

The same logic applies to Constraint **C**. For $y_i = 1$, it simplifies to $t^+ - t_i \leq L_i$, which ensures that t^+ is at most as large as the end time of o_i. Similarly, $y_i = 0$ disables the constraint. **B** and **C** are called big-M constraints[6]. Note that we must choose the value for the big-M constant C such that **B** and **C** are correctly disabled for all values that t_i can take.

[6]Usually, the constant in a big-M constraint is called M—hence the name "big-M". In this paper we use C instead because we already use M to express the cycle length of a rational-II modulo schedule.

$$\max(t^+ - t^-)$$

subject to

$$\textbf{A:} \quad t_j - t_i \geq \delta_{i,j} - d_{i,j} \cdot \Pi \quad \forall i, j \ : \ (o_i \rightarrow o_j) \in E_{SCC}$$

$$\textbf{B:} \quad t^- - t_i - C \cdot x_i \geq -C \quad \forall i \ : \ o_i \in O_{SCC}$$

$$\textbf{C:} \quad t^+ - t_i + C \cdot y_i \leq L_i + C \quad \forall i \ : \ o_i \in O_{SCC}$$

$$\textbf{D:} \quad \sum_{i:o_i \in O_{SCC}} x_i \geq 1$$

$$\textbf{E:} \quad \sum_{i:o_i \in O_{SCC}} y_i \geq 1$$

$$t_i, t^-, t^+ \in \mathbb{N}_0 \qquad x_i, y_i \in \{0, 1\}$$

Fig. 2. ILP formulation for maximum schedule length computation of an SCC

By adding Constraints **D** and **E**, we ensure that at least one **B** constraint and at least one **C** constraint is active. It should be noted that this ILP formulation is only bounded if the graph is an SCC, which is given in our case.

Additionally, we can reduce the complexity of the ILP if we only add Constraints **B** and **D** for operations that have no incoming edges with a distance of zero, and we only add Constraints **C** and **E** for operations without outgoing edges that have a distance of zero.

After successfully computing the schedule length of an SCC, we can use this information to determine earliest and latest start times of all SCC vertices. Trivially, we could set the earliest start time of o_i to zero and the latest start time to $L^\top_{SCC} - L_i$ but this would require more SAT variables than necessary in the scheduling attempt in Line 15 of Algorithm 3.

Another possibility would be using resource-unconstrained ASAP and ALAP schedules. They also under-/overestimate earliest/latest start times because edges with $d_{i,j} > 0$ are ignored. Therefore, we use the ILP formulation in Fig. 3. We use Objective ξ_1 to determine earliest and ξ_2 to determine

$$\xi_1 : \quad \min \sum_{i:o_i \in O_{SCC}} t_i$$

$$\xi_2 : \quad \max \sum_{i:o_i \in O_{SCC}} t_i$$

subject to

$$\textbf{A:} \quad t_j - t_i \geq \delta_{i,j} - d_{i,j} \cdot \Pi \quad \forall i, j \ : \ (o_i \rightarrow o_j) \in E_{SCC}$$

$$\textbf{B:} \quad 0 \leq t_i \leq L^\top_{SCC} - L_i \quad \forall i \ : \ o_i \in O_{SCC}$$

Fig. 3. ILP formulation for earliest and latest start time computations of an SCC

latest start times. Again, Constraint **A** ensures that all dependencies are honored and Constraint **B** is used to obey the previously calculated schedule length.

In Line 15 of Algorithm 3 we compute a schedule for the graph that contains all complex SCCs (i.e., G_S) using the SAT formulation described in Section 2.1 with $t_i^\perp$ and $t_i^\top$ defined via the results

from Lines 8 and 9. If we fail to compute a schedule (either due to a timeout or because the II is infeasible), we can directly stop the scheduling process for the given II. In case we successfully compute a schedule, we can use these start times as a relative schedule that we later (in Line 25) use to construct a final schedule for the overall scheduling problem.

We use the for-loop in Line 19 to compute relative schedules for all basic SCCs. Since they do not comprise any resource limitations, we can schedule them individually with any SDC solver and the objective of schedule length minimization.

For completeness we assume that the relative schedule times of all trivial SCCs are zero. Then we use all the previously determined relative schedules to construct a final schedule as described by Dai & Zhang (Algorithm 1 in [14]). We iterate through the topologically sorted list of SCCs and for each SCC we determine the minimum offset that we have to add to all relative schedule times of that SCC such that all dependencies are honored. For trivial and basic SCCs, we can choose any number for that offset because we do not need to account for the congruence classes determined by the relative schedule. For offsets of complex SCCs we must choose a multiple of the candidate II to avoid changing the congruence class of any resource-limited operation.

Finally, it should be noted that this algorithm, in general, cannot guarantee schedule length optimality. This is no problem because we can always refine the schedule length using Algorithm 1 once we know that the II is feasible and using the obtained schedule length as an upper bound for $\hat{L}$ if the time budget is not exhausted after completing Algorithm 3.

2.5 Algorithm summary

Fig. 4 shows an overview of our proposed algorithm BLOOP. It consists of two parts: the first part tries to identify the optimal II and the second part identifies the optimal schedule length. We start the search for the optimal II with $II^{\perp}$ defined via (9). We check whether the candidate II is feasible via the SCC scheduling algorithm described in Section 2.4 (Algorithm 3). Every time an II is proven to be infeasible under the given dependency and resource constraints, we increase the II by one until we encounter a feasible II. Since all IIs between $II^{\perp}$ and that II are proven to be infeasible, it must be optimal. At this point, our scheduler has determined a valid schedule for the optimal II. Due to the SCC graph reduction, the schedule length is not necessarily optimal, which is why we now proceed with the original graph as an input to all subsequent scheduling attempts. We then start with the schedule length search algorithm outlined in Section 2.2.1 (Algorithm 1) using the schedule length of the result from Algorithm 3 as an upper bound. This procedure guarantees that, at the end, we are left with a schedule for the optimal II and the optimal schedule length.

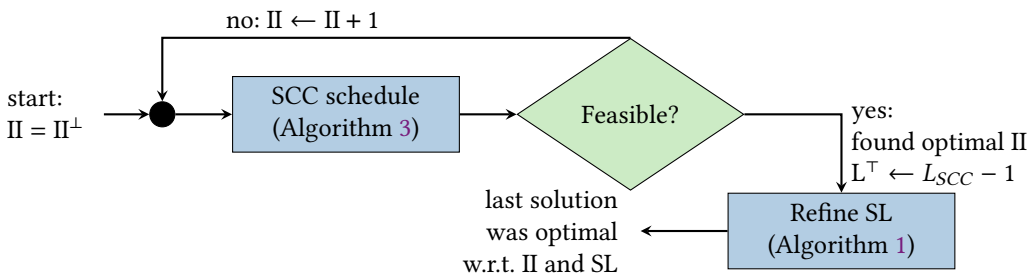


Fig. 4. Flowchart for BLOOP

3 EXTENSION TO RATIONAL IIS

Rational IIs are a technique for throughput maximization [48]. Here, the II is expressed as a rational number instead of an integer one: $\mathrm{II} = \frac{\mathrm{M}}{\mathrm{S}}$. The schedule always repeats itself after M clock cycles and S data samples are processed during this time. Like for integer IIs, the throughput of a rational-II schedule is defined as the reciprocal of the II, namely $\frac{\mathrm{S}}{\mathrm{M}}$.

Recall that an integer-II scheduling algorithm tries to compute a schedule for one data sample in such a way that repeating it every II time steps fulfills all precedence relations and resource constraints [46]. A rational-II scheduling algorithm, on the other hand, computes a schedule for the first S data samples such that repeating it every M time steps fulfills all precedence relations and resource constraints [48]. Therefore, solving scheduling problems for rational IIs is even more complex than solving them for integer IIs.

For the following notation, we always add the superscript (s) to indicate a dependency on the data sample s. This should *not* be confused with exponentiation. For example, $t_i^{(s)}$ denotes the start time of o_i in data sample s. Due to periodicity of the schedule, it suffices to only consider the first S data samples.

Previous work used *uniformity* of the schedule to reduce complexity of the resulting optimization problem for a better scalability of rational-II scheduling to large problem sizes [22]. In a uniform schedule, start times of all operations in the n^{th} data sample are offset by the same constant compared to their respective operations in the 0^{th} sample (e.g., for an integer II, this constant is $n \cdot \mathrm{II}$). Mathematically, this relation is expressed as

$$t_0^{(s)} - t_0^{(0)} = \ldots = t_{|O|-1}^{(s)} - t_{|O|-1}^{(0)} \qquad \forall s : 0 \leq s < \mathrm{S}. \tag{22}$$

Using (22), we follow previous work [48] and define the insertion time of a sample, $I^{(s)}$, as

$$I^{(s)} = t_i^{(s)} - t_i^{(0)} \qquad \forall s : 0 \leq s < \mathrm{S}. \tag{23}$$

This allows us to define the periodically changing initiation interval as

$$\mathrm{II}^{(s)} = \begin{cases} I^{(s+1)} - I^{(s)} & \text{if } s < \mathrm{S} - 1 \\ \mathrm{M} - I^{(s)} & \text{if } s = \mathrm{S} - 1. \end{cases} \tag{24}$$

In the dependency constraint for integer IIs (2) we use the term $d_{i,j} \cdot \mathrm{II}$ to represent a precedence relation between two operations separated by $d_{i,j}$ data samples. A simple product suffices because the II is constant over time. For uniform schedules with rational IIs, this is not the case. The II depends on the sample index as defined via (24). For two operations, o_i and o_j, separated via an edge with a distance of $d_{i,j}$ we use

$$\Delta^{(s)}(d_{i,j}) = \sum_{n=1}^{d_{i,j}} \mathrm{II}^{((s-n) \bmod \mathrm{S})} \tag{25}$$

to replace the term $d_{i,j} \cdot \mathrm{II}$ and get

$$t_i + \delta_{i,j} - \Delta^{(s)}(d_{i,j}) \leq t_j. \tag{26}$$

Note that $\Delta^{(s)}(d_{i,j})$ also depends on s. Rearranging yields

$$t_i - t_j \leq \Delta^{(s)}(d_{i,j}) - \delta_{i,j}. \tag{27}$$

This produces S separate dependency constraints which only differ in their right-hand sides, namely the term $\Delta^{(s)}(d_{i,j})$. As described by Sittel et al. [48], it suffices to only account for the most severe constraint (i.e., the one with the minimum value of Δ_s),

$$t_i + \delta_{i,j} - \Delta_{\min}(d_{i,j}) \leq t_j, \tag{28}$$

with

$$\Delta_{\min}(d_{i,j}) = \min_{0 \leq s < \mathrm{S}} (\Delta^{(s)}(d_{i,j})). \tag{29}$$

3.1 Extending the SAT formulation to rational IIs

By assuming uniformity and using the pre-calculated insertion times, it suffices to only model the schedule time of the first sample insertion of all operations. This means that $t_{i,\tau}$ is true if and only if o_i in data sample 0 is scheduled into time slot τ. All remaining schedule times trivially follow from (23).

From the assignment of S separate time slots to each operation follows that each operation must appear S times in the set in (3). Therefore, we must allocate one binding variable

$$b_{i,\beta}^{(s)} \qquad \forall i, \beta, s \ : o_i \in \check{O}_\omega, 0 \leq \beta < \mathrm{FUs}(\omega), \omega \in \check{\Omega}, 0 \leq s < \mathrm{S} \tag{30}$$

per resource-constrained operation o_i, possible operator that this vertex can be assigned to and each data sample considered in the rational-II schedule.

Similar to the integer-II formulation (cf. Section 2.1), we declare

$$T_{i,j}^{(s_i,s_j)} \qquad \forall i, j, s_i, s_j \ : o_i \in \check{O}_\omega, o_j \in \check{O}_\omega, i > j, \omega \in \check{\Omega}, 0 \leq s_i < \mathrm{S}, 0 \leq s_j, < \mathrm{S} \tag{31}$$

and

$$B_{i,j}^{(s_i,s_j)} \qquad \forall i, j, s_i, s_j \ : o_i \in \check{O}_\omega, o_j \in \check{O}_\omega, i > j, \omega \in \check{\Omega}, 0 \leq s_i < \mathrm{S}, 0 \leq s_j, < \mathrm{S} \tag{32}$$

for all pairs of operations of all resource-constrained operators. Now, $T_{i,j}^{(s_i,s_j)}$ denotes whether o_i in sample s_i and o_j in sample s_j are scheduled in different congruence classes modulo M. Correspondingly, $B_{i,j}^{(s_i,s_j)}$ states whether o_i in sample s_i and o_j in sample s_j are bound to different operators.

Again (cf. (15)), we ensure that each operation is assigned to at least one time step by adding the clauses

$$\bigvee_{\tau=t_i^\perp}^{t^\top} t_{i,\tau} \qquad \forall i \ : \ o_i \in O \tag{33}$$

for all operations. No change to the original SAT formulation regarding the time step assignment is necessary to support rational IIs. For the assignment of each operation in each data sample to at least one operator, we add the clauses

$$\bigvee_{\beta=0}^{\mathrm{FUs}(\omega)-1} b_{i,\beta}^{(s)} \qquad \forall i, s \ : \ o_i \in \check{O}_\omega, \omega \in \check{\Omega}, 0 \leq s < \mathrm{S}. \tag{34}$$

Since the dependency constraint for uniform modulo scheduling with rational IIs is different from the one for integer IIs, we must add the clause

$$\neg t_{i,\tau_i} \vee \neg t_{j,\tau_j} \quad \forall \ i, j, \tau_i, \tau_j \ : (o_i \rightarrow o_j) \in E, t_i^\perp \leq \tau_i \leq t_i^\top, t_j^\perp \leq \tau_j, \tau_j + \Delta_{\min}(d_{i,j}) < \tau_i + \delta_{i,j} \tag{35}$$

for each pair of execution times that violates (28).

Resource conflicts arise if two operations are scheduled in equal congruence classes and are bound to the same operator instance. We prohibit this by adding the clause

$$T_{i,j}^{(s_i,s_j)} \vee B_{i,j}^{(s_i,s_j)} \qquad \forall i, j \ : \ o_i \in \check{O}_\omega, o_j \in \check{O}_\omega, i > j, \omega \in \check{\Omega}, 0 \leq s_i < \mathrm{S}, 0 \leq s_j < \mathrm{S}. \tag{36}$$

Again, note the similarity to the corresponding clause for integer IIs (i.e., (19)). The only difference is that we must prohibit resource conflicts for all combinations of data samples, here, denoted as s_i and s_j.

Finally, we must guarantee that $T_{i,j}^{(s_i,s_j)}$ and $B_{i,j}^{(s_i,s_j)}$ are correctly set to zero when two operations are scheduled into the same congruence class modulo M or are bound to the same operator. We do so by adding the clauses

$$\neg T_{i,j}^{(s_i,s_j)} \vee \neg t_{i,\tau_i} \vee \neg t_{j,\tau_j} \qquad \forall i,j,\tau_i,\tau_j,s_i,s_j \ :$$

$$o_i \in \check{O}_\omega, o_j \in \check{O}_\omega, i > j, \omega \in \check{\Omega}, t_i^\perp \leq \tau_i \leq t_i^\top, t_j^\perp \leq \tau_j \leq t_j^\top,$$
$$0 \leq s_i < \mathrm{S}, 0 \leq s_j < \mathrm{S}, (\tau_i + I^{(s_i)}) \bmod \mathrm{M} = (\tau_j + I^{(s_j)}) \bmod \mathrm{M} \tag{37}$$

and

$$\neg B_{i,j}^{(s_i,s_j)} \vee \neg b_{i,\beta}^{(s_i)} \vee \neg b_{j,\beta}^{(s_j)} \qquad \forall \omega,i,j,\beta,s_i,s_j \ :$$

$$\omega \in \check{\Omega}, o_i \in \check{O}_\omega, o_j \in \check{O}_\omega, i > j, 0 \leq \beta < \mathrm{FUs}(\omega), 0 \leq s_i < \mathrm{S}, 0 \leq s_j < \mathrm{S}. \tag{38}$$

The clause (38) follows from its integer-II counterpart (21) by adding the dependencies on the data samples. For (37) we must make sure to correctly determine whether two operations are scheduled into the same congruence class. Solving (23) for the schedule time of o_i in data sample s yields

$$t_i^{(s)} = t_i^{(0)} + I^{(s)} \qquad \forall s : 0 \leq s < \mathrm{S}. \tag{39}$$

Therefore, we can compute the congruence class as

$$(t_i^{(0)} + I^{(s)}) \bmod \mathrm{M} \tag{40}$$

and use this relation in (37).

3.2 Extending SCC-based modulo scheduling to rational IIs

The procedure outlined in Section 2.4 for a fast initial schedule computation can be easily extended to support rational IIs. In the ILP formulation for maximum schedule length computation (cf. Fig. 2) and for earliest and latest start time computations of an SCC (cf. Fig. 3) we need to account for data precedence for rational IIs as defined in (28) by changing **A** to (28) in Fig. 2 and 3.

Then, we can use the previously discussed SAT formulation for uniform scheduling with a rational II (see Section 3.1) to compute a relative schedule for complex SCCs (Line 15 in Algorithm 3).

When computing a schedule for a basic SCC (Line 21 in Algorithm 3), we must account for the rational II by unrolling the graph by a factor of S and scheduling the unrolled graph with II = M. Note that this in general does not guarantee a uniform schedule anymore. Additionally, we must schedule each operation S times – once for each data sample – when assembling the final schedule (Line 25 in Algorithm 3). When fixing final schedule times of trivial SCCs, we also drop the constraint of uniformity in favor of a possibly shorter schedule length.

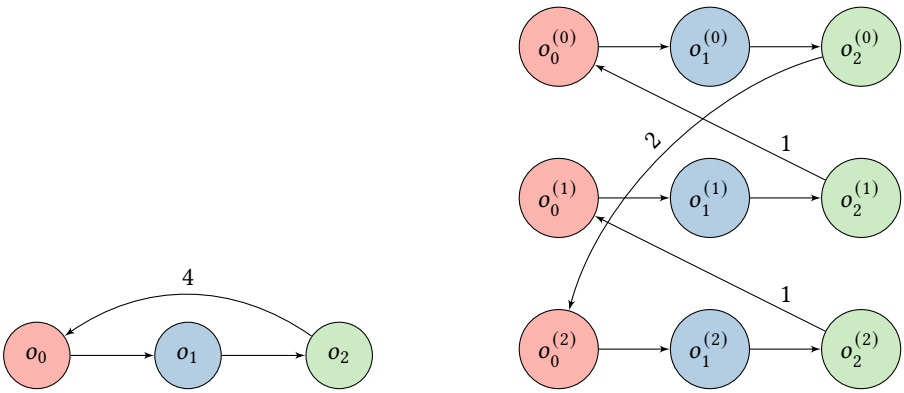
3.3 Nonuniform modulo scheduling for further schedule length optimization

Until now, the scheduling process for rational IIs was described as (i) using the SCC-based heuristic to quickly determine an initial schedule for the minimum possible II, and (ii) refining the schedule length under the assumption of unifomity for complexity reduction.

Once we found the optimum schedule length under the assumption of uniformity we can afterwards drop the uniformity constraint for even further schedule length reduction at the cost of a substantially higher complexity to model the scheduling problem. This is therefore only possible for sufficiently small problem sizes.

To compute a nonuniform modulo schedule for $\mathrm{II} = \frac{\mathrm{M}}{\mathrm{S}}$, we use the fact that unrolling the graph by a factor of S and afterwards scheduling the unrolled graph with II = M corresponds to a rational-II modulo schedule with $\mathrm{II} = \frac{\mathrm{M}}{\mathrm{S}}$ [32].

We illustrate graph unrolling by the example given in Fig. 5. It shows a DFG with three operations: o_0, o_1 and o_2, connected in a loop via two edges with distance zero and one edge with distance four (see Fig. 5(a)). Unrolling that graph by a factor of S = 3 yields a new graph with 9 operations (see Fig. 5(b)). Here, operations are named $o_i^{(s)}$, meaning that they represent o_i in data sample s.



(a) Example DFG from Section 1.2 (b) Unrolled DFG with S = 3

Fig. 5. Example graph (a) unrolled 3 times resulting in 9 vertices (b)

Whereas vertices and edges with distance zero are just duplicated S times, an unrolled edge's source sample index $\hat{s}$ and its distance $\hat{d}_{i,j}$ are calculated via

$$\hat{s} = (s - d_{i,j}) \bmod \mathrm{S} \tag{41}$$

$$\hat{d}_{i,j} = \max\left(0, \left\lceil \frac{d_{i,j} - s}{\mathrm{S}} \right\rceil\right). \tag{42}$$

In the following, we assume that the difference between the optimum schedule length for a nonuniform schedule and the optimum schedule length for a uniform schedule is typically very small. Suppose that the schedule length minimization procedure outlined in Section 2.2 using the modified SAT formulation from Section 3.1 for uniform modulo scheduling with a rational II yields an optimal schedule length L. We can then perform a reversed linear search for the optimal schedule length on the unrolled graph starting at $L - 1$ and decreasing that limit each time we encounter a valid nonuniform modulo schedule. Then, the last valid schedule before encountering a nonsatisfiable schedule length is the optimum one for the given II, and we can stop the search.

4 RELATED WORK

Even though there are several approaches for optimally solving the MSP for given a candidate II (e.g., based on a combination of SAT and SDC [14], based on enumeration [2], or based on constraint programming [4]), ILP can be considered state-of-the-art with multiple formulations available in literature [19, 39, 57].

Eichenberger and Davidson [19] satisfy resource constraints by decomposing the time slots of resource-constrained operations into their congruence class modulo II and an offset which is a

multiple of the candidate II using II-many binary variables per operation. They additionally re-write their formulation to only use 0-1-structured constraints, which reduces algorithm runtime during the ILP solver's branch & bound procedure. In the following we call their formulation ED97.

The strategy in the Moovac ILP formulation by Oppermann et al. [40] is to use overlap variables for each pair of resource-constrained operations of the same operator type and simultaneously compute a binding to force all potentially conflicting operations to be executed in different congruence classes or by different operators.

The formulation by Šůcha and Hanzálek [57] relies on binary variables to count the number of operations scheduled in equal congruence classes to determine the number of necessary operators for the given schedule during the solving process. They use this concept to also support a variable operator allocation if required by the user.

Although there do exist non-iterative optimal scheduling algorithms (e.g., a non-iterative version of Moovac by Oppermann et al. [41]), iterative algorithms have established themselves as state-of-the-art. A major disadvantage of non-iterative scheduling algorithms is the lack of fallback strategies in case of a solver timeout. Then, the user is left without any solution, which is inevitable for ILP-based algorithms and large problem sizes due to the NP-complexity of resource-constrained scheduling [23].

SAT-based techniques have been successfully applied to scheduling problems in other research domains such as employee timetabling [1], resource-constrained [29] or multi-mode project scheduling [51] or task scheduling for big data platforms [28]. Yamada et al. [54] formulated the (non-modulo) scheduling problem as a 0-1 integer programming problem and used a branch-and-bound method to support cardinality constraints. Unfortunately, other SAT-based scheduling techniques cannot simply be used to solve the MSP, because they must adhere to different constraints. For example in the employee timetabling problem [1], the schedule must account for employees that may have days off (which is not true for operators in the MSP); or in multi-mode project scheduling [51], the activities to be scheduled can be executed in different modes which lead to varying resource demands and latencies for that given activity.

A recent study of modulo scheduling algorithms [39] showed that Moovac [41] and the algorithm by Eichenberger and Davidson [19] are the state-of-the-art for latency-optimal iterative modulo scheduling. A widely used heuristic algorithm is the Modulo-SDC algorithm by Canis et al. [5]. It is, e.g., used in the HLS tool LegUp [6]. Therefore, in our following experiments regarding integer IIs, we compare our proposed scheduler to the three aforementioned ones.

Recently, an ILP formulation for optimal uniform modulo scheduling with rational IIs was published, which outperformed all previous exact algorithms in terms of number of solutions and average quality of results [22]. Aside from that, we also compare our proposed approach to the exact nonuniform and heuristic uniform scheduling algorithms proposed in [48] because they also outperformed the state-of-the-art regarding the number of valid solutions.

All evaluated schedulers use an iterative search for the optimum II. For integer IIs, we define $\mathrm{II}^{\top}$ as the schedule length of a non-modulo scheduler that heuristically satisfies resource constraints. In our case, we use the resource-constrained ASAP scheduler from the HatScheT library. For rational IIs, we use the iteration algorithm proposed in [48] without a limitation on the number of data samples S. It enumerates all irreducible fractions $\frac{\mathrm{M}}{\mathrm{S}}$ between the minimum rational II, $\mathrm{II}^{\perp}_{\mathbb{Q}} = \frac{\mathrm{M}^{\perp}}{\mathrm{S}^{\perp}}$, and the minimum integer II and also limits M and S to $\mathrm{M} \leq \mathrm{M}^{\perp}$ and $\mathrm{S} \leq \mathrm{S}^{\perp}$ to minimize the complexity of the resulting ILP/SAT program.

Satisfiability Modulo Theories (SMT) is an extension of SAT to more complex expressions (e.g., linear integer arithmetic). SMT is used, amongst others, in HLS for proving the absence of memory dependence violations, for both static [8] and dynamic scheduling [9]. These approaches differ from

ours in that they solely model memory dependencies in cases where the dependence distances are unknown, while we assume that all dependencies are known prior to scheduling and we incorporate resource constraints.

4.1 SAT in modulo scheduling

The most similar modulo scheduling approach to ours is the one by Dai & Zhang [14], in the following called DZ19, which is an extension of previous work that targets non-pipelined loops [13]. DZ19 uses a combination of SAT and SDC solvers and a graph reduction technique based on SCCs to exactly solve the MSP for the optimal II.

The main idea is to handle dependency constraints by an SDC solver with polynomial time complexity and let a SAT solver take care of resource constraints. In each iteration (1) the SAT solver makes a proposal for an operation ordering that would satisfy resource constraints, (2) these orderings are added as additional constraints to the SDC, and (3) the SDC solver checks whether these constraints lead to a valid modulo schedule. If this is not the case, (4) the SDC solver identifies conflicts in the modulo ordering proposed by the SAT solver. Those conflicts are (5) then prohibited by adding an additional clause to the SAT solver and these five steps repeat until either a valid modulo schedule was found or the whole modulo ordering search space is exhausted, in which case the II is determined to be infeasible and this procedure executes again for the next candidate II.

The first obvious difference to our approach is that Dai & Zhang only formulate the resource constraints as a SAT problem instead of the whole MSP. The solver satisfies resource constraints by simultaneously computing a binding similar to our approach. Whenever two operations are bound to the same operator, Dai & Zhang prohibit resource conflicts by scheduling these operations in different congruence classes modulo II based on a modulo ordering variable $O_{i \to j,k}$. Since only the time difference modulo II must be different from zero, the relative schedule of each pair of conflicting operations is subject to an offset, which is a multiple of the II:

$$O_{i \to j,k} = \text{true implies } (k-1) \cdot \text{II} < t_i - t_j < k \cdot \text{II}, \tag{43}$$

for any $k \in \mathbb{N}$. The additional constraints resulting from the assignment of $O_{i \to j,k}$ and (43) are passed to the SDC solver, which then checks feasibility and reports a minimum set of $O_{i \to j,k}$ variables that cannot lead to a valid modulo schedule.

This approach has the advantage that only a portion of the MSP has to be modeled in SAT, at the cost of multiple solving attempts per candidate II. Another notable difference to our proposed approach is that the schedule length is not an optimization criterion for the scheduler and only results from the first modulo ordering proposed by the SAT solver that leads to a valid schedule, even if there could be another ordering that would lead to a lower schedule length.

4.2 Complexity of state-of-the-art exact approaches

Table 3 summarizes the ILP/SAT formulation size scaling of exact state-of-the-art approaches and the number of solving attempts per candidate II. The ILP-based approaches have the advantage that the ILP solver must only be called once to compute the optimal modulo schedule for a candidate II or to decide that the candidate II is infeasible. Both the ED97 and the Moovac formulation sizes scale quadratically w.r.t. the problem size.[7] We can also see that the Moovac formulation is more appropriate for scheduling problems with few limited operations, since the number of overlap variables and constraints scales quadratically with the amount of resource-limited operations.

In the DZ19 scheduling algorithm, the number of times the SAT solver must be called depends on the number of times the SDC solver rejects a modulo ordering proposal, which we denote as

[7]The problem size is defined via the input parameters to the MSP, as shown in Table 1.

O(#resource conflicts). We also see that the number of variables grows quadratically for large candidate IIs (scheduling problems with few parallelization opportunities) and cubically when the II approaches one (completely parallel scheduling problems).

Our proposed approach, however, only requires a single solving attempt per candidate II until the optimum II is found and then needs $O(\sqrt{\mathrm{L}^{\top} - \mathrm{L}^{\perp}})$ SAT calls for SL optimization. The number of variables grows quadratically, independent of the II and the number of constraints grows quartically with our proposed clause reduction. If the clause reduction is not applied, we still need a quadratic number of variables but the number of clauses grows with the fifth power.

Table 3. Variable/Constraint count for exact modulo scheduling algorithms; we give references to constraints as equation numbers in the original papers; only the variables/constraints with the highest order scaling w.r.t. the input parameters are given since they are assumed to be the bottleneck when scaling to large problem sizes

Variable/constraint	Amount	Explanation
ED97 [19]: $O(1)$ solving attempts per candidate II		
Variable $a_i \in \{0,1\}$	$O(\lvert O\rvert \cdot \mathrm{II})$	Modulo slot variables
Constraint (20)	$O(\lvert E\rvert \cdot \mathrm{II})$	0-1-structured dependency constraints
Moovac [40]: $O(1)$ solving attempts per candidate II		
Variables $\epsilon_{i,j}, \mu_{i,j} \in \{0,1\}$	$O(\lvert\check{O}\rvert^2)$	Time/binding overlap variables
Constraints (6)–(12)	$O(\lvert\check{O}\rvert^2)$	Time/binding overlap constraints
DZ19 [14]: O(#Resource conflicts) solving attempts per candidate II		
Variable $O_{i\to j,k} \in \{0,1\}$	$O(\lvert\check{O}\rvert^2 \cdot \mathrm{L}^{\top}/\mathrm{II})$	Modulo ordering variables
Constraint (7)	$O(\lvert\check{O}\rvert^2 \cdot \mathrm{L}^{\top}/\mathrm{II})$	Modulo ordering clauses
BLOOP: $O(1)$ solving attempts until the first feasible II is found; then: $O(\sqrt{\mathrm{L}^{\top} - \mathrm{L}^{\perp}})$ for SL opt.		
Variable $t_{i,\tau} \in \{0,1\}$	$O(\lvert O\rvert \cdot \mathrm{L}^{\top})$	Schedule time variables
Variables $T_{i,j}, B_{i,j} \in \{0,1\}$	$O(\lvert\check{O}\rvert^2)$	Time/binding difference variables (*with* proposed clause (19))
Constraint (20)	$O(\lvert\check{O}\rvert^2 \cdot \mathrm{L}^{\top 2})$	Time slot difference clauses (*with* proposed clause (19))
Constraint (18)	$O(\lvert\check{O}\rvert^2 \cdot \mathrm{L}^{\top 2} \cdot FUs)$	Resource constraint clauses (*without* proposed clause (19))

5 EXPERIMENTAL RESULTS

We implemented our scheduler in C++ within the open source HatScheT scheduling library [49] and also use this library's implementations of previous work [5, 19, 22, 40, 48], except for DZ19 [14] where we use the original implementation.[8] We use Gurobi 8.1 [26], accessed via the ScaLP [50] (I)LP library, to solve all ILP and SDC problems in HatScheT. For BLOOP's SAT instances we

[8]The DZ19 implementation rejects scheduling an MSP without complex SCCs, since such a scheduling problem can be scheduled for the optimal II in polynomial time using an SDC solver for basic SCCs and scheduling trivial SCCs ASAP with respect to the MRT. This process corresponds to BLOOP (cf. Fig. 4) without SL optimization. This means that the SL achieved depends on the order in which resource-limited trivial SCCs are scheduled. We therefore report numbers for BLOOP with deactivated SL minimization in place of DZ19 whenever DZ19 encounters a problem instance without complex SCC.

use CaDiCaL [3]. None of our used ILP or SAT solvers utilizes a limit on the number of allowed iterations. Exact approaches are only limited by a timeout, whereas for the Modulo SDC algorithm we limit the number of iterations to $6 \cdot |O|$ as suggested by Canis et al. [5].

As benchmark designs we use C-programs from CHStone [27] and MachSuite [47] available within the HatScheT library. Since we focus on Modulo Scheduling within this paper, we only evaluate the schedulers on the innermost loops contained in the benchmark programs. Furthermore, we use the Origami benchmark suite [38] which consists of several Matlab/Simulink models from digital signal processing.

Resource allocations for the CHStone and MachSuite benchmarks are summarized in Table 4. Many operators are modeled as unlimited, and often even without pipeline stages. Therefore, many additional chaining edges are added to prevent long combinatorial paths. Using the operator

Table 4. Operator limits for CHStone and MachSuite benchmarks

Operator type	Limit	Latency
mem read/write	2/1	2/1
32-bit +/−/∗/÷	∞/∞/∞/4	0/0/2/32
64-bit +/−/∗/÷	∞/∞/∞/4	0/0/5/65
relation > /= /<	∞/∞/∞	0/0/0
bit/logic	∞/∞	0/0

constraints from Table 4 for CHStone and MachSuite results in exactly one loop with a rational minimum II. We therefore only evaluate CHStone and MachSuite on integer IIs. For the Origami benchmark suite we instead allocate the minimum number of operators for all Pareto-optimal IIs [42] using

$$\mathrm{FUs}(\omega) = \left\lceil \frac{|\check{O}_\omega|}{\mathrm{II}} \right\rceil. \tag{44}$$

Different scheduling problems for each program in CHStone and MachSuite result from the different loops (i.e., different graphs while using the same resource model, column "#loops" in the following tables). For the models in Origami they result from different resource allocations for the same model (i.e., different resource models and the same graph, column "#allocs" in the following tables). This leads to two very different experimental settings: CHStone and MachSuite are dominated by chaining edges and unlimited operations—often even without pipelining—and Origami mainly consists of limited, pipelined operations and therefore does not contain any chaining edges.

When analyzing results for the experiments we focus on the following five metrics:

(1) the number of successfully computed schedules (in the following tables denoted as solved),
(2) the number of schedules that are proven by the respective scheduler to satisfy the optimum achievable II (denoted as $\mathrm{II} = \mathrm{II}^*_\square$),[9]
(3) the number of schedules that satisfy the optimum achievable II (denoted as $\mathrm{II} = \mathrm{II}^*$); here, it suffices that optimality is proven by any of the examined schedulers,
(4) the number of schedules that satisfy the smallest known II (denoted as $\mathrm{II} = \mathrm{II}_{\mathrm{min}}$); in cases where none of the examined schedulers is able to prove optimality for a benchmark instance, it is unknown whether $\mathrm{II}_{\mathrm{min}}$ is optimal,

[9] An II is proven to be optimal if it is equal to $\mathrm{II}^\perp$ or if all IIs between $\mathrm{II}^\perp$ and the achieved II are proven to be infeasible.

(5) the schedule length quality which we define as the minimum known schedule length for that scheduling problem divided by the schedule length achieved by the respective scheduler (denoted as $\frac{L_{\text{min}}}{L}$).[10]

As an example, assume $\text{II}^{\perp} = 3$ for a given problem instance. Now consider that scheduler A proves that $\text{II} = 4$ is optimal by proving that no schedule exists for $\text{II} = 3$ due to resource and precedence constraints (for scheduler A: $\text{II} = \text{II}^{*}_{\square} = \text{II}^{*} = \text{II}_{\text{min}} = 4$). Now consider that scheduler B fails to prove that $\text{II} = 3$ is infeasible but it is able to find a schedule for $\text{II} = 4$ (for scheduler B: $\text{II} = \text{II}^{*} = \text{II}_{\text{min}} = 4$). Although scheduler B fails to prove optimality, it still finds a schedule for the optimal II.

We chose these metrics because of the following reasons: In Modulo Scheduling, the II minimization is the most important objective and it has a much higher priority than minimizing the schedule length. Therefore, for an exact algorithm, it is the highest priority to achieve the best II possible. When using our proposed algorithm for the evaluation of heuristics it is not only necessary to compute a schedule for the minimum possible II but also prove that this is the optimum II for the given scheduling problem.

For large problem instances with rational IIs, the iteration algorithm proposed in [48] finds several hundreds to thousands of candidate IIs. We therefore manually limit the maximum number of candidate IIs to ten. Otherwise, runtime would be unreasonably long for the ILP-based schedulers that frequently fail to find schedules for large scheduling problems.

For a fair comparison, we define each scheduler's secondary optimization objective as the minimization of the schedule length. In the formulation by Eichenberger and Davidson, we add an additional virtual operation o_v with $L_v = 0$ to the scheduling problem, connect each node without outgoing unweighted edges to it and minimize its start time as described in [11]. This is already done in the Moovac-S formulation introduced in [41], and also in the schedulers for rational IIs [22, 48], which are all implemented in HatScheT.

5.1 Comparison with Dai & Zhang's approach

As our first experiment (cf. Table 5), we compare our proposed approach against the SAT+SDC-based scheduling algorithm by Dai & Zhang [14]. DZ19 classifies the optimal II for some problem instances as infeasible,[11] even though BLOOP is able to compute a valid schedule, which is verified to be optimal w.r.t. II and SL. We therefore exclude these instances from our evaluation and only focus on the remaining ones.

While DZ19 fails to find any modulo schedule for one problem in CHStone, BLOOP is able to compute a valid schedule for all instances. Additionally, we see that BLOOP has a higher number of (verified) optimal schedules than DZ19 and that there is not a single problem instance for which a schedule by DZ19 leads to a higher throughput than BLOOP's. DZ19's unsolved problem is a loop in aes with 1374 vertices, 205 of which are resource-limited, and 2752 edges. Even with a timeout of one hour, DZ19 fails to find a modulo schedule for that problem instance. It is one of the few *hard* instances, for which it is unknown whether BLOOP's solution is optimal. DZ19's slightly worse performance might come from the fact that it must solve numerous SAT instances until it successfully computes a schedule without resource conflicts or proves the absence of such. Furthermore, as seen in Section 4.2, the number of variables in Dai & Zhang's SAT formulation scales slightly worse than ours, which might also contribute to a longer runtime.

Since Dai & Zhang's scheduler does not guarantee latency-optimality, BLOOP regularly finds schedules with a lower schedule length, which leads to a higher average latency quality for BLOOP.

[10] The best possible value for $\frac{L_{\text{min}}}{L}$ is therefore 1 if $L = L_{\text{min}}$, approaching 0 for larger achieved schedule lengths.
[11] five problems from CHStone-jpeg, 34 from CHStone-motion and one from Origami-fir6dlms

Table 5. Comparing BLOOP and DZ19 for a timeout of 600 s; times to schedule the whole benchmark suite are given as t = hours : minutes : seconds; We exclude scheduling problems from our evaluation for which DZ19 classifies the optimal II as infeasible (see column "reject opt."). in the column "#problems" we give the number of evaluated problem instances and the original number (including the excluded ones) is displayed in brackets

		BLOOP t = 22:23:38						DZ19 [14] t = 22:32:09					
Benchmark	#problems	solved	$II = II^*_\square$	$II = II^*$	$II = II_{min}$	avg. $\frac{L_{min}}{L}$	reject opt.	solved	$II = II^*_\square$	$II = II^*$	$II = II_{min}$	avg. $\frac{L_{min}}{L}$	reject opt.
CHStone	224 (263)	224	220	220	224	1.00	0	223	219	219	220	0.93	39
MachSuite	91 (91)	91	90	90	91	1.00	0	91	90	90	91	0.93	0
Origami	161 (162)	161	161	161	161	0.97	0	161	161	161	161	0.89	1
summary	476 (516)	476	471	471	476	0.99	0	475	470	470	472	0.91	40

In the following experiments it will be seen that the remaining schedulers from the current state-of-the-art either time out or find a solution for the optimal II in all problem instances. We therefore exclude DZ19 from subsequent analyses.

5.2 Scheduling experiments

For our first evaluation against the remaining state-of-the-art we focus on the standard case of integer IIs and set a timeout of 10 min for MachSuite and CHStone. We immediately note that MachSuite seems to comprise *easy* benchmark problems, as all examined schedulers are able to compute a valid schedule for each instance. Results only differ in the amount of (verified) optimal solutions, the throughput achieved and the schedule length. CHStone, on the other hand, consists of harder scheduling problems, which is reflected by the higher runtime for each scheduler and the fact that the ILP-based schedulers (ED97 and Moovac) fail to compute valid modulo schedules for some problem instances. We therefore decide to also run experiments with timeouts of 1 min for MachSuite and 1 hour for CHStone.

Tables 6–7 show experimental results for MachSuite with 1 min and 10 min timeouts, respectively. We see that, in both cases, our proposed scheduler computes the highest number of schedules with verified optimal throughput (column $II = II^*_\square$) and only fails to prove optimality for one out of the 91 problems. ED97 and Moovac also fail to prove optimality for that problem within the given time limits. As expected, the heuristic scheduler (SDC) has the lowest runtime among all tested schedulers, but this comes at the cost of the lowest average solution quality regarding throughput and schedule length.

Results for CHStone benchmarks are given in Tables 8–9. Also for this benchmark suite, our proposed scheduler is able to compute a valid modulo schedule for all problem instances. With the 10 min timeout, ED97 and Moovac fail to compute valid schedules for four and two scheduling problems, respectively. Using the timeout of 1 hour, ED97 computes a valid schedule for all problems and Moovac only fails to find a valid solution for one instance. For both timeout settings, no scheduler from the state-of-the-art is able to compute any schedule with higher throughput than our proposed approach or computes more verified optimal solutions regarding throughput. This holds true for both MachSuite and CHStone.

Tables 10–11 show results for the Origami benchmark suites for both integer and rational IIs. We see again that our proposed scheduler is able to compute valid schedules for all scheduling problems. Its solving rate and the number of optimal IIs is only matched by the SCC-based heuristic for

Table 6. Scheduler comparison for MachSuite using a timeout of 600 s; times to schedule the whole benchmark suite are given as t = hours : minutes : seconds

		BLOOP $t = 1{:}13{:}35$					ED97 [19] $t = 0{:}54{:}35$					Moovac [40] $t = 2{:}52{:}19$					SDC [5] $t = 0{:}14{:}38$				
program	#loops	solved	$\text{II} = \text{II}^*_\square$	$\text{II} = \text{II}^*$	$\text{II} = \text{II}_{\min}$	avg. $\frac{L_{\min}}{L}$	solved	$\text{II} = \text{II}^*_\square$	$\text{II} = \text{II}^*$	$\text{II} = \text{II}_{\min}$	avg. $\frac{L_{\min}}{L}$	solved	$\text{II} = \text{II}^*_\square$	$\text{II} = \text{II}^*$	$\text{II} = \text{II}_{\min}$	avg. $\frac{L_{\min}}{L}$	solved	$\text{II} = \text{II}^*_\square$	$\text{II} = \text{II}^*$	$\text{II} = \text{II}_{\min}$	avg. $\frac{L_{\min}}{L}$
aes2	16	16	15	15	16	1.00	16	15	15	16	1.00	16	14	15	15	1.00	16	8	9	10	0.99
bfs_queue	2	2	2	2	2	1.00	2	2	2	2	1.00	2	2	2	2	1.00	2	2	2	2	1.00
fft_strided	2	2	2	2	2	1.00	2	2	2	2	1.00	2	2	2	2	1.00	2	1	2	2	1.00
gemm_blocked	5	5	5	5	5	1.00	5	5	5	5	1.00	5	5	5	5	1.00	5	5	5	5	1.00
gemm_ncubed	3	3	3	3	3	1.00	3	3	3	3	1.00	3	3	3	3	1.00	3	3	3	3	1.00
kmp	4	4	4	4	4	1.00	4	4	4	4	1.00	4	4	4	4	1.00	4	4	4	4	1.00
md_grid	10	10	10	10	10	1.00	10	9	10	10	1.00	10	9	10	10	1.00	10	7	10	10	0.99
md_knn	2	2	2	2	2	1.00	2	2	2	2	1.00	2	2	2	2	1.00	2	2	2	2	1.00
sort_merge	8	8	8	8	8	1.00	8	8	8	8	1.00	8	8	8	8	1.00	8	8	8	8	1.00
sort_radix	15	15	15	15	15	1.00	15	15	15	15	1.00	15	15	15	15	1.00	15	15	15	15	1.00
spmv_crs	2	2	2	2	2	1.00	2	2	2	2	1.00	2	2	2	2	1.00	2	2	2	2	1.00
spmv_ellpack	2	2	2	2	2	1.00	2	2	2	2	1.00	2	2	2	2	1.00	2	2	2	2	1.00
stencil2d	4	4	4	4	4	1.00	4	4	4	4	1.00	4	4	4	4	1.00	4	4	4	4	1.00
stencil3d	9	9	9	9	9	1.00	9	9	9	9	1.00	9	9	9	9	1.00	9	8	9	9	0.99
viterbi	7	7	7	7	7	1.00	7	7	7	7	1.00	7	7	7	7	1.00	7	7	7	7	1.00
summary	91	91	90	90	91	1.00	91	89	90	91	1.00	91	88	90	90	1.00	91	78	84	85	1.00

Table 7. Scheduler comparison for MachSuite using a timeout of 60 s; times to schedule the whole benchmark suite are given as t = hours : minutes : seconds

		BLOOP $t = 0{:}10{:}26$					ED97 [19] $t = 0{:}13{:}51$					Moovac [40] $t = 0{:}19{:}56$					SDC [5] $t = 0{:}07{:}29$				
program	#loops	solved	$\text{II} = \text{II}^*_\square$	$\text{II} = \text{II}^*$	$\text{II} = \text{II}_{\min}$	avg. $\frac{L_{\min}}{L}$	solved	$\text{II} = \text{II}^*_\square$	$\text{II} = \text{II}^*$	$\text{II} = \text{II}_{\min}$	avg. $\frac{L_{\min}}{L}$	solved	$\text{II} = \text{II}^*_\square$	$\text{II} = \text{II}^*$	$\text{II} = \text{II}_{\min}$	avg. $\frac{L_{\min}}{L}$	solved	$\text{II} = \text{II}^*_\square$	$\text{II} = \text{II}^*$	$\text{II} = \text{II}_{\min}$	avg. $\frac{L_{\min}}{L}$
aes2	16	16	15	15	16	1.00	16	15	15	16	1.00	16	14	15	15	1.00	16	8	9	10	0.99
bfs_queue	2	2	2	2	2	1.00	2	2	2	2	1.00	2	2	2	2	1.00	2	2	2	2	1.00
fft_strided	2	2	2	2	2	1.00	2	2	2	2	1.00	2	2	2	2	1.00	2	1	2	2	1.00
gemm_blocked	5	5	5	5	5	1.00	5	5	5	5	1.00	5	5	5	5	1.00	5	5	5	5	1.00
gemm_ncubed	3	3	3	3	3	1.00	3	3	3	3	1.00	3	3	3	3	1.00	3	3	3	3	1.00
kmp	4	4	4	4	4	1.00	4	4	4	4	1.00	4	4	4	4	1.00	4	4	4	4	1.00
md_grid	10	10	10	10	10	1.00	10	9	10	10	1.00	10	9	10	10	1.00	10	7	10	10	0.99
md_knn	2	2	2	2	2	1.00	2	2	2	2	1.00	2	2	2	2	1.00	2	2	2	2	1.00
sort_merge	8	8	8	8	8	1.00	8	8	8	8	1.00	8	8	8	8	1.00	8	8	8	8	1.00
sort_radix	15	15	15	15	15	1.00	15	15	15	15	1.00	15	15	15	15	1.00	15	15	15	15	1.00
spmv_crs	2	2	2	2	2	1.00	2	2	2	2	1.00	2	2	2	2	1.00	2	2	2	2	1.00
spmv_ellpack	2	2	2	2	2	1.00	2	2	2	2	1.00	2	2	2	2	1.00	2	2	2	2	1.00
stencil2d	4	4	4	4	4	1.00	4	4	4	4	1.00	4	4	4	4	1.00	4	4	4	4	1.00
stencil3d	9	9	9	9	9	1.00	9	9	9	9	1.00	9	9	9	9	1.00	9	8	9	9	0.99
viterbi	7	7	7	7	7	1.00	7	7	7	7	1.00	7	7	7	7	1.00	7	7	7	7	1.00
summary	91	91	90	90	91	1.00	91	89	90	91	1.00	91	88	90	90	1.00	91	78	84	85	1.00

rational IIs. This is to be expected because only one of the benchmarks has a recurrence. Therefore, the heuristic only needs to solve an ILP formulation for that one model.

Moovac and ED97 regularly time out for large problem instances (i.e., mat_inv, r2_FFT and r22_FFT) which leads to a high number of solving attempts before a valid schedule is found. Hence, solving time is higher and the number of schedules with optimal throughput is lower than for our proposed approach. Even though ED97 has a slightly better average schedule length quality

Table 8. Scheduler comparison for CHStone using a timeout of 600 s; times to schedule the whole benchmark suite are given as t = hours : minutes : seconds

		BLOOP t = 5:54:05					ED97 [19] t = 53:18:53					Moovac [40] t = 27:33:04					SDC [5] t = 12:21:02				
program	#loops	solved	$II = II^*_{\square}$	$II = II^*$	$II = II_{min}$	avg. $\frac{L_{min}}{L}$	solved	$II = II^*_{\square}$	$II = II^*$	$II = II_{min}$	avg. $\frac{L_{min}}{L}$	solved	$II = II^*_{\square}$	$II = II^*$	$II = II_{min}$	avg. $\frac{L_{min}}{L}$	solved	$II = II^*_{\square}$	$II = II^*$	$II = II_{min}$	avg. $\frac{L_{min}}{L}$
adpcm	30	30	29	29	30	1.00	30	29	29	30	1.00	30	29	29	29	1.00	30	28	28	29	0.99
aes	22	22	20	20	22	0.97	20	20	20	20	1.00	20	20	20	20	1.00	22	18	20	20	1.00
blowfish	1	1	1	1	1	0.99	0	0	0	0	–	1	0	0	0	1.00	1	0	1	1	0.90
dfdiv	2	2	2	2	2	1.00	2	2	2	2	1.00	2	2	2	2	1.00	2	2	2	2	1.00
dfsin	3	3	3	3	3	1.00	2	2	2	2	1.00	3	3	3	3	1.00	3	2	2	2	1.00
gsm	15	15	15	15	15	1.00	15	15	15	15	1.00	15	15	15	15	1.00	15	15	15	15	1.00
jpeg	113	113	113	113	113	1.00	113	113	113	113	1.00	113	113	113	113	1.00	113	111	113	113	0.98
mips	1	1	0	0	1	0.89	1	0	0	0	0.89	1	0	0	0	1.00	1	0	0	0	0.94
motion	51	51	51	51	51	1.00	51	51	51	51	1.00	51	51	51	51	1.00	51	34	51	51	1.00
sha	25	25	25	25	25	1.00	25	25	25	25	1.00	25	25	25	25	1.00	25	21	25	25	1.00
summary	263	263	259	259	263	1.00	259	257	257	258	1.00	261	258	258	258	1.00	263	231	257	258	0.99

Table 9. Scheduler comparison for CHStone using a timeout of 3600 s; times to schedule the whole benchmark suite are given as t = hours : minutes : seconds

		BLOOP t = 31:10:14					ED97 [19] t = 52:25:03					Moovac [40] t = 146:18:43					SDC [5] t = 13:18:13				
program	#loops	solved	$II = II^*_{\square}$	$II = II^*$	$II = II_{min}$	avg. $\frac{L_{min}}{L}$	solved	$II = II^*_{\square}$	$II = II^*$	$II = II_{min}$	avg. $\frac{L_{min}}{L}$	solved	$II = II^*_{\square}$	$II = II^*$	$II = II_{min}$	avg. $\frac{L_{min}}{L}$	solved	$II = II^*_{\square}$	$II = II^*$	$II = II_{min}$	avg. $\frac{L_{min}}{L}$
adpcm	30	30	29	30	30	1.00	30	30	30	30	1.00	30	29	29	29	1.00	30	28	29	29	0.99
aes	22	22	20	20	22	0.97	22	20	20	21	1.00	21	20	20	20	0.98	22	18	20	20	0.98
blowfish	1	1	1	1	1	1.00	1	0	0	0	0.54	1	0	0	0	1.00	1	0	1	1	0.91
dfdiv	2	2	2	2	2	1.00	2	2	2	2	1.00	2	2	2	2	1.00	2	2	2	2	1.00
dfsin	3	3	3	3	3	1.00	3	3	3	3	1.00	3	3	3	3	1.00	3	2	2	2	1.00
gsm	15	15	15	15	15	1.00	15	15	15	15	1.00	15	15	15	15	1.00	15	15	15	15	1.00
jpeg	113	113	113	113	113	1.00	113	113	113	113	1.00	113	113	113	113	1.00	113	111	113	113	0.98
mips	1	1	0	0	1	0.89	1	0	0	0	0.89	1	0	0	0	1.00	1	0	0	0	0.94
motion	51	51	51	51	51	1.00	51	51	51	51	1.00	51	51	51	51	1.00	51	34	51	51	1.00
sha	25	25	25	25	25	1.00	25	25	25	25	1.00	25	25	25	25	1.00	25	21	25	25	1.00
summary	263	263	259	260	263	1.00	263	259	259	260	1.00	262	258	258	258	1.00	263	231	258	258	0.99

than our proposed scheduler, we argue that the greatly reduced runtime and the higher number of (proven) optimal IIs speak for our proposed algorithm.

Because of tight resource constraints, the exact ILP-based algorithms for rational IIs (uniform ILP and nonuniform ILP) regularly fail to find feasible solutions for mat_inv, r2_FFT and r22_FFT. Even when solutions are found, throughput-optimality can often not be guaranteed. It is only given when using our proposed approach or the heuristic one. The heuristic approach has a significantly lower runtime than all exact ones (by a factor of more than 20×) and our proposed scheduler has the lowest total runtime among the exact ones. Even though the uniform ILP-based scheduler has a better average schedule length quality than our proposed one for fir_SAM, iir_sos16 and mat_inv, our proposed scheduler has the best schedule length quality averaged over all models for rational IIs.

Fig. 6 shows a summary of relative run times for all our experiments. Absolute run times (i.e., the times that represent 100 %) are given in Tables 7–11. Note that absolute times partially vary

Table 10. Scheduler comparison for Origami using a timeout of 600 s; times to schedule the whole benchmark suite are given as t = hours : minutes : seconds

		BLOOP t = 15:16:22					ED97 [19] t = 182:45:02					Moovac [40] t = 335:41:32					SDC [5] t = 4:59:13				
model	#allocs	solved	$\Pi = \Pi^*_\square$	$\Pi = \Pi^*$	$\Pi = \Pi_{\min}$	avg. $\frac{L_{\min}}{L}$	solved	$\Pi = \Pi^*_\square$	$\Pi = \Pi^*$	$\Pi = \Pi_{\min}$	avg. $\frac{L_{\min}}{L}$	solved	$\Pi = \Pi^*_\square$	$\Pi = \Pi^*$	$\Pi = \Pi_{\min}$	avg. $\frac{L_{\min}}{L}$	solved	$\Pi = \Pi^*_\square$	$\Pi = \Pi^*$	$\Pi = \Pi_{\min}$	avg. $\frac{L_{\min}}{L}$
e_detect	6	6	6	6	6	1.00	6	6	6	6	1.00	6	6	6	6	1.00	6	6	6	6	0.95
fir6dlms	1	1	1	1	1	1.00	1	1	1	1	1.00	1	1	1	1	1.00	1	1	1	1	0.95
fir_gen	4	4	4	4	4	1.00	4	4	4	4	1.00	4	4	4	4	1.00	4	4	4	4	0.95
fir_GM	5	5	5	5	5	1.00	5	5	5	5	1.00	5	5	5	5	1.00	5	5	5	5	0.94
fir_hilb	3	3	3	3	3	1.00	3	3	3	3	1.00	3	3	3	3	1.00	3	3	3	3	1.00
fir_lms	1	1	1	1	1	1.00	1	1	1	1	1.00	1	1	1	1	1.00	1	0	1	1	1.00
fir_SAM	15	15	15	15	15	1.00	15	15	15	15	1.00	15	10	10	10	0.96	15	14	14	14	0.81
fir_SHI	7	7	7	7	7	1.00	7	7	7	7	1.00	7	7	7	7	1.00	7	7	7	7	0.89
fir_srg	3	3	3	3	3	1.00	3	3	3	3	1.00	3	3	3	3	1.00	3	3	3	3	1.00
iir4	1	1	1	1	1	1.00	1	1	1	1	1.00	1	1	1	1	1.00	1	1	1	1	1.00
iir_biqu	1	1	1	1	1	1.00	1	1	1	1	1.00	1	1	1	1	1.00	1	1	1	1	1.00
iir_bw	1	1	1	1	1	1.00	1	1	1	1	1.00	1	1	1	1	0.99	1	1	1	1	1.00
iir_sos2	1	1	1	1	1	1.00	1	1	1	1	1.00	1	1	1	1	1.00	1	0	0	0	0.97
iir_sos4	2	2	2	2	2	1.00	2	2	2	2	1.00	2	2	2	2	1.00	2	2	2	2	0.91
iir_sos8	3	3	3	3	3	1.00	3	3	3	3	1.00	3	3	3	3	1.00	3	3	3	3	0.88
iir_sos16	5	5	5	5	5	1.00	5	5	5	5	1.00	5	4	4	4	0.98	4	1	1	1	0.94
mat_inv	28	28	28	28	28	0.90	28	27	27	27	0.97	28	1	1	1	0.80	24	23	23	23	0.95
r2_FFT	32	32	32	32	32	0.91	27	15	15	15	0.93	30	1	1	1	0.34	31	31	31	31	0.91
r22_FFT	32	32	32	32	32	0.90	27	18	18	18	0.95	28	1	1	1	0.37	31	31	31	31	0.90
rgb_tr	3	3	3	3	3	1.00	3	3	3	3	1.00	3	3	3	3	1.00	3	3	3	3	1.00
splin_pf	5	5	5	5	5	1.00	5	5	5	5	1.00	5	5	5	5	1.00	5	5	5	5	1.00
ycbcr_tr	3	3	3	3	3	1.00	3	3	3	3	1.00	3	3	3	3	1.00	3	3	3	3	0.97
summary	162	162	162	162	162	0.95	152	130	130	130	0.97	156	67	67	67	0.72	155	148	149	149	0.92

Table 11. Scheduler comparison for Origami using a timeout of 600 s; times to schedule the whole benchmark suite are given as t = hours : minutes : seconds; some models are missing compared to Table 10 because they do not have operator allocations that lead to a rational minimum II

		BLOOP t = 202:29:18					uniform ILP [22] t = 218:40:20					nonuniform ILP [48] t = 231:26:47					SCC+ILP [48] t = 9:57:08				
model	#allocs	solved	$\Pi = \Pi^*_\square$	$\Pi = \Pi^*$	$\Pi = \Pi_{\min}$	avg. $\frac{L_{\min}}{L}$	solved	$\Pi = \Pi^*_\square$	$\Pi = \Pi^*$	$\Pi = \Pi_{\min}$	avg. $\frac{L_{\min}}{L}$	solved	$\Pi = \Pi^*_\square$	$\Pi = \Pi^*$	$\Pi = \Pi_{\min}$	avg. $\frac{L_{\min}}{L}$	solved	$\Pi = \Pi^*_\square$	$\Pi = \Pi^*$	$\Pi = \Pi_{\min}$	avg. $\frac{L_{\min}}{L}$
e_detect	7	7	7	7	7	0.94	7	7	7	7	0.93	7	7	7	7	1.00	7	7	7	7	0.84
fir_gen	4	4	4	4	4	1.00	4	4	4	4	0.93	4	4	4	4	1.00	4	4	4	4	0.70
fir_GM	3	3	3	3	3	1.00	3	3	3	3	0.97	3	3	3	3	1.00	3	3	3	3	0.83
fir_hilb	1	1	1	1	1	1.00	1	1	1	1	0.91	1	1	1	1	1.00	1	1	1	1	0.88
fir_SAM	79	79	79	79	79	0.59	79	79	79	79	0.97	79	75	75	75	0.56	79	79	79	79	0.54
fir_SHI	16	16	16	16	16	0.98	16	16	16	16	0.92	16	16	16	16	1.00	16	16	16	16	0.57
fir_srg	1	1	1	1	1	1.00	1	1	1	1	0.92	1	1	1	1	1.00	1	1	1	1	0.77
iir_sos16	1	1	1	1	1	0.22	1	1	1	1	0.93	1	1	1	1	1.00	1	1	1	1	0.18
mat_inv	230	230	230	230	230	0.67	230	204	204	204	0.72	156	58	58	58	0.69	230	230	230	230	0.40
r2_FFT	269	269	269	269	269	0.92	250	119	119	119	0.56	127	37	37	37	0.62	269	269	269	269	0.60
r22_FFT	269	269	269	269	269	0.92	249	128	128	128	0.55	131	37	37	37	0.59	269	269	269	269	0.59
rgb_tr	1	1	1	1	1	1.00	1	1	1	1	0.93	1	1	1	1	1.00	1	1	1	1	0.88
splin_pf	4	4	4	4	4	0.96	4	4	4	4	0.95	4	4	4	4	1.00	4	4	4	4	0.89
ycbcr_tr	1	1	1	1	1	1.00	1	1	1	1	0.96	1	1	1	1	1.00	1	1	1	1	0.92
summary	886	886	886	886	886	0.83	847	569	569	569	0.66	532	246	246	246	0.65	886	886	886	886	0.55

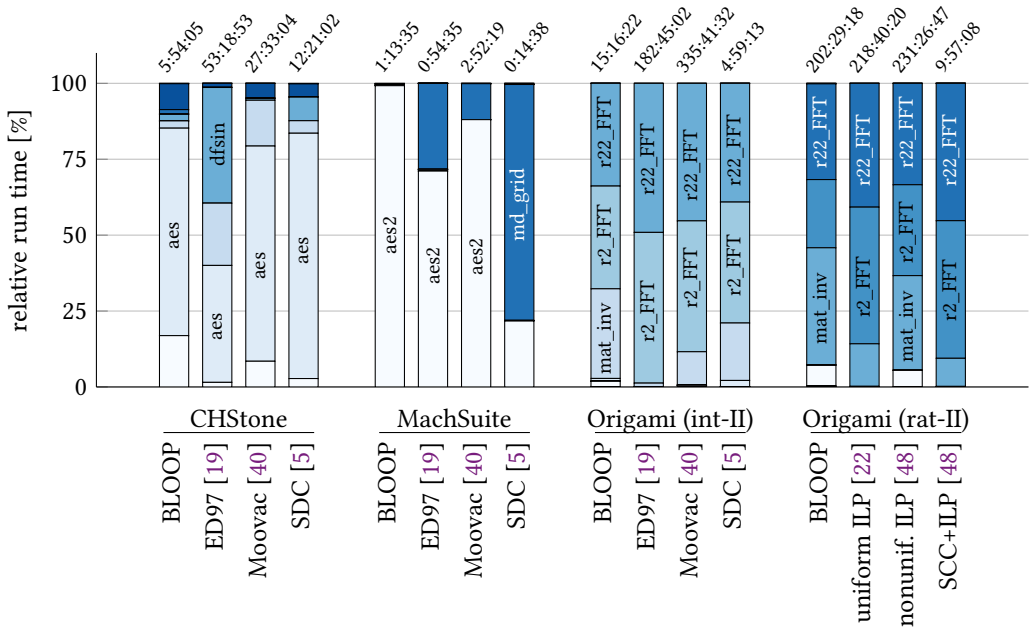


Fig. 6. Relative run times for a timeout of 600 s; we give absolute run times (i.e., the times that represent 100 %) as "hours : minutes : seconds" according to Tables 6–11

considerably. For all benchmark suites we see that there is always a small selection of benchmark problems that takes up a large portion of the whole run time (e.g., aes for CHStone). This situation is most extreme for MachSuite, where BLOOP spends nearly all its time scheduling the aes2 program.

For Origami—both integer IIs and rational IIs—mat_inv and the two FFTs make up nearly 100 % of the run time for all examined schedulers. This also explains the long run time for the exact schedulers in the rational-II experiments. Since mat_inv, r2_FFT and r22_FFT together consist of 768 scheduling problems, even an approach that is able to schedule each problem at the first try within the 10 min time budget (i.e., for $\mathrm{II}^{\perp}$) needs to solve 768 scheduling problems. This issue should in a practical application rather be solved in the allocation step, and *meaningful* resource-throughput tradeoffs should be chosen instead of enumerating all resource allocations that lead to Pareto-optimal implementations.

In our last evaluation, we take a look at the problem sizes, that our examined schedulers can handle with a timeout of 600 s. To do so, we display the largest solved problems and the largest problems with a provably optimal solution in Table 12. A loop from dfsin in CHStone with 2651 operations is the largest MSP in our benchmark suite, which BLOOP and Moovac solved for the verified optimal throughput. Although the SDC scheduler is able to compute a valid schedule for this problem instance, the II is not proven to be optimal and the largest problem with that property is a loop in jpeg with 942 operations for SDC. This is also the largest problem that the ED97 scheduler is able to solve with verified optimal throughput. The largest solved problem by ED97 is mips with 1076 operations.

The one problem in MachSuite for which none of the examined schedulers can produce a verified optimal schedule is a loop in aes2 with 225 vertices and 683 edges. It is an order of magnitude smaller than dfsin, which is provably optimally solved by both BLOOP and Moovac. For the minimum

known feasible II = 32 ED97 builds an ILP model with 7876 variables and 23081 constraints and is able to optimally solve it w.r.t. SL, whereas Moovac fails to find a feasible schedule for II = 32, even though its ILP model comprises half of the number of variables and constraints, 3876 and 11341, respectively. With 5796 variables and 334547 clauses after the SCC reduction, BLOOP is able to find a valid modulo schedule for II = 32 in under one second. From this behavior we can make two observations:

(1) The "difficulty" of a scheduling problem is not only related to its size.
(2) An ILP/SAT model with a high number of variables and constraints/clauses is not necessarily harder to solve than a smaller one.

Table 12. Largest (optimally) solved problems for a timeout of 600 s

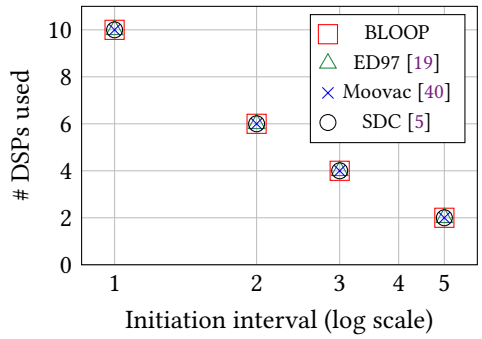
	Valid modulo schedule			For $\mathrm{II} = \mathrm{II}^{*}_{\square}$		
Scheduler	Benchmark	Instance	$\lvert O\rvert$	Benchmark	Instance	$\lvert O\rvert$
BLOOP	CHStone	dfsin	2651	CHStone	dfsin	2651
ED97 [19]	CHStone	mips	1076	CHStone	jpeg	942
Moovac [40]	CHStone	dfsin	2651	CHStone	dfsin	2651
SDC [5]	CHStone	dfsin	2651	CHStone	jpeg	942

5.3 FPGA implementations

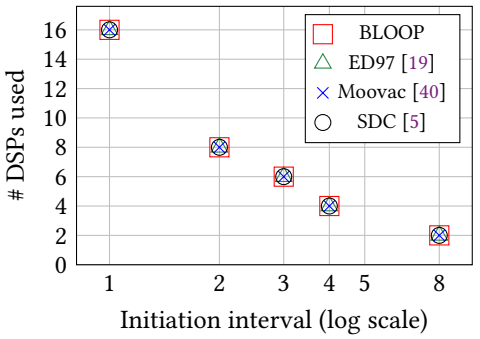
We implemented selected benchmarks on an `xcvu13p-fhga2104-2-e` FPGA using Vivado 2020.2 to show the impact of the chosen scheduler on the hardware. We use Origami HLS [38] with FloPoCo [16] backend for VHDL code generation. For binding we use an ILP-based algorithm for simultaneous lifetime register and multiplexer minimization [21] with a timeout of ten minutes. Although the binding algorithm is not always able to compute the optimum solution, we always obtain a feasible binding. We set a target frequency of 250 MHz, which is met by all implementations; independent of the scheduler choice and the resulting binding.

The number of resulting look-up tables and flip-flops are mainly determined by the HLS steps following the scheduling (i.e., binding, architecture synthesis, platform synthesis, place & route). Our examined schedulers can only control the tradeoff between DSP usage and resulting throughput, which is determined by the II and the clock frequency achieved.

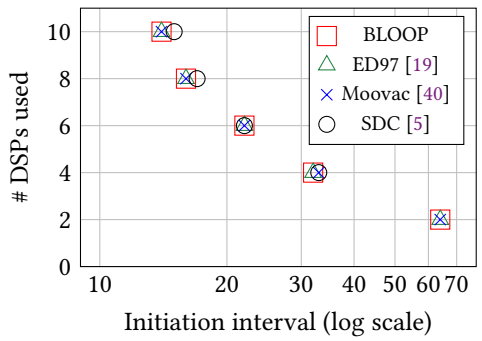
In Fig. 7 we show post place & route implementation results for selected benchmarks. The different implementations result from scheduler and resource allocation variations. With 15 and 26 operations, respectively, the fir_gen and splin_pf benchmarks are comparably small models that can be scheduled by each scheduler for the optimal II. Therefore, the Pareto frontier is independent of the chosen scheduling algorithm. Larger models like iir_sos16 or fir_SAM (194 and 121 operations, respectively) are not scheduled optimally by Moovac and SDC. The iir_sos16 model consists of more limited operations and several recurrences that result in $\mathrm{II}^{\perp}_{\mathrm{rec}} = 14$. The combination of these features seems problematic for Moovac and the SDC-based algorithm. Moovac failed to find a solution for the optimum throughput for one allocation; the SDC-based algorithm is able to guarantee an optimal II for only one out of the five allocations and even failed to find any solution for the implementation with two DSPs. The situation is even more extreme for mat_inv (266 operations). Even though it does not comprise any recurrence, the amount of resource-limited operations is so high that only our proposed scheduler is able to guarantee the optimum II for all resource allocations.



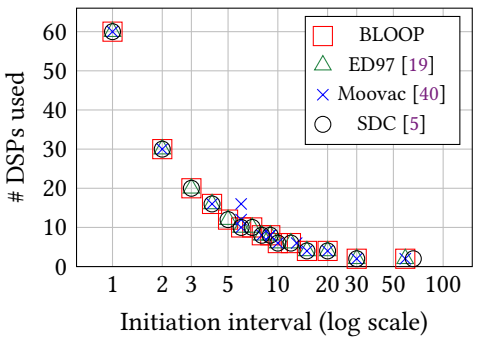
(a) DSP usage for the fir_gen benchmark



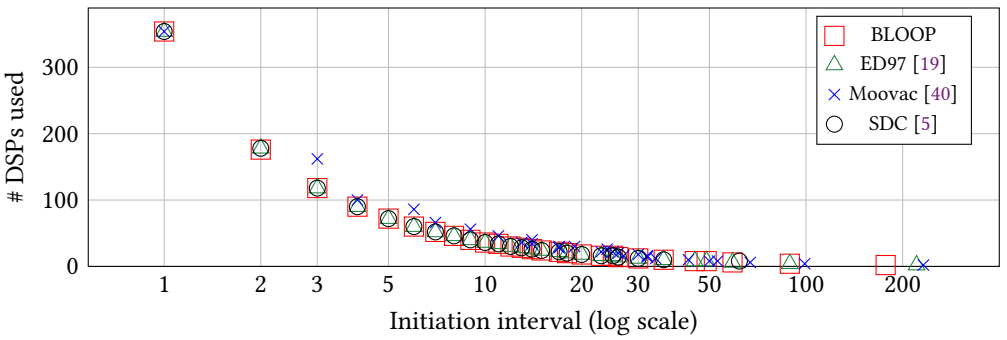
(b) DSP usage for the splin_pf benchmark



(c) DSP usage for the iir_sos16 benchmark



(d) DSP usage for the fir_SAM benchmark



(e) DSP usage for the mat_inv benchmark

Fig. 7. Throughput vs. DSP usage tradeoffs for selected benchmarks after place & route

6 CONCLUSION

In this paper we show that SAT-based optimal modulo scheduling with our proposed formulation and the open-source SAT solver CaDiCaL [3] scales better for large problem sizes than ILP-based approaches with the state-of-the-art commercial solver Gurobi [26].

Our proposed scheduler has the highest number of proven optimal solutions in both the MachSuite and the CHStone benchmark suites. Due to the higher solving rate, it needs substantially fewer

iterations to find a feasible schedule, which also leads to the lowest total runtime among the examined schedulers; including a state-of-the-art heuristic approach [5]. Additionally, none of the other examined schedulers [14, 19, 40] is able to find any schedule with higher throughput than our proposed approach.

We also showed that our proposed scheduler is the best available choice for design space explorations demonstrated on a wide range of DSP applications. It is the only exact approach that is able to compute provably throughput-optimal schedules for all models, resource allocations and both integer and rational IIs.

Open problems for future work remain (i) integrating FPGA-specific operator costs (e.g., the number of available DSPs or LUTs) into our problem formulation, (ii) finding better bounds for $L^{\perp}$ and $L^{\top}$ to further prune the search space, resulting in an even shorter runtime, and (iii) using the scalability of modern SAT solvers to optimally solve more optimization problems in digital hardware synthesis.

ACKNOWLEDGMENTS

We want to thank Steve Dai and Zhiru Zhang for providing us the source code of their SAT+SDC-based modulo scheduling algorithm and the anonymous reviewers for their valuable feedback.

REFERENCES

[1] Fadi Aloul, Bashar Al-Rawi, Anas Al-Farra, and Basel Al-Roh. 2006. Solving Employee Timetabling Problems Using Boolean Satisfiability. In *2006 Innovations in Information Technology*. 1–5. https://doi.org/10.1109/INNOVATIONS.2006.301886

[2] Erik R. Altman and Guang R. Gao. 1998. Optimal Modulo Scheduling Through Enumeration. *International Journal of Parallel Programming* 26, 3 (June 1998), 313–344. https://doi.org/10.1023/A:1018742213548

[3] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. 2020. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling Entering the SAT Competition 2020. In *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions (Department of Computer Science Report Series B, Vol. B-2020-1)*, Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda (Eds.). University of Helsinki, 51–53.

[4] Alessio Bonfietti, Michele Lombardi, Luca Benini, and Michela Milano. 2014. CROSS cyclic resource-constrained scheduling solver. *Artificial Intelligence* 206 (Jan. 2014), 25–52. https://doi.org/10.1016/j.artint.2013.09.006

[5] Andrew Canis, Stephen D. Brown, and Jason H. Anderson. 2014. Modulo SDC scheduling with recurrence minimization in high-level synthesis. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. 1–8. https://doi.org/10.1109/FPL.2014.6927490 ISSN: 1946-1488.

[6] Andrew Canis, Jongsok Choi, Blair Fort, Ruolong Lian, Qijing Huang, Nazanin Calagar, Marcel Gort, Jia Jun Qin, Mark Aldham, Tomasz Czajkowski, Stephen Brown, and Jason Anderson. 2013. From software to accelerators with LegUp high-level synthesis. In *2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. 1–9. https://doi.org/10.1109/CASES.2013.6662524

[7] Jianyi Cheng, Lana Josipovic, George A. Constantinides, Paolo Ienne, and John Wickerson. 2020. Combining Dynamic & Static Scheduling in High-level Synthesis. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, Seaside CA USA, 288–298. https://doi.org/10.1145/3373087.3375297

[8] Jianyi Cheng, John Wickerson, and George A. Constantinides. 2021. Exploiting the Correlation between Dependence Distance and Latency in Loop Pipelining for HLS. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*. 341–346. https://doi.org/10.1109/FPL53798.2021.00066 ISSN: 1946-1488.

[9] Jianyi Cheng, John Wickerson, and George A. Constantinides. 2022. Dynamic C-Slow Pipelining for HLS. In *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 1–10. https://doi.org/10.1109/FCCM53951.2022.9786096 ISSN: 2576-2621.

[10] Jason Cong, Jason Lau, Gai Liu, Stephen Neuendorffer, Peichen Pan, Kees Vissers, and Zhiru Zhang. 2022. FPGA HLS Today: Successes, Challenges, and Opportunities. *ACM Transactions on Reconfigurable Technology and Systems* (April 2022), 3530775. https://doi.org/10.1145/3530775

[11] Jason Cong and Zhiru Zhang. 2006. An Efficient and Versatile Scheduling Algorithm Based on SDC Formulation. In *43rd ACM/IEEE Design Automation Conference*.

[12] Stephen A. Cook. 1971. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing (STOC '71)*. Association for Computing Machinery, New York, NY, USA, 151–158.

https://doi.org/10.1145/800157.805047

[13] Steve Dai, Gai Liu, and Zhiru Zhang. 2018. A Scalable Approach to Exact Resource-Constrained Scheduling Based on a Joint SDC and SAT Formulation. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '18)*. Association for Computing Machinery, New York, NY, USA, 137–146. https://doi.org/10.1145/3174243.3174268

[14] Steve Dai and Zhiru Zhang. 2019. Improving Scalability of Exact Modulo Scheduling with Specialized Conflict-Driven Learning. In *Proceedings of the 56th Annual Design Automation Conference 2019*. ACM, Las Vegas NV USA, 1–6. https://doi.org/10.1145/3316781.3317842

[15] Benoıt Dupont de Dinechin. 2007. Time-Indexed Formulations and a Large Neighborhood Search for the Resource-Constrained Modulo Scheduling Problem. In *proceedings of the 3rd Multidisciplinary International Conference on Scheduling : Theory and Applications (MISTA 2007)*. Paris, France, 144–151.

[16] Florent de Dinechin and Bogdan Pasca. 2011. Designing Custom Arithmetic Data Paths with FloPoCo. *IEEE Design & Test of Computers* (2011).

[17] Giovanni De Micheli. 2003. *Synthesis and Optimization of Digital Circuits*. Tata McGraw-Hill, New Dehli.

[18] Leandro de Souza Rosa, Christos-Savvas Bouganis, and Vanderlei Bonato. 2019. Scaling Up Modulo Scheduling for High-Level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 5 (May 2019), 912–925. https://doi.org/10.1109/TCAD.2018.2834440

[19] Alexandre E. Eichenberger and Edward S. Davidson. 1997. Efficient formulation for optimal modulo schedulers. *ACM SIGPLAN Notices* 32, 5 (May 1997), 194–205. https://doi.org/10.1145/258916.258933

[20] Kevin Fan, Manjunath Kudlur, Hyunchul Park, and Scott Mahlke. 2005. Cost sensitive modulo scheduling in a loop accelerator synthesis system. In *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*. 12 pp.–232. https://doi.org/10.1109/MICRO.2005.17 ISSN: 2379-3155.

[21] Nicolai Fiege, Patrick Sittel, and Peter Zipf. 2022. Optimal Binding and Port Assignment for Loop Pipelining in High-Level Synthesis. In *2022 32st International Conference on Field-Programmable Logic and Applications (FPL)*. Accepted for publication.

[22] Nicolai Fiege, Patrick Sittel, and Peter Zipf. 2022. Speeding Up Optimal Modulo Scheduling with Rational Initiation Intervals. In *2022 32st International Conference on Field-Programmable Logic and Applications (FPL)*. Accepted for publication.

[23] Michael R. Garey and David S. Johnson. 1975. Complexity Results for Multiprocessor Scheduling under Resource Constraints. *SIAM J. Comput.* 4, 4 (1975), 397–411. https://doi.org/10.1137/0204035

[24] Sabih H. Gerez. 2005. *Algorithms for VLSI Design Automation*. Wiley, Chichester.

[25] Hongyan Guo, Feng Liu, Fang Xu, Hong Chen, Dongpu Cao, and Yan Ji. 2019. Nonlinear Model Predictive Lateral Stability Control of Active Chassis for Intelligent Vehicles and Its FPGA Implementation. *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 49, 1 (Jan. 2019), 2–13. https://doi.org/10.1109/TSMC.2017.2749337

[26] Gurobi. 2022. Gurobi Optimizer. https://www.gurobi.com/products/gurobi-optimizer/

[27] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, and Hiroaki Takada. 2009. Proposal and Quantitative Analysis of the CHStone Benchmark Program Suite for Practical C-based High-level Synthesis. *Journal of Information Processing* 17 (2009), 242–254. https://doi.org/10.2197/ipsjjip.17.242

[28] Huang Hong, Latifur Khan, Ayoade Gbadebo, Zhou Shaohua, and Wei Yong. 2018. A Complex Task Scheduling Scheme for Big Data Platforms Based on Boolean Satisfiability Problem. In *2018 IEEE International Conference on Information Reuse and Integration (IRI)*. 170–177. https://doi.org/10.1109/IRI.2018.00033

[29] Andrei Horbach. 2010. A Boolean satisfiability approach to the resource-constrained project scheduling problem. *Annals of Operations Research* 181, 1 (Dec. 2010), 89–107. https://doi.org/10.1007/s10479-010-0693-2

[30] Lana Josipović, Radhika Ghosal, and Paolo Ienne. 2018. Dynamically Scheduled High-level Synthesis. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '18)*. Association for Computing Machinery, New York, NY, USA, 127–136. https://doi.org/10.1145/3174243.3174264

[31] Marcin Kowalczyk, Dominika Przewlocka, and Tomasz Kryjak. 2018. Real-Time Implementation of Contextual Image Processing Operations for 4K Video Stream in Zynq UltraScale+ MPSoC. In *2018 Conference on Design and Architectures for Signal and Image Processing (DASIP)*. 37–42. https://doi.org/10.1109/DASIP.2018.8597105

[32] Daniel M. Lavery and Wen-mei W. Hwu. 1995. Unrolling-based optimizations for modulo scheduling. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*. 327–337. https://doi.org/10.1109/MICRO.1995.476842 ISSN: 1072-4451.

[33] Haoyan Liu, Atiyehsadat Panahi, David Andrews, and Alexander Nelson. 2020. An FPGA-Based Upper-Limb Rehabilitation Device for Gesture Recognition and Motion Evaluation Using Multi-Task Recurrent Neural Networks. In *2020 International Conference on Field-Programmable Technology (ICFPT)*. 296–297. https://doi.org/10.1109/ICFPT51103.2020.00054

[34] Josep Llosa, Eduard Ayguadé, Antonio Gonzalez, Mateo Valero, and Jason Eckhardt. 2001. Lifetime-sensitive modulo scheduling in a production environment. *IEEE Trans. Comput.* 50, 3 (March 2001), 234–249. https://doi.org/10.1109/12.910814

[35] Einstein Morales. 2022. On fast implementations of elliptic curve point multiplication. In *Proceedings of the 2022 ACM Southeast Conference (ACM SE '22)*. Association for Computing Machinery, New York, NY, USA, 173–180. https://doi.org/10.1145/3476883.3520223

[36] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, Jason Anderson, and Koen Bertels. 2016. A Survey and Evaluation of FPGA High-Level Synthesis Tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 10 (Oct. 2016), 1591–1604. https://doi.org/10.1109/TCAD.2015.2513673

[37] Jonas Ney, Dominik Loroch, Vladimir Rybalkin, Nico Weber, Jens Krüger, and Norbert Wehn. 2021. HALF: Holistic Auto Machine Learning for FPGAs. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*. 363–368. https://doi.org/10.1109/FPL53798.2021.00069

[38] Univeristy of Kassel. 2015. Origami HLS. http://www.uni-kassel.de/go/origami. Accessed: 11.01.2022.

[39] Julian Oppermann. 2019. *Advances in ILP-based Modulo Scheduling for High-Level Synthesis*. Ph.D. Thesis. TU Darmstadt. https://tuprints.ulb.tu-darmstadt.de/id/eprint/9272

[40] Julian Oppermann, Andreas Koch, Melanie Reuter-Oppermann, and Oliver Sinnen. 2016. ILP-based modulo scheduling for high-level synthesis. In *2016 International Conference on Compliers, Architectures, and Synthesis of Embedded Systems (CASES)*. 1–10. https://doi.org/10.1145/2968455.2968512

[41] Julian Oppermann, Melanie Reuter-Oppermann, Lukas Sommer, Andreas Koch, and Oliver Sinnen. 2019. Exact and Practical Modulo Scheduling for High-Level Synthesis. *ACM Transactions on Reconfigurable Technology and Systems* 12, 2 (2019), 26.

[42] Julian Oppermann, Patrick Sittel, Martin Kumm, Melanie Reuter-Oppermann, Andreas Koch, and Oliver Sinnen. 2019. Design-Space Exploration with Multi-Objective Resource-Aware Modulo Scheduling. In *Euro-Par 2019: Parallel Processing*, Ramin Yahyapour (Ed.). Springer International Publishing, Cham, 170–183.

[43] Ian Page and Wayne Luk. 1991. Compiling Occam into field-programmable gate arrays. In *FPGAs, Oxford Workshop on Field Programmable Logic and Applications*, Vol. 15. Abingdon EE&CS Books, 271–283.

[44] Arnab Raha, Ankush Chakrabarty, Vijay Raghunathan, and Gregery T. Buzzard. 2020. Embedding Approximate Nonlinear Model Predictive Control at Ultrahigh Speed and Extremely Low Power. *IEEE Transactions on Control Systems Technology* 28, 3 (May 2020), 1092–1099. https://doi.org/10.1109/TCST.2019.2898835

[45] Bantwal R. Rau. 1994. Iterative modulo scheduling: an algorithm for software pipelining loops. In *Proceedings of the 27th annual international symposium on Microarchitecture (MICRO 27)*. Association for Computing Machinery, New York, NY, USA, 63–74. https://doi.org/10.1145/192724.192731

[46] Bantwal R. Rau and Christopher D. Glaeser. 1981. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. *ACM SIGMICRO Newsletter* 12, 4 (Dec. 1981), 183–198. https://doi.org/10.1145/1014192.802449

[47] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. 2014. MachSuite: Benchmarks for accelerator design and customized architectures. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*. 110–119. https://doi.org/10.1109/IISWC.2014.6983050

[48] Patrick Sittel, Nicolai Fiege, John Wickerson, and Peter Zipf. 2022. Optimal and Heuristic Approaches to Modulo Scheduling With Rational Initiation Intervals in Hardware Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 3 (March 2022), 614–627. https://doi.org/10.1109/TCAD.2021.3060320

[49] Patrick Sittel, Julian Oppermann, Martin Kumm, Andreas Koch, and Peter Zipf. 2018. HatScheT: A Contribution to Agile HLS. In *Int. Workshop on FPGAs for Software Programmers*.

[50] Patrick Sittel, Thomas Schönwälder, Martin Kumm, and Peter Zipf. 2018. ScaLP: A Light-Weighted (MI)LP Library. In *Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*. Universität Tübingen, Tübingen, 10.

[51] Babe Sultana, Jannatul Ferdous Katha, Sujan Sarker, and Md. Abdur Razzaque. 2018. Multi-Mode Project Scheduling with Limited Resource and Budget Constraints. In *2018 International Conference on Innovation in Engineering and Technology (ICIET)*. 1–6. https://doi.org/10.1109/CIET.2018.8660864

[52] Robert Tarjan. 1972. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.* 1, 2 (June 1972), 146–160. https://doi.org/10.1137/0201010 Publisher: Society for Industrial and Applied Mathematics.

[53] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. 2017. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, Monterey California USA, 65–74. https://doi.org/10.1145/3020078.3021744

[54] Akihisa Yamada, Satoru Nakamura, Nagisa Ishiura, Isao Shirakawa, and Takashi Kambe. 1995. Optimal scheduling for conditional resource sharing. In *Proceedings of ISCAS'95 - International Symposium on Circuits and Systems*, Vol. 3. 2297–2300 vol.3. https://doi.org/10.1109/ISCAS.1995.523888

[55] Jian Zhao, Yaqin Zhao, Hongbo Li, Yun Zhang, and Longwen Wu. 2020. HLS-Based FPGA Implementation of Convolutional Deep Belief Network for Signal Modulation Recognition. In *IGARSS 2020 - 2020 IEEE International Geoscience and Remote Sensing Symposium*. 6985–6988. https://doi.org/10.1109/IGARSS39084.2020.9324385

[56] Xuan Zhou, Zhong Jun Yu, Yue Cao, and Shuai Jiang. 2019. SAR Imaging Realization with FPGA Based on VIVADO HLS. In *2019 IEEE International Conference on Signal, Information and Data Processing (ICSIDP)*. 1–4. https://doi.org/10.1109/ICSIDP47821.2019.9173161

[57] Přemysl Šůcha and Zdeněk Hanzálek. 2011. A cyclic scheduling problem with an undetermined number of parallel identical processors. *Computational Optimization and Applications* 48, 1 (Jan. 2011), 71–90. https://doi.org/10.1007/s10589-009-9239-4

Received 17 October 2022; revised 24 January 2023; revised 12 May 2023; accepted 14 May 2023