Exact and Practical Modulo Scheduling for High-level Synthesis

JULIAN OPPERMANN, Technische Universität Darmstadt
MELANIE REUTER-OPPERMANN, Karlsruhe Institute of Technology
LUKAS SOMMER, Technische Universität Darmstadt
ANDREAS KOCH, Technische Universität Darmstadt
OLIVER SINNEN, University of Auckland

Loop pipelining is an essential technique in high-level synthesis (HLS) to increase the throughput and resource utilisation of FPGA-based accelerators. It relies on modulo schedulers to compute an operator schedule that allows subsequent loop iterations to overlap partially when executed, while still honouring all precedence and resource constraints. Modulo schedulers face a bi-criteria problem: minimise the initiation interval (II), i.e. the number of time steps after which new iterations are started, and minimise the schedule length.

We present Moovac, a novel exact formulation that models all aspects (including the II minimisation) of the modulo scheduling problem as a single integer linear program (ILP), and discuss simple measures to prevent excessive runtimes, to challenge the old preconception that exact modulo scheduling is impractical.

We substantiate this claim by conducting an experimental study covering 188 loops from two established HLS benchmark suites, four different time limits, and three bounds for the schedule length, to compare our approach against a highly-tuned exact formulation and a state-of-the-art heuristic algorithm. In the fastest configuration, an accumulated runtime of under 16 minutes is spent on scheduling all loops, and proven optimal IIs are found for 179 test instances.

CCS Concepts: • **Hardware → Operations scheduling**; *Reconfigurable logic and FPGAs*;

Additional Key Words and Phrases: modulo scheduling, exact, optimal, II minimisation, high-level synthesis

ACM Reference format:
Julian Oppermann, Melanie Reuter-Oppermann, Lukas Sommer, Andreas Koch, and Oliver Sinnen. 2019. Exact and Practical Modulo Scheduling for High-level Synthesis. *ACM Trans. Reconfig. Technol. Syst.* 12, 2, Article 8 (April 2019), 27 pages.
https://doi.org/10.1145/3317670

Author's addresses: Julian Oppermann, Lukas Sommer, Andreas Koch, {oppermann, sommer, koch}@esa.tu-darmstadt.de, Hochschulstr. 10, 64289 Darmstadt, Germany; Melanie Reuter-Oppermann, melanie.reuter@kit.edu, Kaiserstr. 89, 76131 Karlsruhe, Germany; Oliver Sinnen, o.sinnen@auckland.ac.nz, Private Bag 92019, Auckland 1142, New Zealand.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2019 Association for Computing Machinery.
1936-7406/2019/4-ART8 $15.00
https://doi.org/10.1145/3317670

1 INTRODUCTION

In contrast to software-programmable processors, which in the past have relied on increasing clock frequencies and in recent years switched to coarse-grain parallelism (multi-/many-core), field-programmable gate arrays (FPGA) focus on exploiting fine-grained instruction-level parallelism[1] to achieve high performance. A high-level synthesis (HLS) system that creates FPGA-based accelerators from sequential languages such as C must thus exploit all available sources of parallelism in order to achieve a meaningful speed-up compared to the execution on a software-programmable processor having a higher clock rate. One such source of parallelism is *loop pipelining*[2]: the partially overlapping execution of subsequent loop iterations intended to increase the accelerator's throughput and the utilisation of the functional units within the datapath corresponding to the loop's computation. To this end, new loop iterations are started after a fixed number of time steps, called the *initiation interval (II)*.

Let T be the latency of the datapath representing the loop body. Executing n iterations of the loop sequentially then takes $n \cdot T$ time steps. Assuming that the loop's inter-iteration dependences allow it to be executed with an initiation interval $\lambda < T$, then executing n iterations will require only $(n - 1) \cdot \lambda + T$ time steps, i.e. the last iteration is issued after $(n - 1) \cdot \lambda$ time steps and ends after the T time steps to fully evaluate the result of the datapath. This means that the smaller the interval is relative to the latency of the datapath, the higher is the theoretical speed-up achievable through loop pipelining.

HLS-generated datapaths typically have to obey certain resource constraints, such as the number of requests a memory controller can handle in parallel. Additionally, operations relying on scarce FPGA resources, such as multipliers (called DSPs or DSP slices), might have to be time-multiplexed between different uses in the datapath. We call the operations requiring these constrained hardware blocks *resource-limited operations*.

After an initial warm-up time, a loop's datapath executes operations from different iterations in parallel. Therefore, it is no longer sufficient that an operation schedule fulfils the resource constraints just individually for each time step. With the overlap, the constraints now have to hold for entire congruence classes of time steps (step number modulo II).

Computing a (preferably small) feasible II and the associated operation schedule is called *modulo scheduling*. The corresponding mathematical problem that minimises the II is NP-hard [21].

This paper makes the following contributions to the field.

- We present Moovac[3], an *exact* formulation in the sense that it models all aspects of the problem defined in Section 2.1, allowing it to compute an optimal solution. To the best of our knowledge, Moovac is the first formulation based on integer linear programs (ILP) to integrate the search for the optimal II, but is also competitive in a traditional, one-candidate-II-at-a-time setting. Moovac delivers high-quality results faster than both a state-of-the-art heuristic scheduler and a prior, highly-tuned ILP formulation.
- We investigate strategies to improve the practicability of exact modulo scheduling, namely employing different time limits and upper bounds for the schedule length. In this context, we propose improvements to an existing bound that lead to significantly better schedule length estimates. The effects of the different time limits and bounds on the schedulers' runtime and

[1]Technically, this is *operator*-level parallelism, as computations are typically mapped to datapaths comprised of interconnected operator modules.
[2]In the context of VLIW processors, the equivalent technique is usually called "software pipelining".
[3]"Moovac" is an acronym for "**Mo**dulo **o**verlap **va**riable **c**onstraint".

Table 1. Problem signature for modulo scheduling

Input	
$O = \{1, \ldots, n\}$	operations
$D_i \in \mathbb{N}_0$	delay / latency of operation $i \in O$ (in time steps)
$E = \{(i \to j)\} \subseteq O \times O$	dependence edges
$d_{ij},\ \beta_{ij} \in \mathbb{N}_0$	edge delay, and dependence distance, of $(i \to j) \in E$
$\delta_{ij} = D_i + d_{ij}$	sum of edge's delay and its source operation's delay
$R = \{\text{mem}, \text{dsp}, \ldots\}$	resource types
$a_k \in \mathbb{N}$	available instances of resource type $k \in R$
$L_k \subseteq O$	resource-limited operations of type $k \in R$
$L = \bigcup_{k \in R} L_k$	union of all resource-limited operations
$\lambda^{\perp},\ \lambda^{\top} \in \mathbb{N}$	lower bound and upper bound to the II search space
Output	
$\lambda^{\circ} \in \mathbb{N}$	a feasible initiation interval for the graph
$t_i \in \mathbb{N}_0,\ i \in O$	start time for operation i
$T_{\lambda^{\circ}} = \max_{i \in O}\{t_i + D_i\}$	schedule length for λ°

solution quality are evaluated in an extensive experimental study covering 188 loops from two HLS benchmark suites.

A preliminary version of this work appeared as [24].

2 BACKGROUND

We will now formally define the modulo scheduling problem (MSP) and present prior scheduling approaches.

2.1 The modulo scheduling problem

The MSP is characterised by the signature in Table 1.

Input. The input comprises a directed graph consisting of a set of operations $i \in O$ (nodes) with a fixed integer latency of D_i, and a set $E \subseteq O \times O$ of directed edges $(i \to j)$ modelling the dataflow and other precedence relations among the operations. The edges carry a delay d_{ij}. We define $\delta_{ij} = D_i + d_{ij}$ to denote the sum of an edge's delay and its source operation's delay.

In addition to the usual intra-iteration dependences, a loop may contain inter-iteration dependences, also known as recurrences or loop-carried dependences. As the latter dependences point in the opposite direction of the normal dataflow, we also call them *backedges*. In order to handle both types of dependences uniformly, each edge is associated with a dependence *distance* β_{ij} that specifies how many iterations later the dependence has to hold. Intra-iteration dependences therefore have $\beta_{ij} = 0$, whereas inter-iteration dependences are characterised by $\beta_{ij} \geq 1$.

The operations O are carried out on resources (e.g. logic gates, DSPs, memories …), of which some types, but not all, are limited in number[4]. Let R be the set of distinct resource types, whose

[4]While in reality all resources are limited, some, e.g. logic gates, can be meaningfully considered unlimited on current FPGAs.

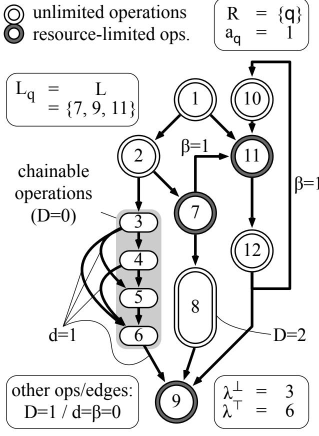


Fig. 1. Example problem

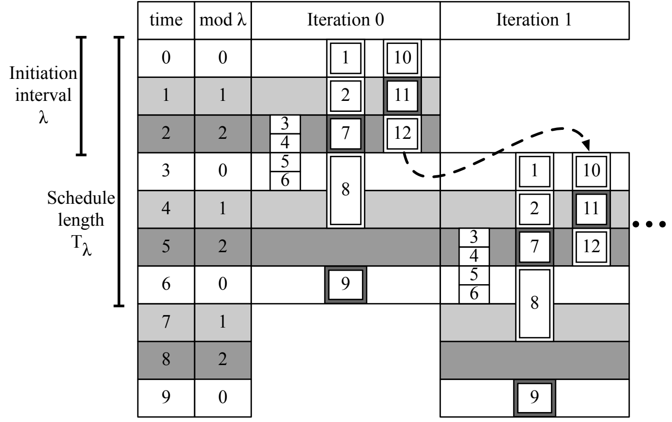


Fig. 2. Modulo schedule for $\lambda = 3$ with $T_3 = 7$

usage has to be limited when scheduling. We assume that every resource type $k \in R$ provides a_k uniform and fully-pipelined *instances* that can accept new input data from at most one operation at any time. This means that at most a_k operations scheduled to start in time steps in the same congruence class (modulo II) can use resources of type k concurrently.

We further assume that every operation requires at most one limited resource of type k. We represent these resource-limited operations as members of the respective sets L_k. For brevity, we define L to be the set of all resource-limited operations.

Lastly, the search space for feasible and practically relevant IIs is bounded by the parameters $\lambda^{\perp}$ (lower bound) and $\lambda^{\top}$ (upper bound), which are explained in Section 2.2.

Figure 1 illustrates a complete MSP instance that will serve as a running example. It contains three operations (7), (9) and (11) (distinguished by their dark border) that compete for access to the single instance of resource type q, as well as two backedges ((7)→(11)) and ((12)→(10)) with a backedge distance of 1. If not stated otherwise, the operations have a delay of 1, and edges do not incur a propagation delay.

Output. We seek an initiation interval λ° and an integer start time t_i for each operation, so that all dependence edges are honoured, i.e.

$$t_i + \delta_{ij} \leq t_j + \beta_{ij} \cdot \lambda^{\circ} \quad \forall (i \rightarrow j) \in E, \tag{1}$$

and no resource type is oversubscribed in any congruence class (modulo II), i.e.

$$|\{i \in L_k : t_i \bmod \lambda^{\circ} = m\}| \leq a_k \quad \forall k \in R \text{ and } m \in [0, \lambda^{\circ} - 1]. \tag{2}$$

For any feasible interval λ, the start times imply the schedule length $T_\lambda = \max_{i \in O}\{t_i + D_i\}$, i.e. the time step in which the last operation finishes. In particular, we are interested in the schedule length $T_{\lambda^{\circ}}$ corresponding to the computed interval λ°. The schedule length is conceptually equivalent to the latency of the datapath and the makespan of the problem graph.

Figure 2 shows a valid modulo schedule for the running example with an initiation interval of 3. The operations' start times t_i can be read off the column representing the first iteration. Observe that the backedge ((12)→(10)) is obeyed under the overlapping, and at any given time step, only one of the resource-limited operations is active.

Objective. Modulo scheduling is a bi-criteria optimisation problem. The first objective is to find the smallest II that satisfies (1) and (2). We refer to the optimal II according to this objective as $\lambda^\star$. The second objective is to find a schedule for an interval λ with the minimal schedule length, denoted by $T^\star_\lambda$.

In practice, minimising the II is far more important than minimising the schedule length, which implies an order of the objectives. To this end, the overall aim is to lexicographically minimise the tuple $S = (\lambda^\circ, T_{\lambda^\circ})$. The optimal solution to the MSP then is a schedule with $S^\star = (\lambda^\star, T^\star_{\lambda^\star})$.

2.2 Bounds for the II search space

Modulo schedulers typically consider one or more *candidate* IIs during their operation. The search space for sensible II values is defined by the following bounds.

A trivial **upper bound** $\lambda^\top$ is the length of any resource-constrained *non-modulo* schedule. IIs larger than this value indicate that it would actually be faster to execute this loop in a non-overlapping manner. We use a non-modulo SDC scheduler [7] with heuristic resource constraints to quickly compute such a fallback schedule, and use its length to define $\lambda^\top$.

The **lower bound** $\lambda^\perp$ is usually defined (e.g. in [25]) as $\lambda^\perp = \max(\lambda^\perp_{\text{rec}}, \lambda^\perp_{\text{res}})$, i.e. the maximum of the recurrence-constrained minimum II and the resource-constrained minimum II. Due to (1), the recurrences (cycles) in the dependence graph impose a lower bound for any feasible II. We compute $\lambda^\perp_{\text{rec}}$ as the optimal solution to the following ILP[5], defined for integer variables t_i that model the start time step for each operation i, and an integer variable λ that models the recurrence-induced II to be minimised.

$$\mathbf{min}\ \lambda \tag{3}$$

$$\text{s.t. } t_i + \delta_{ij} \ \leq\ t_j + \beta_{ij} \cdot \lambda \quad \forall (i \longrightarrow j) \in E \tag{4}$$

The resource-constrained minimum II is defined as $\lambda^\perp_{\text{res}} = \max_{k \in R} \left\lceil \frac{|L_k|}{a_k} \right\rceil$ and follows from (2). It is an application of the pigeonhole principle: A candidate interval λ cannot be feasible if the loop contains more operations using a limited resource k than can be assigned to the available resource instances in every modulo slot $0 \ldots (\lambda - 1)$. Recall that we assume the operations itself to be fully-pipelined, meaning that they are able to accept new input values in every time step.

In the running example, we determine $\lambda^\perp = 3$: the recurrence spanned by backedge ((12)⟶(10)) leads to $\lambda^\perp_{\text{rec}} = 3$. Also, we have $\lambda^\perp_{\text{res}} = 3$ due to the presence of three resource-limited operations competing for access to one instance of type q. In this particular example, a simple as-soon-as-possible schedule has a length of 6 and can serve as a valid resource-constrained non-modulo schedule. Thus, we set $\lambda^\top = 6$.

2.3 Operator chaining

The scheduling problem at hand is defined in terms of time steps. The operations in a loop's datapath will be started according to their assigned start time step by a controller circuit that is also generated by the HLS tool.

Every combinatorial operation i (these operations have $D_i = 0$) instantiated on the FPGA requires a non-zero amount of physical propagation time z_i (e.g. in nanoseconds) to complete. Typically, a

[5]As noted by de Dinechin [8], this is a resource-free cyclic scheduling problem, which can be solved optimally in polynomial time.

HLS tool maintains a desired target cycle time Z for the datapath, which limits the time that can be spent in a single time step. Operations in different time steps are decoupled by registers that hold intermediate results.

However, not all operations require the same amount of time, e.g. logical operations and constant bit shifts are trivial to perform on an FPGA, and have $z_i \ll Z$. This allows HLS tools to schedule a *chain* of data-dependent operations into a single time step, as long as the *accumulated* physical delay of the operations in the chain does not exceed the desired cycle time.

We support operator chaining by allowing *chainable* operations $i \in O$ with $D_i = 0$, and construct additional edges to prohibit excessive chaining right before the actual scheduling. The basic idea is to discover the longest (in terms of physical delay) combinatorial paths $\chi_{u_1 u_p} = (u_1, \ldots, u_p)$ with $D_{u_l} = 0 \; \forall 1 \leq l \leq p$ between all pairs of a chainable operations u_1 and u_p. Let $\zeta_{u_1 u_p} = \sum_{i \in \chi_{u_1 u_p}} z_i$ be the accumulated physical delay along such a path. If $Z < \zeta_{u_1 u_p} \leq 2 \cdot Z$, u_1 and u_p cannot be scheduled to the same time step, and therefore have to be separated by an edge $(u_1 \rightarrow u_p)$ with $d_{u_1 u_p} = 1$. In all other cases, no edge is needed, as either all operations of the chain can be scheduled to the same time step ($\zeta_{u_1 u_p} \leq Z$), or the separation of u_1 and u_p is transitively guaranteed ($\zeta_{u_1 u_p} > 2 \cdot Z$). Additionally, all edges from chainable to non-chainable operations also carry an edge delay of 1.

Note that these edges correspond to the *cycle time constraints* proposed by Cong and Zhang [7]. However, the advantage of explicitly adding them a priori to the MSP, instead of inferring equivalent constraints when constructing the linear programs, is that the underlying path enumeration has to be performed only once and thus can be reused when computing the bounds of the II search space, and for each candidate II.

In the running example, a uniform physical propagation delay of 5 ns for the chainable operations ③ - ⑥, and a target cycle time of 10 ns, was assumed. Note the several edges with an edge delay of 1 that limit the amount of chaining to meet the desired cycle time.

2.4 Related work

Loop pipelining is most useful when targeting processors with multiple parallel function units. Out-of-order processors rely on dynamic scheduling to derive the appropriate execution sequences at run-time. Parallel in-order processors (such as VLIW architectures), or statically scheduled hardware accelerators created by HLS, however, rely on the compiler/synthesis tool to pre-compute their execution schedules.

The underlying MSP is the same, with the primary objective to find a schedule with the minimally feasible II. However, VLIW modulo schedulers (e.g. [9, 12, 17, 21, 23, 25]) typically face tight resource constraints in each cycle (due to the limited number of functional units, registers, and buses). Often, minimising the usage of one or more of these resources, e.g. the register pressure, is used as the secondary objective instead of a schedule length minimisation.

Modulo schedulers targeting HLS (e.g. [5, 29]) face fewer resource constraints (hardware accelerators may use dozens or even hundreds of operators), but need to support operator chaining to achieve acceptable performance. Thus, when considering related work, not all techniques proposed for one target may be beneficial for the other. Especially heuristic approaches may incorporate the chosen secondary objective at the algorithmic level, making it harder to repurpose them for a different problem variant than approaches based on mathematical formulations.

In this work, we evaluate different approaches as measured by their suitability for *HLS modulo scheduling*.

Modulo schedules can be determined either by **heuristics** or by using mathematical formulations to solve the problem **exactly**. While heuristics are not guaranteed to find the optimal solution

(and often only produce feasible solutions), they usually come with a shorter computation time compared to the exact approaches, which are typically associated with impractically long runtimes.

Heuristics. Since Lam [21] and Rau [25] laid most of the groundwork of the modulo scheduling framework, many heuristic modulo schedulers have been proposed (e.g. [17, 23]). Codina et al. [6] explain the differences in the most relevant approaches, and provide a detailed performance evaluation. More recently, Zhang and Liu [29] implemented a modulo scheduler on top of a System of Difference Constraints (SDC) [7]. By construction, an SDC is represented by a totally unimodular constraint matrix, which guarantees that an optimal solution found by an LP solver will only consist of integer values for the decision variables. The flexibility of the SDC framework enables the easy integration of operator chaining into the modulo scheduling process. Improvements to this idea resulted in the Modulo SDC scheduler proposed by Canis et al. [5], which uses a modulo reservation table and a backtracking algorithm to handle the resource constraints heuristically, solving the underlying SDC multiple times in the process. We include this approach in our experimental evaluation as an advanced heuristic algorithm, and the representative of the state-of-the-art in HLS-specific modulo scheduling.

All known heuristic approaches attempt to minimise the II by trying to find a feasible schedule for various candidate IIs.

Exact approaches. Previously, the MSP has also been modelled exactly as an integer linear program (ILP) for a single candidate interval λ. Several formulations have been proposed that can be classified roughly according to their modelling of the operations' start times t_i, and handling of the resource constraints.

Time-indexed formulations use binary decision variables $x_{i,t}$ per operation i and time step t to represent that i starts in t. The resource usage is constrained by comparing the sum of the $x_{i,t}$ variables over the relevant time steps with the number of available resource instances.

The formulation by Dinechin [9] exclusively uses such binaries to model the time steps up to a predefined time horizon, i.e. an upper bound U for the schedule length, as $t_i = \sum_{t=0}^{U} t \cdot x_{i,t}$. As the formulation cannot be efficiently solved for large instances, a large neighbourhood search (LNS) heuristic is proposed. It speeds up the scheduling process as a schedule for $\lambda - 1$ can easily be constructed (if it exists) given a feasible solution for λ.

In the formulation by Eichenberger and Davidson [12], the start times are decomposed according to the Euclidean division by λ, as $t_i = y_i \cdot \lambda + \sum_{t=0}^{\lambda-1} t \cdot x_{i,t}$. Here, binary decision variables model the assignment of the start time to the congruence classes $0 \ldots \lambda - 1$, but integer variables y_i represent the so-called stage, i.e. the multiple of λ contained in t_i. The formulation models dependence edges as λ-many, but 0-1-structured constraints, which resulted in an almost tenfold speed-up over a prior version [13].

Ayala and Artigues [3] compare the aforementioned time-indexed formulations and report that the one by Eichenberger and Davidson [12] is always faster on their set of industrial instances. The authors obtain stronger variants of the formulations by applying a Dantzig-Wolfe decomposition, but rely on a column-generation scheme to cope with the increased number of decision variables.

Formulations using *integer* variables to directly model the (potentially decomposed) start times promise to work with fewer decision variables overall. Our proposed formulation, Moovac, is an extension to such a formulation from a task scheduling context by Venugopalan and Sinnen [28]. A common element in integer-based formulations is the concept of *overlap* variables, i.e.

binary variables that express an ordering relation on the values of the t_i. Similar formulations were proposed by Altman et al. [2] and Šůcha and Hanzálek [27].

In [2], the overlap scheme was inspired by a graph colouring problem. In contrast to Moovac, the set of constraints in the formulation is not independent of λ, which is a precondition for the integration of the II minimisation into the formulation.

The overlap mechanism in the formulation[6] by Šůcha and Hanzálek [27] differs semantically from ours: they define binary decision variables $\hat{y}_{ij}$ to be 1 iff operations i and j are scheduled to the same congruence class and therefore cannot use the same resource instance. For every operation i, the number of conflicts i is involved in is counted and must be less than the number of available resource instances of a given type. In comparison, our formulation approach models conflicting resource instance and congruence class usage in a combined constraint that is defined pairwise. Besides being more concise, this allows for future extensions such as selectively allowing oversubscription of resource instances for mutually exclusive operations.

Fimmel and Müller [15] present an approach with II minimisation for a variant of the MSP where the II is allowed to be a rational number, which they claim to be beneficial especially for small IIs. Their formulation is a linear program comprised of binary and rational decision variables. The resulting schedules associate each operation with two iteration-dependent start time steps. While it would be possible to support such an execution model in an HLS tool, doing so would incur an additional multiplexing overhead. Also, it remains unclear how they modelled the rational variables in a standard ILP solver, which usually only support real- and integer-valued variables. Therefore, we do not include this approach in our evaluation.

In addition to ILPs, the MSP can be solved with an enumeration scheme and extensive pruning [1], or constraint programming (CP) [4, 11]. The CP framework allows to formulate more powerful constraints, and therefore needs less abstractions as an ILP model. For example, comparable to our proposed approach, Bonfietti et al. [4] also integrate the II minimisation into their scheduler. The CP-based approaches can achieve competitive or better performance compared to heuristics and ILP formulations by implementing highly problem-specific search strategies instead of the more generic branch-and-bound techniques typically employed in ILP solvers. On the other hand, implementing such a scheduler involves a significant engineering effort using a CP solver library, whereas an ILP model such as ours can be passed almost verbatim to the ILP solver's API. Also, ILP-based approaches may benefit from improvements to the solver's internal algorithms without modification.

From this variety of exact approaches, we compare our proposed formulation against the formulation by Eichenberger and Davidson [12], as it is a common reference point in the works cited above, and generally had a lasting impact on the field [14]. At the same time, it appears to be the best-known alternative formulation to be solved by an unmodified ILP solver, due to its efficient internal structure. For brevity, we refer to this work as just "Eichenberger's" formulation in the rest of this paper.

3 THE MOOVAC FORMULATION

We propose to tackle the MSP by including as much information as possible into an Integer Linear Program (ILP). To this end, we integrate the II minimisation into the formulation and obtain a bi-criteria problem that we call **Moovac-I**. As discussed in Section 2.4, there are various different

[6] More precisely, our resource model with fully-pipelined operators corresponds to their unit processing time model.

Table 2. Moovac-S: Decision variables

T	$\in \mathbb{N}_0$		latest finish time (schedule length)
t_i	$\in \mathbb{N}_0$	$\forall i \in O$	start time
r_i	$\in \mathbb{N}_0$	$\forall i \in L$	index of resource instance used by operation
m_i	$\in \mathbb{N}_0$	$\forall i \in L$	index of congruence class (modulo II)
y_i	$\in \mathbb{N}_0$	$\forall i \in L$	helper in congruence class computation
ε_{ij}	$\in \{0, 1\}$	$\forall k : i, j \in L_k$	$\begin{cases} 1 & r_i < r_j \\ 0 & \text{otherwise} \end{cases}$
μ_{ij}	$\in \{0, 1\}$	$\forall k : i, j \in L_k$	$\begin{cases} 1 & m_i < m_j \\ 0 & \text{otherwise} \end{cases}$

approaches how to formulate a scheduling problem as an ILP, using different decision variables. Recent work on a related scheduling problem has shown that using overlap variables is more efficient than other approaches [28], so we take this approach here. Together with a linearisation of otherwise quadratic constraints, we expect to achieve a problem structure that can be solved in reasonable time for practically relevant problem instances. In addition, the *entire* runtime is used to determine a feasible and potentially optimal solution, while a practical downside of single-II approaches is that the time spent on *unsuccessful* candidate IIs is ultimately lost.

We also introduce a non-integrated variant of the Moovac formulation that follows the traditional approach to modulo scheduling and models a *single* scheduling attempt for a particular candidate II. Thus, we solve a formulation with only one objective, i.e. to minimise the schedule length. Solving the MSP then requires an external driver that traverses[7] the II search space until a feasible solution is found. This single-II variant of the Moovac formulation is called **Moovac-S**.

We start by defining the simpler formulation **Moovac-S** and then present the necessary extensions for obtaining the more general **Moovac-I** formulation.

3.1 Moovac-S

The problem signature of the Moovac-S formulation differs slightly from the general signature in Table 1, as the II minimisation is handled outside of the linear program that is introduced in the following sections. The bounds to the II search space, $\lambda^{\perp}$ and $\lambda^{\top}$, are therefore passed to an external driver, which is also responsible to return the resulting II value λ°. During the search, the driver chooses one or more candidate intervals λ that are passed to the ILP. Inside of the ILP, λ is a constant.

Decision variables. We model the problem with the decision variables shown in Table 2: As stated in the problem signature, the output we are seeking is a start time t_i for every operation $i \in O$. These variables are directly part of our ILP formulation. With the externally specified delay D_i, an operation i's result will be available in time step $t_i + D_i$. The variable T captures the latest finish time across all operations.

[7]Traditionally, increasing candidate intervals $\lambda = \lambda^{\perp}, \lambda^{\perp} + 1, \ldots, \lambda^{\top}$ are tried. We discuss in Section 5.4 why this is still the most viable approach for our set of test instances.

For every resource-limited operation $i \in L_k$ using resource type k, we model the allocation of one of the a_k available instances to i with an integer variable r_i, which contains an index in the range $[0, a_k - 1]$.

Due to the nature of the modulo schedule, the start time of every operation t_i can be decomposed into a multiple y_i of λ plus a remainder m_i which is less than λ. The start time t_i is then $t_i = y_i \cdot \lambda + m_i$ We define such variables for all resource-limited operations $i \in L$. m_i is the congruence class (modulo II) implied by the current start time t_i and is represented by an integer in the range $[0, \lambda - 1]$. y_i's value is bound to the integer division t_i/λ.

Resource limit mechanism. A valid modulo schedule must not oversubscribe any of the resource types $k \in R$ in any congruence class $m \in [0, \lambda^\circ]$ (cf. (2) in Section 2.1). A common abstraction of this condition is the modulo reservation table (MRT) [25]. As illustrated in Figure 3, an MRT contains a row for each resource instance, and a column for each congruence class (modulo the current candidate II). Heuristic modulo schedulers often use an MRT as an explicit data structure, and successively allocate operations to the cells of the table. In order for a schedule to be valid, it must be ensured that each cell is occupied by at most one operation.

While we do not use an explicit MRT in the Moovac formulation, it is still a useful intuition, as the r_i and m_i variables belonging to an operation $i \in L_k$ induce an MRT-like structure for resource type k. Our goal therefore is to model that no pair of operations $i, j \in L_k$ shares an MRT cell. Formally, we wish to enforce

$$r_i \neq r_j \vee m_i \neq m_j \quad \forall i, j \in L_k, i \neq j.$$

However, the inequality relations as well as the disjunction need to be linearised for the ILP formulation.

To this end, we introduce the following *overlap* variables on all pairs of operations $i, j \in L_k, i \neq j$ that use the same resource type k: $\varepsilon_{ij} = 1$ indicates that i's resource instance index is strictly less than j's resource instance index. Analogously, $\mu_{ij} = 1$ models that i's congruence class index is strictly less than j's congruence class index. With these variables, we can express $r_i \neq r_j \Leftrightarrow r_i < r_j \vee r_i > r_j \Leftrightarrow \varepsilon_{ij} + \varepsilon_{ji} \geq 1$, and $m_i \neq m_j \Leftrightarrow \mu_{ij} + \mu_{ji} \geq 1$. It follows that i and j are not in resource conflict if and only if $\varepsilon_{ij} + \varepsilon_{ji} + \mu_{ij} + \mu_{ji} \geq 1$ is satisfied. Figure 4 demonstrates the interaction between two operations' r and m variables and their corresponding overlap variables.

Note that the externally specified resource limits a_k are modelled indirectly through the range of resource instance indices $[0, a_k - 1]$ that can be assigned to the r_i variables, or in terms of the MRT intuition, through the number of rows that can be occupied by operations in each congruence class. In contrast to time-indexed formulations such as Eichenberger's, we do not explicitly count the number of limited operations in a particular congruence class to ensure (2).

Constraints. Using the decision variables defined in Table 2, the ILP formulation of Moovas-S is given in Figure 5. The default objective is to minimise the schedule length (M1).

We model the dependence edges with (M2), which are directly adopted from the precedence constraint (1) in the definition of the MSP.

As both sets of overlap variables are defined by a *strictly less* relation, for a given pair of operations i, j, ε_{ij} and ε_{ji}, as well as μ_{ij} and μ_{ji} cannot be 1 at same time. This is ensured by (M3) and (M6).

The overlap variables are bound to their desired values by the pairs of constraints (M4)+(M5) and (M7)+(M8), respectively. For brevity, we explain their function in the context of μ_{ij}, as the constraints for ε_{ij} work analogously. (M7) are fulfilled if $m_i < m_j$ or $\mu_{ij} = 0$, and (M8) are fulfilled if $m_i \geq m_j$ or $\mu_{ij} = 1$. In these constraints, the second expression uses the candidate II as a big-M constant, meaning that its value is big enough to fulfil the constraint regardless of the rest of the

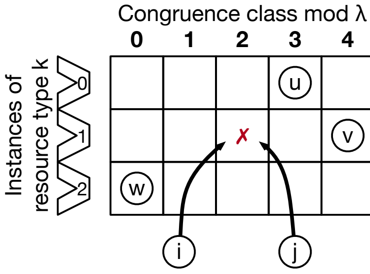


Fig. 3. Modulo reservation table as induced by the operations' r- and m-decision variables. Each resource instance can only be used by one operation per congruence class, thus each cell can be occupied by at most one operation.

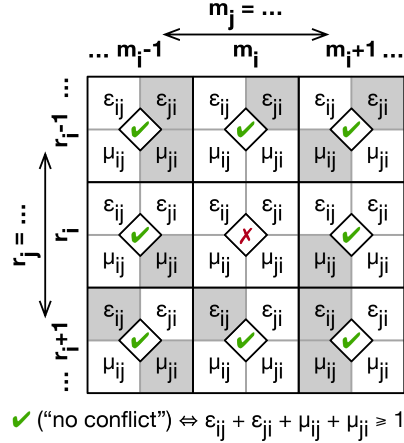


Fig. 4. Consider two operations i and j that compete for the same MRT cell as indicated in Figure 3. This sketch shows the values of the overlap variables (white = '0', grey = '1') for different assignments of r_j and m_j in relation to r_i and m_i. For example, in the top right corner, we assume $r_j = r_i - 1$ and $m_j = m_i + 1$, which is an assignment that does not result in a resource conflict.

expression. Due to the apparent contradictions, these constraints can only be fulfilled if and only if $m_i < m_j$ and $\mu_{ij} = 1$, or $m_i \geq m_j$ and $\mu_{ij} = 0$, resulting in the desired behaviour.

As derived above, constraints (M9) ensure a conflict-free resource usage for every pair of operations i, j, i.e. the operations are either assigned to different resource instances, mapped to different congruence classes, or both.

(M10) define the congruence class index m_i as expressed by the operation's start time t_i modulo II.

(M11) bound the resource instance indices for each limited operation of type k to be less than the externally specified limit a_k. Analogously, (M12) ensure that an operation's congruence class index is less or equal to the candidate II.

The latest finish time T is defined by (M13) to be greater or equal to the finish times of sink operations, i.e. operations that do not have outgoing forward edges $(i \longrightarrow j)$ with $\beta_{ij} = 0$.

(M14) - (M16) are domain constraints to enforce non-negativity respectively boolean values for the decision variables.

3.2 Moovac-I

The Moovac-I formulation conforms to the problem signature in Table 1.

Note that the structure of the linear program as defined by the Moovac-S formulation is already independent of the concrete value of the candidate II: The set of decision variables and constraints is the same, and only the numerical values in the constraints differ between scheduling attempts. This is not the case in time-indexed formulations, such as Eichenberger's, where the numbers of binary variables $x_{i,m}$ and constraints (e.g. (20) in [12]) vary with the candidate II.

$$
\begin{array}{llll}
\mathbf{min}\ T & & & \text{(M1)} \\
\text{s.t.}\quad t_i + \delta_{ij} & \leq t_j + \beta_{ij} \cdot \lambda & \forall (i \rightarrow j) \in E & \text{(M2)} \\
\\
\varepsilon_{ij} + \varepsilon_{ji} & \leq 1 & \forall k \in R : \forall i,j \in L_k, i \neq j & \text{(M3)} \\
r_j - r_i - 1 - (\varepsilon_{ij} - 1) \cdot a_k & \geq 0 & \forall k \in R : \forall i,j \in L_k, i \neq j & \text{(M4)} \\
r_j - r_i - \varepsilon_{ij} \cdot a_k & \leq 0 & \forall k \in R : \forall i,j \in L_k, i \neq j & \text{(M5)} \\
\\
\mu_{ij} + \mu_{ji} & \leq 1 & \forall k \in R : \forall i,j \in L_k, i \neq j & \text{(M6)} \\
m_j - m_i - 1 - (\mu_{ij} - 1) \cdot \lambda & \geq 0 & \forall k \in R : \forall i,j \in L_k, i \neq j & \text{(M7)} \\
m_j - m_i - \mu_{ij} \cdot \lambda & \leq 0 & \forall k \in R : \forall i,j \in L_k, i \neq j & \text{(M8)} \\
\\
\varepsilon_{ij} + \varepsilon_{ji} + \mu_{ij} + \mu_{ji} & \geq 1 & \forall k \in R : \forall i,j \in L_k, i \neq j & \text{(M9)} \\
\\
t_i & = y_i \cdot \lambda + m_i & \forall i \in L & \text{(M10)} \\
r_i & \leq a_k - 1 & \forall k \in R : \forall i \in L_k & \text{(M11)} \\
m_i & \leq \lambda - 1 & \forall i \in L & \text{(M12)} \\
\\
t_i + D_i & \leq T & \forall i \in O : \nexists j \in O : (i \rightarrow j) \in E \wedge \beta_{ij} = 0 & \text{(M13)} \\
\\
t_i & \in \mathbb{N}_0 & \forall i \in O & \text{(M14)} \\
r_i, y_i, m_i & \in \mathbb{N}_0 & \forall i \in L & \text{(M15)} \\
\varepsilon_{ij}, \mu_{ij} & \in \{0,1\} & \forall k \in R : \forall i,j \in L_k, i \neq j & \text{(M16)}
\end{array}
$$

Fig. 5. Moovac-S: Objective function and constraints for a candidate interval λ

In order to integrate the II minimisation into the Moovac-S formulation, we replace the formerly constant candidate II with a new integer decision variable λ that is bounded to the II search space as $\lambda^{\perp} \leq \lambda \leq \lambda^{\top}$. However, integrating the II minimisation naively has a major drawback: It results in quadratic (i.e. containing a multiplication of decision variables) constraints (M7), (M8) and (M10).

We linearise constraints (M7), (M8) by replacing the occurrence of λ with the upper bound of the II search space, $\lambda^{\top}$:

$$m_j - m_i - 1 - (\mu_{ij} - 1) \cdot \boxed{\lambda^{\top}} \geq 0 \quad \forall k \in R : \forall i,j \in L_k, i \neq j \tag{M17}$$

$$m_j - m_i - \mu_{ij} \cdot \boxed{\lambda^{\top}} \leq 0 \quad \forall k \in R : \forall i,j \in L_k, i \neq j \tag{M18}$$

This has no effect on the constraints' functionality, as the interval value is used as a big-M constant here.

The remaining quadratic constraints (M10) are broken down into individual constraints for all possible values of y_i. To this end, we need to introduce an upper bound Y for the y_i variables. Recall that the y_i variables represent the value $\left\lfloor \frac{t_i}{\lambda} \right\rfloor$ in the calculation of operation i's congruence class.

Given an upper bound U for the schedule length[8] (and in consequence, for all t_i), and using the II search space's lower bound $\lambda^{\perp}$, the y_i variables are bounded by $Y = \left\lfloor \frac{U}{\lambda^{\perp}} \right\rfloor$.

With this bound and introducing new binary variables $\gamma_{\bar{y}i}$ for $\bar{y} \in [0, Y]$ and all resource-limited operations, constraints (M10) are replaced by:

$$\gamma_{\bar{y}i} = 1 \quad \rightarrow t_i = \bar{y} \cdot \lambda + m_i \quad \forall \bar{y} \in [0, Y] : \forall i \in L \tag{M19}$$

$$\sum_{\bar{y}=0}^{Y} \gamma_{\bar{y}i} \quad \geq 1 \quad \forall i \in L \tag{M20}$$

$$\sum_{\bar{y}=0}^{Y} \bar{y} \cdot \gamma_{\bar{y}i} = y_i \quad \forall i \in L \tag{M21}$$

The indicator[9] constraints (M19) conditionally model the modulo decomposition for every possible value of y_i. Constraints (M20) force that at least one of these linearised decompositions is selected for the solution. Note that the $\geq$-constraints are sufficient here, as at most one $\gamma_{\bar{y}i}$ can be non-zero due to the mutually-exclusive nature of the decomposition. Lastly, constraints (M21) define the value of y_i according to the selected decomposition. These constraints are not required for the correctness of the Moovac-I formulation, but make solutions interchangeable between Moovac-I and Moovac-S, a property we leverage in Section 4.1.

The MSP's bi-criteria objective can now be modelled directly in the Moovac-I formulation, which is defined by:

$$\mathbf{min}\ \lambda \tag{M22}$$

$$\mathbf{min}\ T \tag{M23}$$

$$s.t.\ (M2) - (M6), (M9), (M11) - (M16), (M17) - (M21)$$

4 STRATEGIES FOR MODULO SCHEDULING IN PRACTICE

We now discuss the use of time limits, and bounds on the schedule length. A time limit allows to cap the worst-case solution times, whereas a schedule length bound is required to make the integration of the II search in the Moovac-I formulation possible, and can help to narrow down the solution space.

4.1 Time-limited operation

Since modulo scheduling is an NP-hard problem [21], we cannot rule out the possibility to encounter problem instances that lead to exponential runtimes when solved with the exact approaches, or cause the heuristic approach to get stuck.

To this end, we impose a time limit τ per candidate II. For all approaches, this includes the time κ to construct the linear program via the solver's API. Combining the exact approaches with a time limit can lead to non-optimal solutions, however, we retain the ability to make statements about the optimality of the returned solutions, as discussed in the following paragraphs.

Moovac-S, Eichenberger. Each scheduling attempt with the single-II Moovac formulation as well as with Eichenberger's formulation is executed as one invocation of the ILP solver, so a time limit of $\tau - \kappa$ can directly be specified via its API.

[8] We discuss possible upper bounds in Section 4.2.

[9] An indicator constraint is of the form $x = f \rightarrow \langle \text{cons} \rangle$: if the binary variable x has the value f, the right-hand side constraint $\langle \text{cons} \rangle$ must be satisfied; otherwise, it may be violated. Indicator constraints are supported natively in modern ILP solvers.

When the solver returns from a scheduling attempt for a candidate interval λ, it reports one of the following outcomes:

- Optimal. A schedule was found, and the solver *proved* that its length T_λ is optimal, i.e. $T_\lambda = T^\star_\lambda$.
- Feasible. A schedule with length T_λ was found, but the solver was unable to determine whether it is optimal within the time limit. We only know that $T_\lambda \geq T^\star_\lambda$.
- Unknown. No schedule was found within the time limit. We conservatively consider λ to be infeasible.
- Infeasible. The solver proved that the attempt is infeasible.

Let λ° be the smallest II for which at least a feasible modulo schedule was found. λ° is the optimal II if, trivially, $\lambda^\circ = \lambda^\perp$, or if all candidates $\lambda^- \in [\lambda^\perp, \lambda^\circ)$ are known to be infeasible. In case scheduling for λ° yielded an optimal schedule length, we have also found an overall optimal solution $S^\star = (\lambda^\circ, T^\star_{\lambda^\circ})$, otherwise we only have $\lambda^\circ = \lambda^\star$.

Modulo SDC. The Modulo SDC algorithm starts a scheduling attempt with computing a non-resource constrained schedule, and iteratively tries to assign resource-limited operations to their designated time steps. The modulo reservation table (MRT) is queried for resource conflicts, and if no conflicts arise, an operation's assignment is fixed in the underlying SDC by adding new equality constraints. Otherwise, a new constraint to move the conflicting operation to the next time step is introduced. The SDC is solved afterwards to check the feasibility of the current partial schedule. Should the schedule become infeasible, the algorithm uses backtracking to revert some of the previous assignments and resumes.

It is apparent that such a scheduling attempt necessitates constructing and solving multiple linear programs. We thus do not impose our time limit on the individual invocations of the LP solver, but instead on the scheduling attempt as a whole, i.e. including the solver runtimes and all MRT and backtracking operations.

In case the current candidate II is infeasible, the algorithm would potentially run until all possible schedules were inspected. To that end, the algorithm maintains a budget of $6 \cdot |O|$ backtracking steps [5] to provide another failsafe for fruitless scheduling attempts. This mechanism is still in place in our implementation, as otherwise, even simple problem instances would deplete the whole time budget on infeasible candidate IIs.

The only situation in which an interval λ° returned by the Modulo SDC scheduler is known to be optimal is if $\lambda^\circ = \lambda^\perp$, as the algorithm can only run out of time or backtracking steps, but never prove infeasibility of a candidate II.

The algorithm contains no means to determine the optimality of the found schedule length.

Moovac-I. The integration of the II search into the Moovac-I formulation enables a different solution strategy. Instead of performing the minimisation of the first objective (= the initiation interval) by iteratively solving and minimising multiple ILPs for the second objective (= here, the schedule length), we can optimise both objectives directly using a single Moovac-I-ILP.

ILP solvers handle multiple objectives either by attempting to solve the problem only once with user-specified weights for the objective functions, or by addressing the different objectives in individual steps, according to priorities given by the user. The latter, multi-step approach is more suitable in the context of the modulo scheduling problems, because practitioners would almost certainly choose weights that strongly favor better IIs anyway, as discussed in Section 2.1. Some ILP solvers offer an API to perform multi-objective optimisation automatically. However, in order for our approach to be solver-independent, and because it makes it easier to reason about the solution quality in the presence of time limits, we implement the multi-step approach ourselves.

To this end, we begin by setting a time limit of τ and instruct the solver to only minimise the II. When the solver returns, it reports one of the following outcomes:

- Optimal. A schedule for an interval λ° was found and *proven* to be optimal, i.e. $\lambda^\circ = \lambda^\star$.
- Feasible. A schedule for an interval λ° was found, but the solver was unable to determine whether it is optimal within the time limit. We only know that λ° is optimal if it is equal to $\lambda^\perp$.
- Unknown. No schedule was found within the time limit; give up and report failure.

The interval computed in the first step, λ°, is fixed during the second step. This allows us to reduce the complexity of the current ILP by "downgrading" it to resemble the Moovac-S formulation. We add a constraint to bind the decision variable λ to λ°, and replace the constraints (M19)-(M21) by the simpler Moovac-S-style version of the modulo decomposition, $t_i = y_i \cdot \lambda^\circ + m_i,\ \forall i \in L$.

Again with a time limit of τ, the solver is now instructed to minimise only the schedule length. Note that this is a "warm start" made possible by constraints (M21), meaning the feasible solution from the first step is still valid for the modified ILP, and the solver will work on improving it. Therefore, only two outcomes are possible after the solver returns the second time:

- Optimal. The solver *proved* that the current schedule's length T_{λ° is optimal, i.e. $T_{\lambda^\circ} = T^\star_{\lambda^\circ}$.
- Feasible. The current schedule has a length of T_{λ°, but the solver was unable to determine whether it is optimal within the time limit. We only know that $T_{\lambda^\circ} \geq T^\star_{\lambda^\circ}$.

The reasoning about the optimality of the overall solution $S = (\lambda^\circ, T_{\lambda^\circ})$ is straight-forward: We have $S = S^\star$ if and only if both steps yielded an optimal result according to their respective objective. If optimality can be proven in the first step, but not in the second step, we trivially know $\lambda^\circ = \lambda^\star$. It is possible that we just find a feasible interval, but determine the optimal schedule length for that interval. This situation is counted as an overall feasible solution.

Fallback schedule. Recall that we save a resource-constrained non-modulo schedule from the computation of $\lambda^\top$. This schedule serves as a fallback in the unfortunate case that all modulo scheduling attempts fail, and allows the compilation to continue after a guaranteed maximum time.

4.2 Bounded schedule length

Imposing an upper bound U on the schedule length is beneficial for practical modulo scheduling, as it reduces the overall solution space of the combinatorial problem, and helps the LP solver to give up earlier on fruitless branches in situations where the infeasibility of a partial solution is not obvious to the solver, causing it to try increasingly late start times for operations in the search. Additionally, such a bound is required to make the proposed linearisation of constraints (M19) in the Moovac-I formulation possible.

While we desire U to be as tight as possible, any chosen bound needs to be an overestimate, i.e. higher or equal to the optimal schedule length for a given λ, to enable the solver to find a feasible solution for the given λ (and not to rule that λ out incorrectly).

Eichenberger's bound. Eichenberger et al. [13] proposed and proved the following upper bound for the length of a modulo schedule, which we adapt here to our notation. Let $\Delta = \max_{(i \to j) \in E} \delta_{ij}$, i.e. the maximum number of time steps that two operations connected by a dependence edge must be started apart. We define

$$U_{Eb} = |O| \cdot (\Delta + \lambda^\top - 1)\,. \tag{5}$$

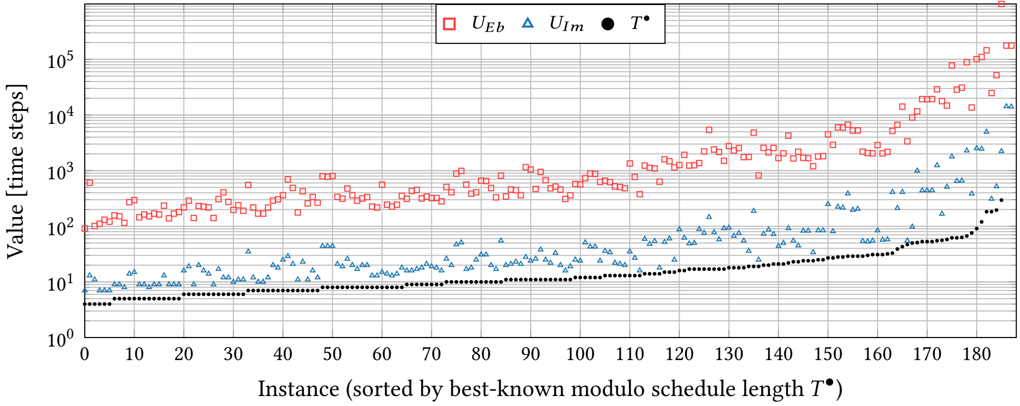


Fig. 6. Distribution of modulo schedule lengths, and their bounds. Note the logarithmic scale on the Y-axis.

Intuitively, each operation $i \in O$ requires at most Δ time steps before the next dependent operation can start, but may need to be deferred for up to $\lambda^\top - 1$ time steps[10] due to the modulo resource constraints.

Improved bound. Figure 6 shows the distribution of best-known schedule lengths, and corresponding bound values for U_{Eb}, for the test instances used in our experimental evaluation (Section 5.4). It is obvious that U_{Eb} overestimates the actual schedule length by roughly an order of magnitude. We propose two improvements to Eichenberger's bound that result in a safe upper bound U_{Im} that is much closer to the actual modulo schedule length (also shown in Figure 6).

Let $\Delta_i = \max_{j \in O:(i \to j) \in E} \delta_{ij}$ denote the maximum number of time steps that any successor j of i needs to be started after i. We define the improved bound as

$$U_{Im} = \underbrace{\sum_{i \in O} \Delta_i}_{\text{a) assume sequential schedule}} + \underbrace{\sum_{k \in R} \sum_{q=0}^{|L_k|-1} \left\lfloor \frac{q}{a_k} \right\rfloor}_{\text{b) account for shifts due to modulo resource conflicts}} . \quad (6)$$

This definition keeps the basic ideas in Eichenberger's bound of a) assuming that in the worst case, all operations need to be scheduled sequentially, and b) of accounting for the need to shift operations to a later start time due to the modulo resource conflicts.

Our improvement to a) is straight-forward: Instead of assuming $|O|$-many same-sized windows of time steps that are large enough to accomodate every operation in the MSP, we determine the maximum number of time steps individually for each operation.

We now quantify the worst-case number of modulo resource conflicts more precisely to derive a tighter estimate for b). First, unlimited operations never need to be shifted as, by definition, they never compete for resources. Next, recall the intuition of the modulo reservation table (Figure 3) for a resource type k with one available instance, and imagine it is successively filled with the

[10]More accurately, given a candidate interval λ, an operation $i \in L$ may be deferred for $\lambda - 1$ time steps, because at that point all possible congruence classes would have been considered. Choosing the upper bound instead results in a slightly looser bound, but makes the individual scheduling attempts better comparable as the numerical value of the bound remains constant.

operations $\{l_0, \ldots, l_{|L_k|-1}\} = L_k$ competing for that k-instance during scheduling[11]. Operation l_0 will not need to be shifted as it is the first operation to be assigned to the MRT. l_1 may need to be shifted for at most one time step if it conflicts with l_0. l_2 may need to be shifted for at most two time steps if it conflicts with l_0, and then with l_1, which was already shifted by one time step to resolve its own conflict with l_0. In the worst case, all operations in L_k are in initially in conflict for using the k-instance in the same congruence class. In general, an operation l_q will need to be shifted for at most q time steps, i.e. equal to the number of operations that are already assigned to the MRT.

This reasoning is easily extended to resource types k that provide $a_k > 1$ available instances. In that case, the required conflict-resolving shift amount increases only every a_k operations, which means that an operation l_q will need to be shifted at most $\left\lfloor \frac{q}{a_k} \right\rfloor$ time steps in the worst case.

Summing the maximum shift amounts for all operations over all limited resource types, we arrive at the expression in Eq. (6-b).

Note that we neither make any assumptions about the particular conflict-resolving shift amount an individual operation might require, nor about the order in which operations are assigned to the MRT, but rather estimate the maximum number of shifts to be expected for a set L_k as a whole.

For the running example (Figure 1), we compute $U_{Eb} = 12 \cdot (2 + 6 - 1) = 84$, and $U_{Im} = (11 \cdot 1 + 1 \cdot 2) + (0 + 1 + 2) = 16$.

5 EXPERIMENTAL EVALUATION

We now compare scheduler implementations based on the Moovac-S, Moovac-I and Eichenberger's formulations, and the Modulo SDC scheduler, on a large set of typical high-level synthesis loops, in terms of both the scheduling time and the quality of the resulting schedule lengths and minimal IIs.

5.1 Compiler context

We implemented all schedulers in the Nymble HLS compiler [18]. Nymble is based on the LLVM framework [22], version 3.3 and uses the framework's analyses and optimisations.

All function calls in the benchmark programs are inlined exhaustively. The resulting modules are optimised with LLVM's preset -O2, but without performing loop unrolling.

Intermediate representation. Nymble uses a hierarchical control-dataflow graph (CDFG) as its main intermediate representation, i.e. the compiler constructs a CDFG for each natural loop in the input program. Within such a graph, control information is translated to predicated dataflow, and nested loops are represented as special operations. This IR allows us to attempt to modulo schedule loops on all nesting levels and with arbitrary control structures, without the need to perform preparative transformations such as if-conversion.

We rely on LLVM's dependence analysis to discover intra- and inter-iteration memory dependences. These dependences are encoded as additional edges in the CDFG, and handled uniformly by the schedulers. Due to a technical limitation, we currently consider all of these backedges to express loop-carried dependences to the *immediately preceding* iteration, leading to conservatively larger IIs. However, we expect the impact on our results to be small, as in our compiler context, the dependence analysis could only determine exact backedge distances $\beta_{ij} > 1$ for less than 0.3 % of the backedges that were constructed.

The schedulers operated according to the resource limits in Table 3.

[11] Note that while the ILP-based schedulers discussed in this work do not operate in such a way internally, it is still a helpful conceptual model.

Table 3. Resource limits

Resource type	# available
Memory `Load/Store`	1 each
Nested loops	1
Integer `Div`/other	8/∞
FP `Add/Sub/Mul`/other	4/4/4/2 each

Table 4. Problem sizes

Metric	min.	med.	avg.	max.
# operations	19	54	125	2654
# res.-lim. ops.	0	4	12	221
# forward edges	28	81	245	6752
# backedges	2	4	42	1222

5.2 Reference schedulers

We implemented the Modulo SDC scheduler according to [5], but used a simpler height-based priority function. Our implementation of Eichenberger's formulation is based on the constraints (1), (2), (5) and (20) in [12]. In both cases, the objective is changed to minimise the schedule length.

The dataflow graphs constructed by Nymble contain a unique start operation that is required to be scheduled to time step 0. A corresponding constraint is added to all schedulers.

5.3 Test setup

We used Gurobi 8.0 as (I)LP solver for all schedulers.

The experiments were conducted on 2x12-core Intel Xeon E5-2680 v3 systems running at 2.8 GHz with 64 GiB RAM. The schedulers were allowed to use up to 8 threads and 16 GiB of memory per loop.

Each experiment was repeated three times to compensate for varying system load, and for each loop, we include in our evaluation the result of the "best" modulo scheduling attempt with regard to the smallest II, schedule length, and lastly, scheduling runtime. The scheduling runtime always includes the time to construct the respective linear programs via the Gurobi API. The generated schedules were verified by RTL simulation of the hardware accelerators generated by Nymble.

5.4 Test instances

The modulo scheduling test instances, i.e. loops/graphs, used in this evaluation originate from the HLS benchmark applications in the CHStone [16] and MachSuite [26] collections. We excluded `backprop`, `bfs/bulk`, `fft/transpose` and `nw` from MachSuite due to limitations in Nymble that prohibited the synthesis of these applications even without using the modulo schedulers evaluated here. Also, we removed `printf` statements inside the computational kernels of CHStone's `aes` and `jpeg` applications.

In total, we obtained 354 loops to modulo schedule. Due to the exhaustive inlining performed in Nymble, and the presence of small, idiomatic loops (e.g. array initialisations), we detected that 166 loops are identical or isomorphic to other loops.

The remaining **188 loops** comprise our set of test instances used in the rest of this evaluation. Table 4 characterises the sizes of the corresponding MSPs. The histogram in Figure 7 shows the distribution of the best-known intervals $\lambda^\bullet$ for the test instances, normalised to the range between 0 (= $\lambda^\perp$) and 1 (= $\lambda^\top$). In summary, the majority of loops in our test set have a best-known II equal or close to the lower bound of the search space. To this end, the single-II approaches in this evaluation traverse the II search space in the traditional ascending order. In order to be able to complete each individual experiment within 24 hours, we limit the search to at most 23 candidate IIs for the single-II approaches, i.e. we attempt to schedule for $\lambda^\perp, \lambda^\perp + 1, \lambda^\perp + 2, \ldots, \lambda^\perp + 22$. All loops in our

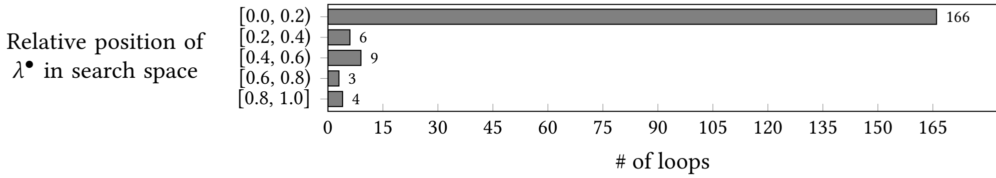


Fig. 7. Histogram of the number of loops according to the relative position of their $\lambda^\bullet$ within the II search space, i.e. $0 = \lambda^\perp$ and $1 = \lambda^\top$.

evaluation have $\min(\lambda^\top, \lambda^\bullet) \leq \lambda^\perp + 22$, i.e. this attempt limit did not preclude any of the schedulers from finding a feasible solution.

5.5 Comparison of approaches, time limits and bounds

In our first experiment, we schedule the test instances with the single-II Moovac formulation (denoted by *Mv-S*), the Moovac formulation with integrated II minimisation (*Mv-I*), Eichenberger's formulation (*EB*), and the Modulo SDC algorithm (*MSDC*).

We investigate time limits (Section 4.1) τ of 1, 5, 15 and 60 minutes, and impose either no upper bound on the schedule length (denoted by ∞), or use the bounds U_{Eb} or U_{Im} from Section 4.2.

While the time limit and schedule length bound seem independent on first sight, they both limit the computational effort that the solver spends on a particular scheduling attempt. In the case of the schedule length bound, this limit works indirectly by reducing the branch-and-bound solution space.

To this end, we present the results of modulo scheduling with the different time limits and upper bounds together, both in terms of scheduler runtimes (Table 5) and schedule quality (Table 6). Our methodology here is twofold: On the one hand, we provide accumulated runtimes and quality measures for the computed initiation intervals across all instances as intuitive metrics for the performance of a given configuration. However, this presentation disregards that all instances are in fact *independent* of each other. To this end, we additionally count the number of instances for which a configuration achieves a particular result, e.g. to schedule an instance between 10 and 100 seconds, or to prove optimality for a solution. This allows us to assess to which degree a given configuration is practical on our representative benchmark set, and highlights the presence of outlier instances (i.e. with exceptionally long runtimes or low quality schedules) as well as their influence to the accumulated metrics. Note that outliers have to be expected for every exact modulo scheduling approach due to the NP-hard nature of the underlying problem. However, if their number is small, this does not automatically impair the scheduler's practicability. Additionally, a benefit of the ILP method is that a quality measure (i.e. the branch-and-bound method's "gap" value [19]) is available for any feasible solution, which allows the practitioner to decide to try another scheduler (exact or heuristic) for the same problem.

We introduce the notion of the *best-known* interval $\lambda^\bullet$ and tuple of interval and schedule length $S^\bullet$ to indicate the best solution found across all experiments conducted in this work, i.e. including all approaches and configurations, as a reference point for loops for which the optimal solution is not known.

The **Moovac**-based schedulers outperform the other approaches in this experiment: The overall fastest configuration is *Mv-I-*U_{Im}-1min with an accumulated runtime of 15.7 minutes for all 188 test instances. At first glance, *Mv-S-*U_{Im}, which finds the highest-quality solutions overall, seemingly

has a significant advantage concerning the result quality over *Mv-I-*U_{Im}, but note that the sum of II differences (column "$\lambda^\circ - \lambda^\bullet : \Sigma$") is concentrated on only three loops (column "$\lambda^\circ - \lambda^\bullet$: #"). For all other loops, *Mv-I-*U_{Im} delivers results on par with its single-II variant (columns under "# loops with...").

Mv-S-U_{Im} and *Mv-I-*U_{Im} achieve the best trade-off between scheduler runtime and result quality already with the 1 minute time limit. The runtimes increase with the time budget, but the result quality is only improved marginally. In the 60 minute configurations, the solution space covered by the Moovac-based schedulers for some instances becomes too large to fit in the 16 GiB allocated to the experiments (column "OOM").

Mv-S clearly benefits, in terms of runtime and quality, from the presence of an upper bound for the schedule length, and from the tighter estimate that U_{Im} gives compared to U_{Eb}. This effect is even more pronounced for *Mv-I* , because the value of the bound directly affects the complexity of the Moovac-I-ILP.

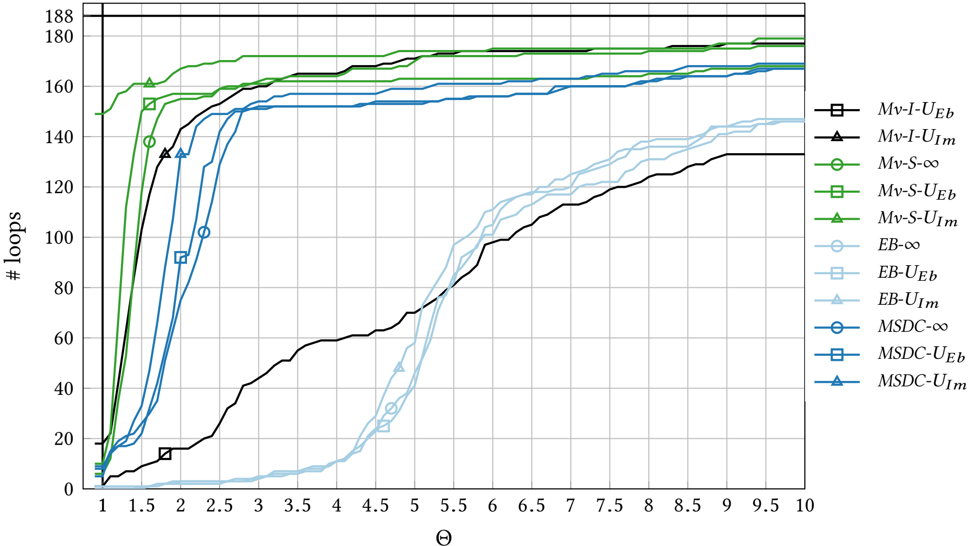
Being an exact scheduler as well, **Eichenberger's** formulation achieves optimal results for almost as many loops as the Moovac-based approaches. However, especially with the smallest time budget, it is unable to find modulo schedules for several large instances (column "no *S*"), and runs out of memory for one loop. Given more time, the solution quality improves, but unfortunately four more loops cannot be scheduled within the 16 GiB memory limit. The accumulated runtimes and solution quality is mostly unaffected by the choice of schedule length bound.

As expected, the **ModuloSDC** algorithm is quite fast, but lacks the capability to prove optimality of its solutions, and misses the the best-known IIs by at least 77, spread over 20 instances. Starting with the 5 minute configuration, imposing an upper bound on the schedule length effectively serves as a third measure (besides time limit and backtracking budget) to give up on fruitless scheduling attempts, and limits the overall runtime to roughly 90 minutes. The quality metrics remain unchanged over the four different time limits. Curiously, without a bound, one more loop can be scheduled, which alone contributes an II difference of 77 to *MSDC-*U_{Eb} and -U_{Im}. The length of the schedule found by *MSDC-*∞ for this particular instance does not come near the values of the bounds, though.

The maximum ILP **construction time** per scheduling attempt is below 10 seconds for the exact approaches, and turned out to be negligible compared to the runtime of the solver's branch-and-bound algorithm. This excludes loops for which a scheduling attempt ran out of memory, because we did not record the precise point in time when the memory limit was hit. We also did not record a separate construction time for *MSDC*, as the linear program is constantly changed over the course of the algorithm.

The **performance profile** [10] in Figure 8 shows a different view on the dataset for the 5 minute experiment, as it relates the runtimes of different configurations for each individual loop: for every configuration α and loop x, a ratio $\rho_{\alpha,x}$ of the runtime of α on x divided by the fastest runtime for x is computed. In the plot, we count the number of loops that have $\rho_{\alpha,x} \leq \Theta$ for a given configuration α, i.e. the number of loops whose runtime with α is at most Θ times worse than the best-known runtime for x. The performance profile can be interpreted as the probability[12] that a configuration is able to modulo schedule a loop within a certain performance envelope. The higher the curve the better. For example, the data point marked with dark blue square on the *MSDC-*U_{Eb} plot signifies that the scheduler runtime for 92 loops (= $\frac{92}{188} \approx 49\%$) was at most twice as long as the fastest runtime across all configurations in the performance profile. The intersection of the plots with the

[12]By dividing the loop counts by the total number of loops.



Approach / Bound	*Mv-S-∞*	*Mv-S-U_{Eb}*	*Mv-S-U_{Im}*	*Mv-I-U_{Eb}*	*Mv-I-U_{Im}*	*EB-∞*	*EB-U_{Eb}*	*EB-U_{Im}*	*MSDC-∞*	*MSDC-U_{Eb}*	*MSDC-U_{Im}*
# loops at $\Theta = 1$	6	10	149	1	18	0	0	1	8	5	9

Fig. 8. Performance profile of the scheduling times (5 minute time limit), showing the number of loops for which a combination of an approach and a bound resulted in a scheduling time that is at most Θ times slower than the fastest scheduling time for each individual loop. The table shows the values for the special case $\Theta = 1$, i.e. the number of loops for which a configuration defined the fastest scheduler runtime.

"$\Theta = 1$"-line shows the number of loops for which a configuration set the fastest runtime. This data is repeated in the table below the performance profile.

We observe that *Mv-S-U_{Im}* is the fastest configuration to schedule 149 of the 188 loops ($\approx$ 79%), followed by *Mv-I-U_{Im}* with 18 fastest runtimes. In general, the plots for the Moovac-based configurations (excluding *Mv-S-U_{Eb}*) and *MSDC* rise quickly, meaning that they are close to the fastest runtime for the majority of instances. The slope of the plots for *EB* indicates that for roughly 75 % of the loops, these configurations require at least 5x longer than the respective fastest configuration.

The performance profile also hints at a key **difference between *Mv-S* and *Mv-I***. The single-II variant is often faster for an individual loop due to the simpler ILP formulation and the location of the first feasible interval in the II search space (cf. Fig. 7). However, the practical advantage of *Mv-I* is that its runtime is capped at $2 \cdot \tau$ by design, which is reflected in the much faster accumulated runtimes shown in Table 5. As discussed above, the quality metrics in Table 6 show that this novel way of approaching the modulo scheduling problem comes with almost no loss in solution quality, with *Mv-I-U_{Im}* missing the best-known interval $\lambda^{\bullet}$ for at most two additional instances compared to *Mv-S-U_{Im}*.

5.6 FPGA implementation

We used Nymble [18] to generate accelerator modules that are pipelined according to the schedules computed in the experiments with the U_{Im}-bound and 5 minute time limit. The non-loop parts of the applications, as well as loops for which a particular scheduler could not compute a feasible

Table 5. Scheduling times for combinations of approaches, time limits and bounds

Approach / Bound		Loops classified by scheduler runtime (188 loops)										
		< 10 s		10-100 s		100-1k s		1k-10k s		≥ 10k s		all
		#	Σ [min]	#	Σ [min]	#	Σ [min]	#	Σ [min]	#	Σ [min]	Σ [min]
Time limit: 1 minute												
Mv-S	∞	168	0.7	8	5.9	12	92.0	-	-	-	-	98.5
Mv-S	U_{Eb}	173	0.5	5	2.7	10	82.9	-	-	-	-	86.1
Mv-S	U_{Im}	174	0.4	5	2.8	9	71.3	-	-	-	-	74.5
Mv-I	U_{Eb}	155	0.7	32	26.1	1	2.0	-	-	-	-	28.7
Mv-I	U_{Im}	171	0.5	15	11.2	2	4.0	-	-	-	-	15.7
EB	∞	165	1.2	10	6.1	8	60.0	5	112.6	-	-	179.8
EB	U_{Eb}	167	1.5	7	3.0	9	55.8	5	112.7	-	-	173.0
EB	U_{Im}	167	1.3	9	5.5	7	52.1	5	112.7	-	-	171.7
MSDC	∞	172	0.7	7	4.6	8	46.8	1	17.1	-	-	69.2
MSDC	U_{Eb}	172	1.0	8	5.5	7	44.1	1	16.8	-	-	67.5
MSDC	U_{Im}	172	1.0	8	5.4	7	44.4	1	16.9	-	-	67.8
Time limit: 5 minutes												
Mv-S	∞	168	0.6	3	0.9	8	55.2	9	378.3	-	-	435.0
Mv-S	U_{Eb}	173	0.5	3	0.7	4	32.5	8	339.5	-	-	373.1
Mv-S	U_{Im}	174	0.4	3	0.8	3	25.1	8	292.8	-	-	319.1
Mv-I	U_{Eb}	155	0.7	11	4.2	22	117.1	-	-	-	-	122.0
Mv-I	U_{Im}	172	0.7	5	1.8	11	70.3	-	-	-	-	72.7
EB	∞	165	1.2	10	6.4	5	39.8	8	520.1	-	-	567.6
EB	U_{Eb}	167	1.5	8	4.4	5	45.6	8	538.0	-	-	589.4
EB	U_{Im}	167	1.3	9	5.0	4	34.8	8	552.0	-	-	593.2
MSDC	∞	172	0.7	7	4.6	6	23.8	3	94.9	-	-	124.1
MSDC	U_{Eb}	172	1.1	8	5.6	6	30.5	2	49.0	-	-	86.1
MSDC	U_{Im}	172	1.1	8	5.6	6	30.9	2	48.1	-	-	85.5
Time limit: 15 minutes												
Mv-S	∞	168	0.7	3	1.0	5	62.4	11	952.1	1	225.0	1241.1
Mv-S	U_{Eb}	173	0.5	3	0.7	3	37.5	8	797.1	1	225.0	1060.8
Mv-S	U_{Im}	174	0.4	3	0.8	2	30.0	8	655.0	1	225.0	911.2
Mv-I	U_{Eb}	154	0.5	12	4.5	20	268.2	2	49.0	-	-	322.2
Mv-I	U_{Im}	171	0.5	6	2.0	8	120.3	3	90.1	-	-	212.9
EB	∞	165	1.2	10	6.8	3	38.7	6	388.4	4	1058.5	1493.5
EB	U_{Eb}	167	1.5	9	6.0	1	15.7	7	436.6	4	1064.3	1524.1
EB	U_{Im}	167	1.3	9	5.1	2	22.8	6	407.5	4	1074.4	1511.2
MSDC	∞	172	0.7	7	4.8	6	24.1	3	180.7	-	-	210.3
MSDC	U_{Eb}	170	0.8	10	5.9	6	31.3	2	54.1	-	-	92.1
MSDC	U_{Im}	170	0.7	10	5.8	6	31.2	2	52.8	-	-	90.6
Time limit: 60 minutes												
Mv-S	∞	168	0.7	3	0.9	1	2.4	6	422.8	10	5614.8	6041.7
Mv-S	U_{Eb}	173	0.5	3	0.7	1	7.3	2	120.0	9	4996.2	5124.7
Mv-S	U_{Im}	174	0.4	3	0.8	-	-	4	250.0	7	4380.1	4631.3
Mv-I	U_{Eb}	155	0.7	11	4.3	4	27.2	18	1140.2	-	-	1172.4
Mv-I	U_{Im}	172	0.7	5	1.8	-	-	11	720.9	-	-	723.4
EB	∞	165	1.2	10	6.7	1	8.1	6	407.1	6	4877.1	5300.2
EB	U_{Eb}	167	1.5	9	5.7	-	-	6	508.8	6	4886.9	5403.0
EB	U_{Im}	167	1.4	9	5.3	1	7.4	5	373.4	6	4753.0	5140.4
MSDC	∞	172	0.8	7	4.9	6	26.1	2	52.8	1	420.4	505.0
MSDC	U_{Eb}	171	0.9	9	5.8	6	32.7	2	49.6	-	-	89.0
MSDC	U_{Im}	172	1.1	8	5.8	6	32.3	2	49.4	-	-	88.6

The loops are classified into time brackets according to the scheduler runtime with a given approach. Columns "#" show the number of loops that fell into a particular time bracket. Columns "Σ" show the accumulated scheduling time (in **minutes**) for the loops within a time bracket. Loops that caused the scheduler to run out of memory (cf. Table 6) are accounted for with the maximum available time to them, i.e. $(\min(\lambda^{\top}, \lambda^{\perp} + 22) - \lambda^{\perp} + 1) \cdot \tau$ for the single-II approaches, and τ for *Mv-I*.

Table 6. Schedule quality for combinations of approaches, time limits and bounds

Approach / Bound		# loops with ...								$\lambda^\circ - \lambda^\bullet$	
		$\lambda^\star\square$	$\lambda^\circ = \lambda^\star$	$\lambda^\circ = \lambda^\bullet$	$S^\star\square$	$S = S^\star$	$S = S^\bullet$	no S	OOM	Σ	#
Time limit: 1 minute											
Mv-S	∞	173	183	184	171	182	183	2	-	5	2
Mv-S	U_{Eb}	178	183	184	176	182	183	2	-	4	2
Mv-S	U_{Im}	179	184	185	177	182	182	2	-	2	1
Mv-I	U_{Eb}	167	176	177	166	176	176	11	-	328	9
Mv-I	U_{Im}	179	182	183	177	181	181	5	-	52	3
EB	∞	173	176	176	173	176	176	9	1	120	10
EB	U_{Eb}	174	176	176	174	176	176	9	1	115	10
EB	U_{Im}	175	177	177	174	176	176	9	1	114	9
MSDC	∞	146	166	166	-	152	152	3	-	77	20
MSDC	U_{Eb}	143	163	163	-	143	143	4	-	141	23
MSDC	U_{Im}	143	163	163	-	143	143	4	-	141	23
Time limit: 5 minutes											
Mv-S	∞	174	183	184	172	182	183	2	-	3	2
Mv-S	U_{Eb}	179	183	184	177	183	184	2	-	3	2
Mv-S	U_{Im}	179	184	185	177	183	183	2	-	1	1
Mv-I	U_{Eb}	170	177	178	168	177	177	9	-	246	8
Mv-I	U_{Im}	179	182	183	177	182	183	4	-	16	3
EB	∞	176	181	181	176	180	180	1	5	34	5
EB	U_{Eb}	176	181	181	176	181	181	1	5	36	5
EB	U_{Im}	177	181	181	177	181	181	1	5	40	5
MSDC	∞	146	166	166	-	152	152	3	-	77	20
MSDC	U_{Eb}	143	163	163	-	143	143	4	-	141	23
MSDC	U_{Im}	143	163	163	-	143	143	4	-	141	23
Time limit: 15 minutes											
Mv-S	∞	174	183	184	172	183	184	2	-	2	2
Mv-S	U_{Eb}	179	183	184	177	183	184	2	-	3	2
Mv-S	U_{Im}	179	184	185	177	183	183	2	-	1	1
Mv-I	U_{Eb}	172	179	180	170	179	180	6	-	239	6
Mv-I	U_{Im}	179	182	183	177	182	183	4	-	12	3
EB	∞	177	181	183	177	181	182	-	5	31	3
EB	U_{Eb}	176	181	183	176	181	183	-	5	31	3
EB	U_{Im}	177	181	182	177	181	182	-	5	32	4
MSDC	∞	146	166	166	-	152	152	3	-	77	20
MSDC	U_{Eb}	143	163	163	-	143	143	4	-	141	23
MSDC	U_{Im}	143	163	163	-	143	143	4	-	141	23
Time limit: 60 minutes											
Mv-S	∞	174	182	182	172	182	182	2	3	30	4
Mv-S	U_{Eb}	179	182	182	177	182	182	2	3	30	4
Mv-S	U_{Im}	181	183	184	179	182	182	2	2	28	2
Mv-I	U_{Eb}	173	179	180	171	179	180	5	1	238	6
Mv-I	U_{Im}	179	182	182	177	182	182	3	2	30	4
EB	∞	178	181	183	178	181	183	-	5	31	3
EB	U_{Eb}	178	181	183	178	181	183	-	5	31	3
EB	U_{Im}	179	181	183	179	181	183	-	5	31	3
MSDC	∞	146	166	166	-	152	152	3	-	77	20
MSDC	U_{Eb}	143	163	163	-	143	143	4	-	141	23
MSDC	U_{Im}	143	163	163	-	143	143	4	-	141	23

The 188 loops are counted according to properties of the schedules (see Table 7) computed by a given approach.

Table 7. Quality metrics used in Table 6

Column	Description
$\lambda^{\star}\square$	counts loops where the II is *proven* to be optimal by the approach, as discussed in Section 4.1
$\lambda^{\circ} = \lambda^{\star}$	counts loops where the II is equal to the *optimal* II
$\lambda^{\circ} = \lambda^{\bullet}$	counts loops where the II is equal to the *best-known* II across all experiments, including loops where $\lambda^{\bullet} = \lambda^{\star}$
$S^{\star}\square$	counts loops where the tuple $S = (\lambda^{\circ}, T_{\lambda^{\circ}})$ is *proven* to be optimal by the approach, as discussed in Section 4.1
$S = S^{\star}$	counts loops where the tuple $S = (\lambda^{\circ}, T_{\lambda^{\circ}})$ is equal to the *optimal* solution
$S = S^{\bullet}$	counts loops where the tuple $S = (\lambda^{\circ}, T_{\lambda^{\circ}})$ is equal to the *best-known* solution across all experiments, including loops where $S^{\bullet} = S^{\star}$
no S	counts loops for which no modulo schedule could be computed within the time and attempt limits
OOM	counts loops for which no modulo schedule could be computed within the memory limit
$\lambda^{\circ} - \lambda^{\bullet}$	considers the differences between each loop's actual and best-known II. We set $\lambda^{\circ} = \lambda^{\top}$ in case no modulo schedule could be computed, and $\lambda^{\bullet} = \lambda^{\top}$ in case no modulo schedule is known (2 loops)
Σ	accumulates the differences
#	shows the number of loops that have $\lambda^{\circ} > \lambda^{\bullet}$

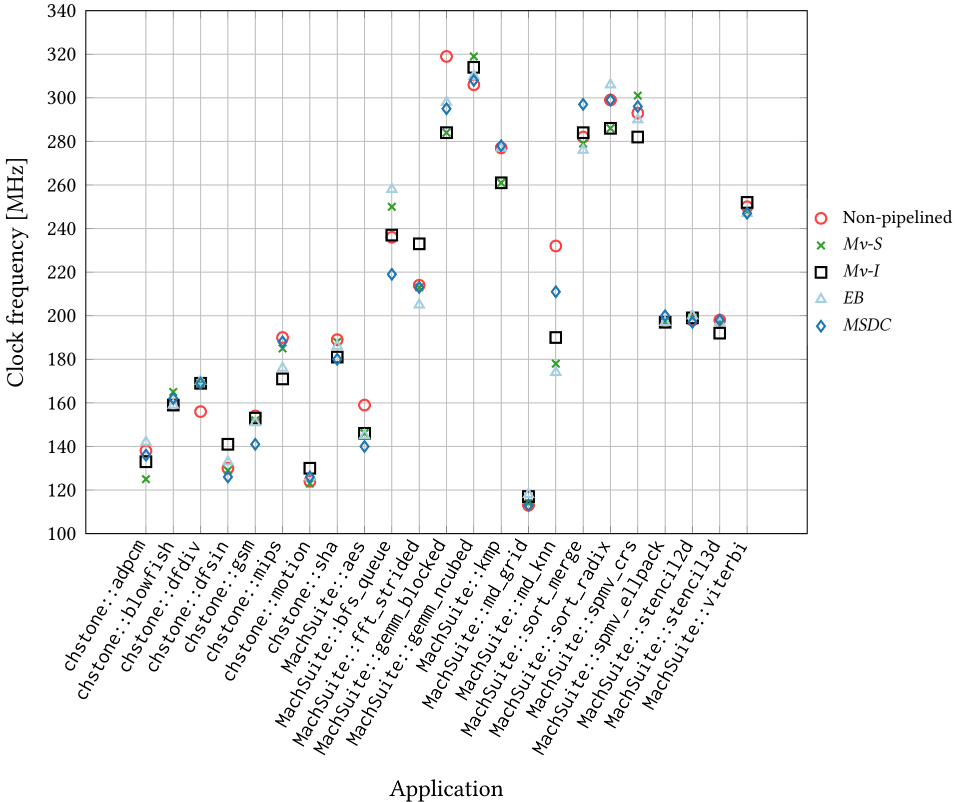


Fig. 9. Maximum clock frequencies on Virtex-7 after HLS and place & route for the different schedulers (U_{Im}-5min configurations). Note that the Y-axis starts at 100 MHz.

modulo schedule, are not pipelined and rely on the non-modulo schedules computed by the heuristic fallback scheduler.

TaPaSCo [20] was employed to perform an out-of-context evaluation of these accelerators using Vivado 2018.2 targeting a Virtex-7 `xc7vx690tffg1761-2` device. The target frequency was set to 200 MHz for the CHStone applications, `MachSuite::aes` and `MachSuite::md_grid`, and to 320 MHz for the remaining MachSuite applications.

Figure 9 shows the maximum clock frequencies when using the modulo schedulers, in comparison to a entirely non-pipelined accelerator using only the fallback non-modulo schedule for all loops. Unfortunately, `chstone::aes` and `chstone::jpeg` could not be implemented on the target device due to routing congestion in the Nymble-generated microarchitecture for any scheduler configuration, and are therefore excluded in the plot.

Overall, the frequency variations when using the different schedulers are moderate, and none of the approaches (especially not the non-pipelined one) dominates this experiment. However, note that the scheduling step is so early in the HLS flow that it cannot directly influence the later decisions made by the logic synthesis tool that ultimately determine the design's cycle time, so this result is not unexpected.

6 CONCLUSION AND OUTLOOK

In this paper, we proposed Moovac, an exact bi-criteria formulation of the modulo scheduling problem (MSP) that integrates the actual II minimisation and can be solved optimally by a standard ILP solver.

An extensive experimental study in the context of a high-level synthesis compiler showed that combining the Moovac formulation with a short time limit of 1 or 5 minutes per candidate II, and an improved upper bound for the schedule length, results in a practically usable modulo scheduling approach that finds optimal IIs in over 95 % of the test instances, and near-optimal IIs otherwise.

Exact modulo scheduling is justified by the clearly higher quality of the solutions compared to a state-of-the-art heuristic approach. In the Moovac formulation, modelling the central modulo-resource-constraints with overlap variables leads to overall shorter solver runtimes compared to a prior, well-tuned ILP formulation. Often, our approach is even faster than the heuristic SDC-based scheduler.

The focus of this work was to empirically study the practicability of a novel and exact formulation of the MSP. In the future, we plan to investigate how the model can be improved based on research into the more general class of cyclic scheduling problems in operations research literature.

MSP instances in the HLS context appear to comprise larger and denser dependence graphs than those that occur in VLIW compilers. This could by caused, for example, by the many resource-unconstrained operations, and the additional edges needed to model operator chaining. We plan to devise a formulation-independent problem reduction algorithm that abstracts non-critical subgraphs away in order to lower some of the HLS-specific complexity before passing the instance to the actual modulo scheduler.

Additionally, we would like to devise a generator for MSP instances in order to determine which problem sizes are still solvable in reasonable amounts of time, as well as to investigate which particular problem structures cause long runtimes for the various schedulers.

Lastly, the overlap variable-based combined modulo resource constraints in the Moovac formulation lend themselves to be selectively disabled. This can be exploited to perform exact *predicate-aware* modulo scheduling, i.e., oversubscribing resource units with operations that are mutually-exclusive at runtime.

ACKNOWLEDGMENTS

The experiments for this research were conducted on the Lichtenberg high performance computing cluster of TU Darmstadt. The authors would like to thank Xilinx, Inc. for supporting their work by hardware and software donations.

REFERENCES

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

[2] Erik R. Altman, Ramaswamy Govindarajan, and Guang R. Gao. 1995. Scheduling and Mapping: Software Pipelining in the Presence of Structural Hazards. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI), La Jolla, California, USA, June 18-21, 1995*, David W. Wall (Ed.). ACM, 139–150. https://doi.org/10.1145/207110.207128

[3] Maria Ayala and Christian Artigues. 2010. *On integer linear programming formulations for the resource-constrained modulo scheduling problem.* Technical Report LAAS no. 10393. Archive ouverte HAL. https://hal.archives-ouvertes.fr/hal-00538821

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

[5] Andrew Canis, Stephen Dean Brown, and Jason Helge Anderson. 2014. Modulo SDC scheduling with recurrence minimization in high-level synthesis. In *24th International Conference on Field Programmable Logic and Applications, FPL 2014, Munich, Germany, 2-4 September, 2014*. IEEE, 1–8. https://doi.org/10.1109/FPL.2014.6927490

[6] Josep M. Codina, Josep Llosa, and Antonio González. 2002. A comparative study of modulo scheduling techniques. In *Proceedings of the 16th international conference on Supercomputing, ICS 2002, New York City, NY, USA, June 22-26, 2002*, Kemal Ebcioglu, Keshav Pingali, and Alex Nicolau (Eds.). ACM, 97–106. https://doi.org/10.1145/514191.514208

[7] Jason Cong and Zhiru Zhang. 2006. An efficient and versatile scheduling algorithm based on SDC formulation. In *Proceedings of the 43rd Design Automation Conference, DAC 2006, San Francisco, CA, USA, July 24-28, 2006*, Ellen Sentovich (Ed.). ACM, 433–438. https://doi.org/10.1145/1146909.1147025

[8] Benoît Dupont de Dinechin. 1994. *Simplex Scheduling: More than Lifetime-Sensitive Instruction Scheduling.* Technical Report PRISM 1994.22.

[9] B. D. De Dinechin. 2007. Time-Indexed Formulations and a Large Neighborhood Search for the Resource-Constrained Modulo Scheduling Problem. In *In proceedings of the 3rd Multidisciplinary International Conference on Scheduling : Theory and Applications (MISTA 2007), 28 -31 August 2007, Paris, France*, P. Baptiste, G. Kendall, A. Munier-Kordon, and F. Sourd (Eds.). 144–151.

[10] Elizabeth D. Dolan and Jorge J. Moré. 2002. Benchmarking optimization software with performance profiles. *Math. Program.* 91, 2 (2002), 201–213. https://doi.org/10.1007/s101070100263

[11] Łukasz Domagała. 2012. *Application of CLP to instruction modulo scheduling for VLIW processors.* Ph.D. Dissertation. Silesian University of Technology.

[12] Alexandre E. Eichenberger and Edward S. Davidson. 1997. Efficient Formulation for Optimal Modulo Schedulers. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI), Las Vegas, Nevada, June 15-18, 1997*, Marina C. Chen, Ron K. Cytron, and A. Michael Berman (Eds.). ACM, 194–205. https://doi.org/10.1145/258915.258933

[13] Alexandre E. Eichenberger, Edward S. Davidson, and Santosh G. Abraham. 1995. Optimum Modulo Schedules for Minimum Register Requirements. In *Proceedings of the 9th international conference on Supercomputing, ICS 1995, Barcelona, Spain, July 3-7, 1995*, Mateo Valero (Ed.). ACM, 31–40. https://doi.org/10.1145/224538.224542

[14] Alexandre E. Eichenberger, Edward S. Davidson, and Santosh G. Abraham. 2014. Author retrospective for optimum modulo schedules for minimum register requirements. In *ACM International Conference on Supercomputing 25th Anniversary Volume*, Utpal Banerjee (Ed.). ACM, 35–36. https://doi.org/10.1145/2591635.2591653

[15] Dirk Fimmel and Jan Müller. 2002. Optimal Software Pipelining with Rational Initiation Interval. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA '02, June 24 - 27, 2002, Las Vegas, Nevada, USA, Volume 2*, Hamid R. Arabnia (Ed.). CSREA Press, 638–643.

[16] 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. *JIP* 17 (2009), 242–254. https://doi.org/10.2197/ipsjjip.17.242

[17] Richard A. Huff. 1993. Lifetime-Sensitive Modulo Scheduling. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, Robert Cartwright (Ed.). ACM, 258–267. https://doi.org/10.1145/155090.155115

[18] Jens Huthmann, Björn Liebig, Julian Oppermann, and Andreas Koch. 2013. Hardware/software co-compilation with the Nymble system. In *2013 8th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), Darmstadt, Germany, July 10-12, 2013*. IEEE, 1–8. https://doi.org/10.1109/ReCoSoC.2013.6581538

[19] Ed Klotz and Alexandra M. Newman. 2013. Practical guidelines for solving difficult mixed integer linear programs. *Surveys in Operations Research and Management Science* 18, 1 (2013), 18 – 32. https://doi.org/10.1016/j.sorms.2012.12.001

[20] Jens Korinth, David de la Chevallerie, and Andreas Koch. 2015. An Open-Source Tool Flow for the Composition of Reconfigurable Hardware Thread Pool Architectures. In *23rd IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2015, Vancouver, BC, Canada, May 2-6, 2015*. IEEE Computer Society, 195–198. https://doi.org/10.1109/FCCM.2015.22

[21] Monica S. Lam. 1988. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*, Richard L. Wexelblat (Ed.). ACM, 318–328. https://doi.org/10.1145/53990.54022

[22] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*. IEEE Computer Society, 75–88. https://doi.org/10.1109/CGO.2004.1281665

[23] Josep Llosa, Eduard Ayguadé, Antonio González, Mateo Valero, and Jason Eckhardt. 2001. Lifetime-Sensitive Modulo Scheduling in a Production Environment. *IEEE Trans. Computers* 50, 3 (2001), 234–249. https://doi.org/10.1109/12.910814

[24] Julian Oppermann, Andreas Koch, Melanie Reuter-Oppermann, and Oliver Sinnen. 2016. ILP-based modulo scheduling for high-level synthesis. In *2016 International Conference on Compilers, Architectures and Synthesis for Embedded Systems, CASES 2016, Pittsburgh, Pennsylvania, USA, October 1-7, 2016*. ACM, 1:1–1:10. https://doi.org/10.1145/2968455.2968512

[25] B. Ramakrishna Rau. 1996. Iterative Modulo Scheduling. *International Journal of Parallel Programming* 24, 1 (1996), 3–65.

[26] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David M. Brooks. 2014. MachSuite: Benchmarks for accelerator design and customized architectures. In *2014 IEEE International Symposium on Workload Characterization, IISWC 2014, Raleigh, NC, USA, October 26-28, 2014*. IEEE Computer Society, 110–119. https://doi.org/10.1109/IISWC.2014.6983050

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

[28] Sarad Venugopalan and Oliver Sinnen. 2015. ILP Formulations for Optimal Task Scheduling with Communication Delays on Parallel Systems. *IEEE Trans. Parallel Distrib. Syst.* 26, 1 (2015), 142–151. https://doi.org/10.1109/TPDS.2014.2308175

[29] Zhiru Zhang and Bin Liu. 2013. SDC-based modulo scheduling for pipeline synthesis. In *The IEEE/ACM International Conference on Computer-Aided Design, ICCAD'13, San Jose, CA, USA, November 18-21, 2013*, Jörg Henkel (Ed.). IEEE, 211–218. https://doi.org/10.1109/ICCAD.2013.6691121

Received February 2018; revised November 2018; accepted February 2019