ILP-based Modulo Scheduling for High-level Synthesis

Julian Oppermann[1] Andreas Koch[1] Melanie Reuter-Oppermann[2] Oliver Sinnen[3]
[1]Embedded Systems and Applications Group, Technische Universität Darmstadt, Germany
[2]Discrete Optimization and Logistics Group, Karlsruhe Institute of Technology, Germany
[3]Parallel and Reconfigurable Computing Group, University of Auckland, New Zealand
{oppermann, koch}@esa.tu-darmstadt.de, melanie.reuter@kit.edu, o.sinnen@auckland.ac.nz

ABSTRACT

In high-level synthesis, loop pipelining is a technique to improve the throughput and utilisation of hardware datapaths by starting new loop iterations after a fixed amount of time, called the initiation interval (II), allowing to overlap subsequent iterations. The problem is to find the smallest II and corresponding operation schedule that fulfils all data dependencies and resource constraints, both of which are usually found by *modulo scheduling*.

We propose Moovac[1], a novel integer linear program (ILP) formulation of the modulo scheduling problem based on overlap variables to model exact resource constraints. Given enough time, Moovac will find a mimimum-II solution. This is in contrast to Canis' state-of-the-art Modulo SDC approach, which requires heuristic simplifications of the resource constraints. Moovac can thus be used as a reference to evaluate heuristics, or in a time-limited mode as a heuristic itself to provide a best-so-far solution.

We schedule kernels from the CHStone and MachSuite benchmarks for loop pipelining with Moovac, Modulo SDC and a prior exact formulation by Eichenberger.

Moovac has competitive performance in its time-limited mode, and delivers better results faster than the Modulo SDC scheduler for some loops. Often its structure leads to quicker solution times than Eichenberger's formulation.

Using the Moovac-computed optimal solutions as a reference, we can confirm that the Modulo SDC heuristic is indeed capable of finding optimal or near-optimal solutions for the majority of small- to medium-sized loops. However, for larger loops the two algorithms begin to diverge, with Moovac often being significantly faster to prove the infeasibility of a candidate II. This can be exploited by running both schedulers synergistically, leading to a quicker convergence to the final II.

CCS Concepts

•**Hardware** → **Operations scheduling;** *Reconfigurable logic and FPGAs;*

CASES '16, October 01-07, 2016, Pittsburgh, PA, USA

DOI: http://dx.doi.org/10.1145/2968455.2968512

1. INTRODUCTION

To achieve compute performance, the FPGA-based reconfigurable computing generally relies on spatial parallelism, while conventional CPUs still emphasise higher clock rates. A high-level synthesis (HLS) system that creates FPGA-based accelerators from sequential languages such as C must exploit all available sources of parallelism in order to achieve a meaningful speed-up compared to the execution on a software-programmable processor at a higher clock rate. One such source of parallelism is *loop pipelining*: New loop iterations are started after a fixed number of time steps, called the *initiation interval (II)*. This can result in a partially overlapping execution of subsequent loop iterations.

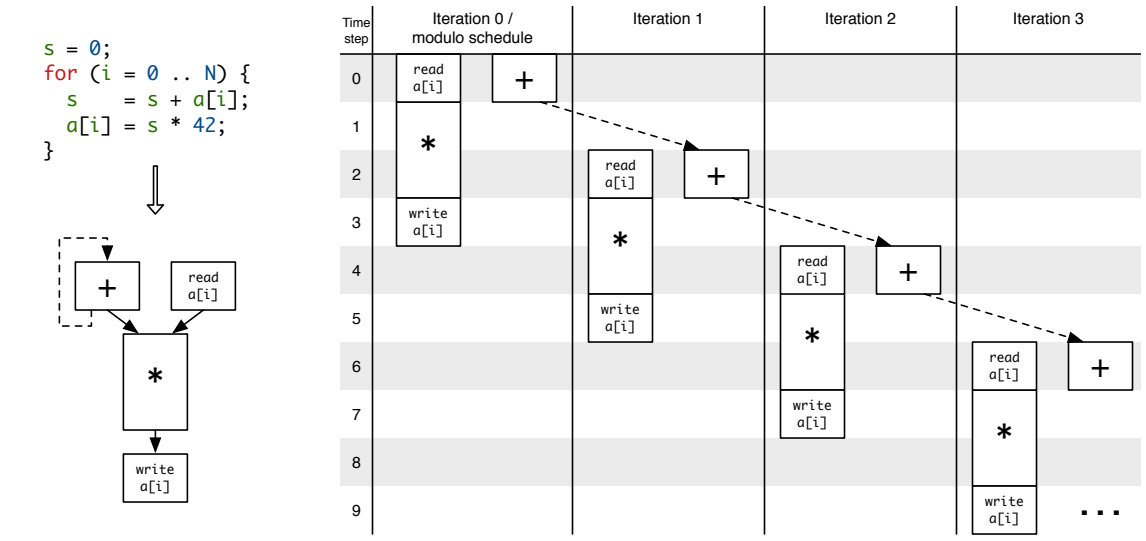
Let SL be the latency of the datapath representing the loop body. Executing n iterations of the loop sequentially then takes $n{\cdot}\mathrm{SL}$ time steps. Assume the loop's inter-iteration dependencies allow it to be executed with an initiation interval $\mathrm{II} < \mathrm{SL}$, then executing n iterations will require only $(n-1)\cdot\mathrm{II}+\mathrm{SL}$ time steps, i.e. the last iteration is issued after $(n-1)\cdot\mathrm{II}$ time steps and ends after the SL time steps to fully evaluate the result of the datapath. This means that the smaller II is relative to the schedule length, 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 the memory controller can handle in parallel. Additionally, operations relying on scarce on-chip resources, such as DSP slices, might be 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 individually for each time step. With the overlap, the constraints now have to hold for congruence classes of time steps (step# modulo II).

Consider the example loop and its corresponding dataflow graph in Fig. 1a. We assume that the multiplication has a latency of 2, and all other operations have a latency of 1, i.e. their result is available in the next time step. The solid edges are precedence edges; the dashed edge represents a dependency on the addition's value in the previous iteration. For brevity, the loop counter and the test for the exit condition are omitted. The limited resource in this example is the memory port, which can serve either the read or the write operation in each time step. Fig. 1b illustrates the pipelined execution of this loop for an II of 2. Note that the

[1]**Mo**dulo **o**verlap **va**riable **c**onstraints



(a) Example loop, and data-flow graph of its body

(b) Modulo schedule and its overlapping execution for II=2

Figure 1: Pipelined execution of a loop

inter-iteration dependency and the resource constraint are still fulfilled by the overlapping execution of the iterations. The read operation is started in every even time step, i.e. in the congruence class 0 modulo 2, and the write operation is started in every odd time step, i.e. in the congruence class 1 modulo 2.

Computing a (preferably small) feasible II and the associated operation schedule is called *modulo scheduling*.

1.1 Related work

Loop pipelining is most useful when targeting processors having 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 modulo scheduling problem is the same, with the primary objective being to find a schedule with the minimally feasible II. However, VLIW modulo schedulers typically face tight resource constraints in each cycle (due to the limited number of functional units, registers, and buses). Modulo schedulers targeting HLS face fewer resource constraints (hardware accelerators may use dozens or even hundreds of operators), but need to support *operator chaining*, i.e. scheduling *multiple* low-latency operators into a *single* clock cycle, to achieve acceptable performance. Thus, when considering related work, not all techniques proposed for one target may be beneficial for the other.

Modulo schedules can be determined either by heuristics or by solving mathematical formulations, typically in the form of integer linear programs (ILP). While heuristics are not guaranteed to find the optimal solution (and often only produce feasible solutions), they usually come with a shorter computation time.

Rau and Glaeser first described the concept of modulo scheduling for a VLIW architecture [15]. Since then, several heuristic algorithms were proposed (e.g. [9, 14, 13]) and compared in a survey by Codina et al. [2].

More recently, Zhang and Liu [20] implemented a modulo scheduler on top of the SDC scheduling framework [3]. The flexibility of the SDC framework enables the easy integration of operator chaining into the modulo scheduling process, in contrast to previous attempts to extend other modulo scheduling algorithms with this capability [17]. The idea was improved by Canis et al. [1], marking the state of the art in HLS modulo scheduling. However, no comprehensive study investigating the quality of scheduling results for loops from typical HLS input programs has been published yet.

An early work by Hwang et al. [11] presented a mathematical formulation of a modulo scheduler capable of handling many HLS-specific requirements, but its practicality was only evaluated with one small example loop.

The most recent and advanced mathematical formulations of the problem were proposed by Eichenberger and Davidson [7] and Dupont de Dinechin [6]. Both formulations employ time-indexed decision variables, i.e. a binary variable x_i^t models that an operation i is scheduled to time step t with $t \in [0, II)$ [7] or $t \in [0, T)$ for an expected maximum schedule length T [6].

Our proposed formulation uses integer valued variables to model the start times, and overlap variables [5] to handle resource and modulo constraints. Venugopalan and Sinnen [18] showed that, compared to classical approaches, the overlap formulation leads to a smaller specification of the underlying problem structure and thus faster solution times.

We expect to retain these beneficial properties when re-

purposing the formulation for the modulo scheduling problem. For example, in contrast to the time-indexed approaches, the number of decision variables is independent of the candidate II, and resource-unlimited operations, which are common in the HLS setting, are represented by only one variable. A comparison of our approach against a reimplementation of Eichenberger's formulation will show that this is indeed the case (see Section 3.2.2).

Without explicitly naming them, overlap variables were used in a non-modulo scheduling formulation by Wilson et al. [19].

Still, to the best of our knowledge, overlap formulations so far have not been presented and evaluated for solving the HLS modulo scheduling problem.

1.2 Contributions

As its core contribution, this paper proposes a new ILP-based formulation of the HLS modulo scheduling problem with exact resource constraints using overlap variables that is faster on average than both a heuristic approach and a previous ILP formulation. A second contribution of this work is an extensive experimental evaluation of the techniques using two popular benchmark suites, with observations on the applicability of modulo scheduling to loops typically found in HLS kernels. This work is also the first to assess the quality of results of the Modulo SDC heuristic at this scale.

1.3 Structure

First, we discuss considerations that are common to all modulo scheduling approaches. Then, we present Moovac, our novel ILP-based formulation of the problem, in comparison to Canis' state-of-the-art Modulo SDC algorithm. In the experimental evaluation section, we show the results of scheduling loops from the CHStone and MachSuite benchmark suites and discuss specific findings. Before concluding, we propose ideas to further improve Moovac to make it feasible to schedule even larger loops in the future.

2. MODULO SCHEDULING

The primary objective for any modulo scheduling algorithm is to find the smallest II that guarantees that all inter-iteration dependencies and all resource constraints are met.

Modulo schedulers usually determine a lower bound for the II and attempt to schedule a loop with increasing candidate IIs until a feasible solution is found. We review the well-known calculation of such a bound and propose an improvement at the end of this section.

The problem signature in Table 1 characterises a single scheduling attempt for one fixed candidate II.

The input consists of a directed graph consisting of a set of operations O (nodes) with a fixed integer latency of D_i, and a set of directed edges E modeling the dataflow and other precedence relations among the operations. In addition to the usual intra-iteration dependencies, a loop may contain inter-iteration dependencies, also known as recurrences or loop-carried dependencies. As these dependencies point in the opposite direction of the normal dataflow, we call them *backedges*.

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. Let R be the set of distinct resources types, whose usage has to be limited when scheduling. We assume that every resource type $k \in R$ provides a_k uniform *instances* that can be used by 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 k concurrently. We will identify the different instances by an integer index in the range $[0, a_k - 1]$.

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

Once a feasible solution has been found, we are interested in an integer start time t_i for each operation and, for resource-limited operations, an assignment to a specific instance of the appropriate resource type.

Table 2 shows the scheduling problem of the initial example from Fig. 1a in this notation.

Table 1: Problem signature for modulo scheduling

Input	
II	candidate initiation interval
$O = \{0, \dots, n-1\}$	operations
$D_i,\ i \in O$	delay / latency of operation i
$E = \{(i \to j; b)\} \subseteq O \times O \times \{0,1\}$	dependency edges in datapath. $b = 1 \Leftrightarrow$ edge is a backedge
$R = \{\mathsf{mem}, \mathsf{dsp}, \dots\}$	resource types
$A = \{a_k \mid k \in R\}$	available instances of resource type k
$L_k \subseteq O$	resource-limited operations of type k
$L = \bigcup_{k \in R} L_k$	union of all resource-limited operations
Output	
$t_i,\ i \in O$	start time for operation i
$r_i,\ i \in L$	index of resource instance used by operation i

Table 2: Problem specification and resulting modulo schedule for the loop in Fig. 1a

$$
\begin{aligned}
O &= \{\texttt{+}, \texttt{*}, \texttt{read}, \texttt{write}\} \\
D_{\texttt{+}} &= D_{\texttt{read}} = D_{\texttt{write}} = 1 \quad D_{\texttt{*}} = 2 \\
E &= \{(\texttt{+} \to \texttt{*}; 0), (\texttt{read} \to \texttt{*}, 0), (\texttt{+} \to \texttt{write}; 0)\} \\
&\quad \cup \{(\texttt{+} \to \texttt{+}; 1)\} \\
R &= \{\mathsf{mem}\} \\
a_{\mathsf{mem}} &= 1 \\
L_{\mathsf{mem}} &= \{\texttt{read}, \texttt{write}\} = L \\
t_{\texttt{+}} &= 0 \quad t_{\texttt{read}} = 0 \quad t_{\texttt{*}} = 1 \quad t_{\texttt{write}} = 3 \\
r_{\texttt{read}} &= 0 = r_{\texttt{write}} = 0
\end{aligned}
$$

2.1 Modulo SDC

Canis et al.'s recently proposed Modulo SDC algorithm [1] builds on top of a special type of linear program (LP) called system of difference constraints (SDC). Such an LP

only contains constraints of the form $x_1 - x_2 \leq C$, in which the decision variables x_1, x_2 represent operation start or end times, and C is a constant integer. By construction, a 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. Therefore, the operation start times extracted from such a solution can be used in the HLS flow without further processing.

It was shown that typical constraints occurring in HLS scheduling, especially the intra- and inter-iteration dependencies between operations, can be mapped to the SDC framework. However, due to their non-linear nature, it is not possible to directly handle the resource constraints for modulo scheduling within the framework.

To circumvent this problem, the Modulo SDC algorithm uses a heuristic that starts with a non-resource constrained schedule and iteratively assigns resource-limited operations to time steps. This assignment is fixed in the underlying SDC by adding new equality constraints. 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.

Canis mentions using a perturbation-based priority function to determine the order in which resource-limited operations are selected to be assigned, but states that no particular priority function is necessary due to the backtracking approach. We therefore opted for a simpler height-based priority function in our implementation of the Modulo SDC scheduler. In order to make it more comparable to our novel approach, we use a more generous (wall-clock) time budget instead of limiting the number of attempts to assign resource-limited operations in the original paper, and use CPLEX [4] as the underlying LP solver.

In summary, the Modulo SDC approach aims to stay within the confines of the SDC framework, which is LP-solvable in polynomial time, and handles the resource constraints heuristically on top of it.

2.2 Moovac

We propose to tackle the modulo scheduling problem differently: Our formulation will handle the resource constraints *integrated* with all other constraints, thus making *all* information on the problem available to the solver. We accept that this makes the formulation a general **Integer** LP (ILP) with exponential runtime in the worst case.

However, using a novel formulation approach [18] for the resource constraints, we expect to achieve a problem structure that can be solved in reasonable time for problem instances relevant in practice.

We first describe a HLS compiler-agnostic version of the formulation, and discuss additional constraints potentially required by concrete HLS systems at the end of this section.

2.2.1 Decision variables

We model the problem with the decision variables shown in Table 3: As stated at the beginning of the section, the output we are seeking is a start time t_i for every operation $i \in O$, and the resource instance index r_i for each resource-limited operation. Let $i \in L_k$, then r_i is identified by an integer index in the range $[0, a_k - 1]$. With the externally specified delay D_i, operation i's result will be available in time step $t_i + D_i$. These variables are directly part of our ILP formulation.

The ILP formulation needs to solve an ordering problem (ordering of the operations) and an allocation problem (allocating operations to available resources). There are various different approaches how to formulate such a scheduling problem in 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 [18]. We take this approach here.

First we define variables for all resource-limited operations $i \in L$ to reflect the nature of the modulo scheduling: m_i is the congruence class (modulo II) implied by the current start time t_i. m_i is represented by an integer in the range $[0, II - 1]$. y_i is a helper variable in the computation of m_i; its value is bound to the integer division t_i/II. The start time t_i can then be expressed as $t_i = y_i \cdot II + m_i$

The *overlap variables* on all pairs of operations $i, j \in L_k$ that use the same resource type k are then defined as follows: $\varepsilon_{ij} = 1$ 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. We will use these overlap variables to express the "inequality" relation in the constraints below to enforce correct feasible ordering and resource allocation.

2.2.2 Objective function

We instruct the ILP solver to compute an ASAP schedule by requesting the sum of each operation's start time to be minimized (Eq. (4) in Fig. 3).

2.2.3 Constraints

Fig. 3 shows the constraints in the Moovac formulation.

Precedence.

All precedence relations are modeled by constraint (5). It is best explained by considering the cases "normal edge" and "backedge" separately. For $b = 0$, the constraint models the fact that the target operation j can only be started after the source operation i computed its result. For $b = 1$, note that i will be started later than j in the schedule. The constraint now enforces that the distance between i's end time and j's start time is shorter or equal to the candidate II, i.e. i will have computed its result before j is started in the next iteration.

Resources.

We use the overlap variables discussed above to model whether two operations use the same resource instance (ε_{ij}) or whether they start in time steps in the same congruence class (μ_{ij}) in an overlapping manner.

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 constraints (6) and (9).

The overlap variables are bound to their desired values by the constraint pairs (7), (8) and (10), (11), respectively. For brevity, we explain their function in the context of μ_{ij} (10), (11), as the constraints for ε_{ij} (7), (8) work analogously.

Constraint (10) is fulfilled if

$$m_i < m_j \quad \text{or} \quad \mu_{ij} = 0 \tag{1}$$

Constraint (11) is fulfilled if

$$m_i \geq m_j \quad \text{or} \quad \mu_{ij} = 1 \tag{2}$$

In both 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 expression. Due to the apparent contradictions, this pair of 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.

With the help of these overlap variables, the actual combined modulo and resource constraint (12) ensures that every pair of operations i, j is either assigned to different resource instances ($\varepsilon_{ij} + \varepsilon_{ji} = 0 \Leftrightarrow r_i = r_j$) or mapped to different congruence classes, or both.

Constraint (13) defines the congruence class index m_i as expressed by the operation's start time t_i modulo II.

Constraint (14) bounds the resource instance indices for each limited operation of type k to be less than the externally specified limit a_k. Analogously, constraint (15) ensures that an operation's congruence class index is less or equal to the candidate II.

An upper bound for the schedule length.

With constraint (16) we cap each operation's latest finish time, and in effect the datapath's overall latency, to a candidate-II-independent upper bound (explained below), which limits the ILP solver's freedom to fruitlessly try out increasingly late start times for operations in the search.

A modulo schedule may finish later than a non-modulo schedule adhering to the same precedence and resource constraints, e.g., when an operation on the datapath's critical path has to be started later because the desired modulo slot is already occupied by another operation. This is generally acceptable, as the expected performance gains by executing the datapath with a smaller II outweigh the increased latency, especially if it is known that a loop runs for many iterations once started.

The proposed bound is based on a construction scheme. We illustrate its steps in Fig. 2 by applying it to our introductory example. Assume we schedule all operations strictly sequentially (1). By ignoring the backedges, the dependencies between the operations are acyclic, therefore a topological order exists. We search for the operation with the longest latency D_{max} (2), and reserve a window of D_{max} time steps for every operation in the loop (3). Some windows' start times may be mapped to the same congruence class. In the example, we show the situation for $D_{max} = II = 2$, where all windows start in time steps congruent to 0 modulo II. In order to resolve the resource conflict between the read and the write operation, an empty time step is introduced before the window containing the write operation, effectively moving the write's start time and the start times of all subsequent operations into another congruence class. In the worst case, D_{max} is a multiple of the candidate II. Then, the start times of all moved windows still fall into the same congruence class. It may thus be required to insert empty time steps in front of every window.

The latest finish time of such a schedule, and also our proposed upper bound, is

$$SL_{max} = |O| \cdot (D_{max} + 1) \tag{3}$$

Note that this scheme does not lead to a feasible modulo schedule in general, as it does not obey the fact that all backedges have to be of shorter or equal length (defined as $t_i + D_i - t_j$ for a backedge $i \rightarrow j$) than the II.

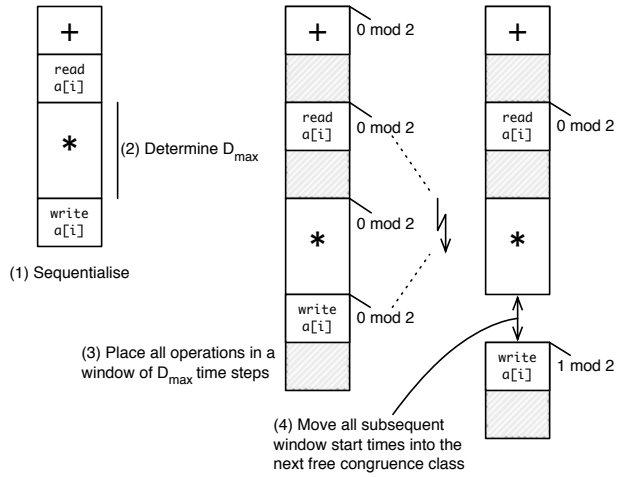


Figure 2: Construction scheme for SL_{max}, illustrated for the example loop in Fig. 1a and II = 2

Domain constraints.

Constraints (17) - (22) are domain constraints to enforce non-negativity respectively boolean values (expressed as integers 0 and 1) for the decision variables.

Additional constraints.

In its generic form above, the Moovac formulation is applicable to any HLS. Concrete HLS systems however may impose additional constraints on the resulting schedule, e.g. in Nymble [10], a HLS system that internally uses a per-loop control-dataflow graph as its main intermediate representation, there are special transfer operations that retrieve values from inner loops. Nymble's backend expects that these operations are scheduled exactly one time step after the operation representing the inner loop finishes. Let *loop* be the loop operation, and $xfer$ the transfer operation. Then a constraint of the form $t_{loop} + D_{loop} + 1 = t_{xfer}$ is required to achieve the desired schedule. These *relative timining constraints* [3] can be added easily to the Modulo SDC's underlying LP as well as to Moovac's ILP - a benefit of using a mathematical framework for scheduling.

Operator chaining, i.e. combinatorially combining multiple operations in a time step, is an essential technique in HLS to achieve good performance. We support operator chaining by allowing operations with $D_i = 0$. *Cycle time constraints* [3] prohibit excessive chaining, based on platform-specific operator latencies. The basic idea is to estimate the combinatorial delay between pairs of operations i, j. Then, by dividing the delay by the desired cycle time (e.g. 10 ns for 100 MHz), the number of time steps s that must separate i's and j's start times are computed and encoded as a constraint of the form $t_i + s \leq t_j$.

In our Modulo SDC and Moovac implementations, we use the same relative timing and cycle time constraints.

2.2.4 Time-limited operation

Each scheduling attempt with the Moovac formulation is executed as one invocation of the ILP solver. This enables

$$\textbf{minimize} \sum_{i \in O} t_i \tag{4}$$

$$\begin{aligned}
\text{subject to } & t_i + D_i && \leq t_j + b \cdot II && \forall i \to j; b \in E && (5)\\
& \epsilon_{ij} + \epsilon_{ji} && \leq 1 && \forall k \in R : \forall i, j \in L_k, i \neq j && (6)\\
& r_j - r_i - 1 - (\epsilon_{ij} - 1) \cdot a_k && \geq 0 && \forall k \in R : \forall i, j \in L_k, i \neq j && (7)\\
& r_j - r_i - \epsilon_{ij} \cdot a_k && \leq 0 && \forall k \in R : \forall i, j \in L_k, i \neq j && (8)\\
& \mu_{ij} + \mu_{ji} && \leq 1 && \forall k \in R : \forall i, j \in L_k, i \neq j && (9)\\
& m_j - m_i - 1 - (\mu_{ij} - 1) \cdot II && \geq 0 && \forall k \in R : \forall i, j \in L_k, i \neq j && (10)\\
& m_j - m_i - \mu_{ij} \cdot II && \leq 0 && \forall k \in R : \forall i, j \in L_k, i \neq j && (11)\\
& \epsilon_{ij} + \epsilon_{ji} + \mu_{ij} + \mu_{ji} && \geq 1 && \forall k \in R : \forall i, j \in L_k, i \neq j && (12)\\
& t_i && = y_i \cdot II + m_i && \forall i \in L && (13)\\
& r_i && \leq a_k - 1 && \forall k \in R : \forall i \in L_k && (14)\\
& m_i && \leq II - 1 && \forall i \in L && (15)\\
& t_i + D_i && \leq SL_{max} && \forall i \in O && (16)\\
& t_i && \in \mathbb{N} && \forall i \in O && (17)\\
& r_i && \in \mathbb{N} && \forall i \in L && (18)\\
& y_i && \in \mathbb{N} && \forall i \in L && (19)\\
& m_i && \in \mathbb{N} && \forall i \in L && (20)\\
& \varepsilon_{ij} && \in \{0, 1\} && \forall k \in R : \forall i, j \in L_k, i \neq j && (21)\\
& \mu_{ij} && \in \{0, 1\} && \forall k \in R : \forall i, j \in L_k, i \neq j && (22)
\end{aligned}$$

Figure 3: Moovac: Objective function and constraints

Table 3: Moovac: Decision variables

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

t_i, r_i, y_i and m_i are non-negative integer variables. ε_{ij} and μ_{ij} are integer variables constrained to $\{0, 1\}$.

us to impose a time limit on the solution search for a more practical application. Should the solver deplete the time budget without finding any feasible solution, we register the current candidate II as infeasible. When the solver does find a solution within the time limit, but was not able to prove its optimality in respect to the objective function (Eq. (4) in Fig. (3)), we accept that best-so-far solution. Alongside the solution, the ILP solver returns a gap value, specifying how close the feasible solution is to the yet-to-be-found optimal solution, in terms of the ASAP objective function. When the solver returns a solution without depleting any time limit, the solution is optimal, both in terms of the II (it is the minimal feasible II) and the ASAP schedule.

2.3 Bounds for the II

All approaches compared in this paper use the II as an input parameter for the modulo scheduling problem. They start with a lower bound value for the II and increment it by one until a feasible schedule is found. The closer the lower bound is to the smallest feasible II, the less scheduling attempts are needed. Tight bounds therefore reduce the expected overall scheduling runtime.

A trivial upper bound $MaxII$ for the candidate IIs is the length of any resource-constrained *non-modulo* schedule. Candidate IIs larger than this value indicate that it would actually be faster to execute this loop in a non-overlapping manner.

A well-known lower bound $MinII$ is defined as

$$MinII = \min(ResMII, RecMII) \tag{23}$$

that is, the minimum of the resource-constrained minimum II and the recurrence-constrained minimum II. The former is defined as

$$ResMII = \max_{k \in \text{resource types}} \left\lceil \frac{\#\text{ops using resource k}}{\#\text{available slots for k}} \right\rceil \quad (24)$$

This is an application of the pigeonhole principle: A candidate II cannot be feasible if the loop contains more operations using a limited resource k than can be assigned to the available resource slots in every modulo slot $0 \dots (II - 1)$.

The recurrence-constrained minimum II is induced by the backedges. For example, consider an operation in iteration i that depends on the value of another operation computed in iteration $i - 1$. For a candidate II to be feasible, the precedence between the two operations must still hold even with the overlapping implied by such an II. Formally, we define

$$RecMII = \max_{b \in \text{backedges}} \{\min \text{length}(b)\} \quad (25)$$

Canis et al. [1] propose to enumerate all cycles in the dataflow graph representing the loop's body in order to calculate the $RecMII$.

However, this can be simplified if the backedges are known in advance: We use a non-modulo and non-resource-constrained SDC scheduler to quickly calculate optimal ASAP and ALAP start times for the source and target operations of each backedge $b = i \to j$. Its minimum length can then be determined as

$$\min \text{length}(b) = \text{ASAP}(i) + D_i - \text{ALAP}(j) \quad (26)$$

The ASAP/ALAP scheduler takes the same compiler-specific constraints as the modulo schedulers into account, resulting in a more realistic lower bound for the II.

Note that both approaches (Canis' and ours) actually underestimate the $RecMII$ somewhat. This is due to them considering each backedge *individually*. However, due to interactions between the dependency cycles formed by the backedges, it might not be possible to schedule them all at their minimal total length (sum of edge lengths). This inaccuracy (which we will address in future work) does not affect the solution quality of our approach, it may just induce futile solver iterations attempting to reach an unreachable minimum II.

Also, we currently consider all backedges to express loop-carried dependencies to the *immediately preceding* iteration, leading to conservatively larger IIs. A better dependency analysis (also the subject of future work) will be able to accurately discover longer dependency distances, and thus allow for smaller II (as the dependency will be fulfilled over *multiple* loop iterations).

3. EXPERIMENTAL EVALUATION

We now evaluate the Moovac and Modulo SDC schedulers, implemented as presented in this work, on a large number of benchmarks from the CHStone [8] and MachSuite [16] collections.

In order to assess the solution speed of the Moovac scheduler in comparison to prior mathematical formulations, we also implemented Eichenberger's modulo scheduling approach (denoted as "EB" in the discussion below) as a representative of the traditional time-indexed formulations. It is defined by Equations (1), (2), (5) and (20) in [7]. We addded the same cycle time- and relative timing constraints and objective function as used in the other schedulers.

Table 4: Resource limits

Resource type	Available instances
Memory `Load/Store`	1 each
Integer `Div`	8
Integer other	∞
FP `Add/Sub/Mul`	4 each
FP other	2 each

3.1 Test setup

We implemented all schedulers in the Nymble HLS compiler [10]. Nymble is based on the LLVM framework [12], version 3.3[1], 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.

The generated schedules were verified by RTL simulation of the accelerator modules generated by Nymble.

The scheduling approaches were run on a server system with Intel Xeon E5-2667 v2 processors operating at 3.3 GHz, and 256 GB RAM. We used CPLEX 12.6.3 [4] as (I)LP solver for all schedulers. CPLEX was configured to run in its single-thread mode.

The schedulers operated according to the resource limits in Table 4.

3.2 Results

We present the scheduling results for the benchmark programs in Table 5. In our experiments, we used a baseline time budget of 5 minutes, and an extended budget of 60 minutes to assess whether allowing more runtime improves the overall scheduling quality. The time budgets were used to limit the search for a feasible schedule for a single candidate II, including the time to construct the respective linear programs via the CPLEX API.

For each test case, we list the number of loops it contains, as well as the average number of operations, resource-limited operations, and precedence edges contained in these loops. We summarise the scheduling attempts with the different approaches by counting the number of loops for which a valid modulo schedule was found. For these loops, we count the graphs that could be scheduled with the candidate II achieving the lower bound (column "II = MinII"), and the graphs that could be scheduled with an II strictly less than the trivial upper bound (column "II < MaxII"). The latter two columns are independent and do not necessarily add up to the overall number of feasible schedules. Also, note that $MinII$ and $MaxII$ may be equal. The column "timeout" gives the number of scheduling attempts that were canceled because the time budget was depleted for a candidate II. We prefix the number of timeouts with a star for each loop for which a feasible solution was found, but the solver was unable to prove its optimality. To characterise the effort needed

[1] A port to the most recent version 3.8 is currently underway.

for scheduling, we present the accumulated time required for all scheduling attempts. In the remaining columns, we compare the IIs found by Moovac against the results obtained with Modulo SDC and EB, and count the number of graphs for which either approach found the shortest II.

As an example, we interpret the 5 minute budget results for CHStone/adpcm: The test program contains 30 loops that have on average 94 operations, 9 resource-limited operations and 317 precedence edges.

Moovac and Modulo SDC are able to find a feasible modulo schedule for all 30 loops, whereas EB only does so for 28 loops. For 28 of the 30 loops, these schedules achieve the lower bound $MinII$ for the respective loop. However, many loops in this example have $MinII = MaxII$. Thus, a feasible II lower than $MaxII$ exists only for 9 of the 30 loops. Conversely, this means that 21 of the 30 loops are not amenable to loop pipelining at all. Scheduling the 30 loops took a total of 256 minutes with Moovac, 232 minutes with Modulo SDC and 408 minutes with EB. These total times include attempts in which the 5 minute time budget would be exceeded for a candidate II. In this case, Moovac exceeded its allocated budget 51 times, whereas Modulo SDC only did so 45 times, and EB hit the limit 81 times. For two loops, the ILP solver found a feasible, but possibly non-optimal solution for the linear program generated by Moovac within the time limit. For one loop each, Moovac and Modulo SDC found a schedule for a smaller II than the other approach. The table shows that Moovac finds schedules with a shorter II than EB for 2 loops. In this particular case, these two loops are the ones for which EB does not find any valid modulo schedule at all within the time budget. For the remaining 28 loops, all schedulers where able to schedule the loop with the same II.

3.2.1 Moovac vs. Modulo SDC

The results show that it is possible to find modulo schedules for almost all loops in 22 typical HLS input programs in just over 8 hours when using the 5 minute time limit per scheduling attempt. Both approaches encountered at least two loops each where they struggled to find a solution even for the trivial $MaxII$. In terms of the general applicability of modulo scheduling to HLS kernels, 85 % of loops are scheduled with the lower bound II, and roughly a half of the loops are in principle amenable to loop pipelining, as they are scheduled with an II $< MaxII$.

Our study is the first to confirm that the state-of-the-art Modulo SDC heuristic is actually finding optimal schedules for the majority of loops. However, we also observed that Moovac, despite its ability to find provably optimal IIs, is surprisingly fast: It schedules the 200+ smallest loops (with less than 125 operations) in under 3 minutes. A key benefit of the integrated problem formulation is Moovac's ability to actually *prove* the infeasibility of the ILP constructed for a scheduling attempt, whereas the Modulo SDC scheduler might get stuck without ever finding a solution, indicated here by the number of loops that encountered a timeout. Among the remaining 22 larger loops, we discover a few loops (in CHStone/adpcm and MachSuite/aes) where the scheduling problems apparently become too complex to be solved by the ILP solver in the given time budget. Here, the Modulo SDC approach is faster and better, as it finds a solution for a smaller II than the Moovac scheduler.

These findings lead us to propose a *synergistic* scheduling approach, where we run *both* schedulers in *parallel* on each loop, and accept the *first* feasible solution from either Moovac or the Modulo SDC scheduler. With this approach, we are able to schedule all loops in 429 minutes, in comparison to 489 minutes when using only Moovac and 753 minutes when using Modulo SDC alone, showing that the strengths of both approaches complement each other.

In these experiments, Moovac is approximately 1.5x times faster than Modulo SDC, but computing similar results. Furthermore, given sufficient time, Moovac finds the optimal solution. When combining Moovac with Modulo SDC, the scheduling time is reduced by an additional 15 %. Please note that the initially published version of Modulo SDC did not use a time limit, but imposed a limit on the number of backtracking *steps* before increasing the candidate II. However, our early experiments showed that this mechanism prevented Modulo SDC from finding solutions for large graphs in CHStone, as the original experimental evaluation of Modulo SDC used smaller graphs than those in our benchmarks. For a fair comparison, we run both algorithms with the same *time limit* for each candidate II instead.

Compared with the state-of-the-art Modulo SDC approach, Moovac not only computes optimal results, but is also surprisingly competitive with regard to its execution time (which can be reduced even further, see Section 4).

3.2.2 Moovac vs. Eichenberger's formulation

A comprehensive evaluation of Moovac necessitates a comparison with another optimal approach. EB (also ILP-based) is such a formulation. While it was not designed for the requirements of HLS modulo scheduling, e.g. handling large numbers of resource-unlimited operations and operator chaining, our experimental results show that it is actually capable of finding valid modulo schedules for the majority of loops of our benchmark programs in a HLS context. In MachSuite/aes, it is even faster than both Moovac and Modulo SDC. However, overall, the other approaches require less time for scheduling, indicated by the many timeouts for EB in the "large loop" category, and the accumulated scheduling time for the "small loop" category, i.e. 3 minutes for Moovac compared to 5 minutes for EB.

3.2.3 Extended time budget

The results for the 60 minute time budget show that Modulo SDC and, even more clearly, EB benefit from the extra time: Beyond the reduced number of timeouts overall, Modulo SDC and EB now find *shorter* IIs than Moovac for loops in CHStone/adpcm, CHStone/aes and MachSuite/aes. EB completes the latter in less than half time of the other approaches. But even with the increased budget, Moovac schedules the entire benchmark suite in the shortest run time.

4. CONCLUSION AND FUTURE WORK

This paper proposed Moovac, a novel, overlap-variable based ILP formulation of the high-level synthesis modulo scheduling problem.

While Moovac can be used to assess the performance of modulo scheduling heuristics like Modulo SDC, we showed in this paper that it can also be used as an approach in itself. Moovac was able to find the optimal solutions for most of the loops in only a matter of minutes, often scheduling

Table 5: Scheduling results

	loops	Average problem size: operations	limited ops	edges	Moovac: Schedule found	II = MinII	II < MaxII	timeouts	Time (min)	Modulo SDC: Schedule found	II = MinII	II < MaxII	timeouts	Time (min)	Eichenberger: Schedule found	II = MinII	II < MaxII	timeouts	Time (min)	Shorter II: Moovac	Both same Modulo SDC	Moovac	Both same Eichenberger
Time budget: 5 min																							
CHStone																							
adpcm	30	94	9	317	30	28	9	⋆51	**256**	30	28	9	45	**232**	28	28	7	81	**408**	1	28	1	2	28	0
aes	24	177	27	1327	22	19	12	⋆16	**82**	22	19	12	21	**106**	22	19	12	15	**89**	0	24	0	0	24	0
blowfish	1	784	107	9600	1	0	1	⋆8	**41**	0	0	0	14	**71**	0	0	0	14	**72**	1	0	0	1	0	0
dfdiv	2	60	0	198	2	2	0	0	**< 1s**	2	2	0	0	**< 1s**	2	2	0	0	**< 1s**	0	2	0	0	2	0
dfsin	3	935	23	57465	3	2	1	0	**3**	3	2	1	5	**27**	2	2	0	31	**156**	1	2	0	1	2	0
gsm	15	81	4	329	15	13	6	0	**< 1s**	15	13	6	1	**6**	15	13	6	0	**2**	0	15	0	0	15	0
mips	1	1124	65	40796	1	0	1	⋆2	**11**	1	0	1	2	**14**	0	0	0	21	**109**	1	0	0	1	0	0
motion	51	69	5	217	51	51	34	0	**< 1s**	51	51	34	0	**< 1s**	51	51	34	0	**1**	0	51	0	0	51	0
sha	25	87	3	287	25	18	7	0	**3**	25	18	7	14	**71**	25	18	7	0	**10**	1	24	0	0	25	0
MachSuite																							
aes	16	84	13	342	16	13	15	⋆19	**96**	16	13	16	20	**101**	15	13	15	15	**84**	0	15	1	1	15	0
bfs_bulk	3	77	5	230	3	2	2	0	**< 1s**	3	2	2	2	**11**	3	2	2	0	**1**	0	3	0	0	3	0
bfs_queue	2	132	9	520	2	0	1	0	**1**	2	0	1	7	**36**	2	0	1	0	**1**	0	2	0	0	2	0
gemm_blocked	5	45	2	115	5	5	2	0	**< 1s**	5	5	2	0	**< 1s**	5	5	2	0	**< 1s**	0	5	0	0	5	0
gemm_ncubed	3	49	2	129	3	3	2	0	**< 1s**	3	3	2	0	**< 1s**	3	3	2	0	**< 1s**	0	3	0	0	3	0
kmp	4	76	4	323	4	3	1	0	**< 1s**	4	3	1	0	**< 1s**	4	3	1	0	**1**	0	4	0	0	4	0
md_knn	2	124	22	312	2	1	2	0	**2**	2	1	2	8	**41**	2	1	2	0	**5**	1	1	0	0	2	0
nw	6	81	4	372	6	4	3	0	**1**	6	4	3	5	**26**	6	4	3	0	**1**	0	6	0	0	6	0
sort_merge	8	57	3	167	8	7	3	0	**< 1s**	8	7	3	0	**< 1s**	8	7	3	0	**< 1s**	0	8	0	0	8	0
sort_radix	15	49	3	131	15	14	7	0	**< 1s**	15	14	7	0	**< 1s**	15	14	7	0	**1**	0	15	0	0	15	0
spmv_ellpack	2	65	6	167	2	0	2	0	**< 1s**	2	0	2	2	**11**	2	0	2	0	**1**	0	2	0	0	2	0
stencil2d	4	57	2	144	4	4	2	0	**< 1s**	4	4	2	0	**< 1s**	4	4	2	0	**< 1s**	0	4	0	0	4	0
stencil3d	3	67	4	263	3	2	1	0	**1**	3	2	1	2	**11**	3	2	1	0	**1**	0	3	0	0	3	0
Total																							
all	225	106	9	1356	223	191	114	96	**489**	222	191	114	148	**753**	217	191	109	177	**932**	6	217	2	6	219	0
< 125 ops	203	64	4	196	203	187	96	0	**3**	203	187	96	26	**131**	203	187	96	0	**5**	1	202	0	0	203	0
≥ 125 ops	22	495	54	12058	20	4	18	96	**486**	19	4	18	122	**623**	14	4	13	177	**927**	5	15	2	6	16	0
Time budget: 60 min																							
CHStone																							
adpcm	30	94	9	317	30	28	9	⋆47	**2821**	30	28	9	43	**2608**	29	28	8	⋆41	**2461**	1	28	1	1	28	1
aes	24	177	27	1327	22	19	12	⋆16	**962**	24	19	14	10	**648**	22	19	12	15	**915**	0	22	2	0	24	0
blowfish	1	784	107	9600	1	0	1	⋆8	**481**	0	0	0	14	**841**	0	0	0	14	**844**	1	0	0	1	0	0
dfdiv	2	60	0	198	2	2	0	0	**< 1s**	2	2	0	0	**< 1s**	2	2	0	0	**< 1s**	0	2	0	0	2	0
dfsin	3	935	23	57465	3	2	1	0	**4**	3	2	1	1	**88**	2	2	0	31	**1861**	0	3	0	1	2	0
gsm	15	81	4	329	15	13	6	0	**< 1s**	15	13	6	1	**61**	15	13	6	0	**2**	0	15	0	0	15	0
mips	1	1124	65	40796	1	0	1	⋆2	**121**	1	0	1	2	**124**	0	0	0	21	**1263**	1	0	0	1	0	0
motion	51	69	5	217	51	51	34	0	**< 1s**	51	51	34	0	**< 1s**	51	51	34	0	**1**	0	51	0	0	51	0
sha	25	87	3	287	25	18	7	0	**3**	25	18	7	14	**841**	25	18	7	0	**10**	1	24	0	0	25	0
MachSuite																							
aes	16	84	13	342	16	13	15	⋆19	**1141**	16	13	16	20	**1201**	16	13	16	⋆8	**529**	0	15	1	0	15	1
bfs_bulk	3	77	5	230	3	2	2	0	**< 1s**	3	2	2	2	**121**	3	2	2	0	**1**	0	3	0	0	3	0
bfs_queue	2	132	9	520	2	0	1	0	**1**	2	0	1	7	**421**	2	0	1	0	**1**	0	2	0	0	2	0
gemm_blocked	5	45	2	115	5	5	2	0	**< 1s**	5	5	2	0	**< 1s**	5	5	2	0	**< 1s**	0	5	0	0	5	0
gemm_ncubed	3	49	2	129	3	3	2	0	**< 1s**	3	3	2	0	**< 1s**	3	3	2	0	**< 1s**	0	3	0	0	3	0
kmp	4	76	4	323	4	3	1	0	**< 1s**	4	3	1	0	**< 1s**	4	3	1	0	**1**	0	4	0	0	4	0
md_knn	2	124	22	312	2	1	2	0	**2**	2	1	2	8	**481**	2	1	2	0	**5**	1	1	0	0	2	0
nw	6	81	4	372	6	4	3	0	**1**	6	4	3	5	**301**	6	4	3	0	**1**	0	6	0	0	6	0
sort_merge	8	57	3	167	8	7	3	0	**< 1s**	8	7	3	0	**< 1s**	8	7	3	0	**< 1s**	0	8	0	0	8	0
sort_radix	15	49	3	131	15	14	7	0	**< 1s**	15	14	7	0	**< 1s**	15	14	7	0	**1**	0	15	0	0	15	0
spmv_ellpack	2	65	6	167	2	0	2	0	**< 1s**	2	0	2	2	**121**	2	0	2	0	**1**	0	2	0	0	2	0
stencil2d	4	57	2	144	4	4	2	0	**< 1s**	4	4	2	0	**< 1s**	4	4	2	0	**< 1s**	0	4	0	0	4	0
stencil3d	3	67	4	263	3	2	1	0	**1**	3	2	1	2	**121**	3	2	1	0	**1**	0	3	0	0	3	0
Total																							
all	225	106	9	1356	223	191	114	92	**5530**	224	191	116	131	**7969**	219	191	111	130	**7885**	5	216	4	4	219	2
< 125 ops	203	64	4	196	203	187	96	0	**3**	203	187	96	26	**1561**	203	187	96	0	**5**	1	202	0	0	203	0
≥ 125 ops	22	495	54	12058	20	4	18	92	**5527**	21	4	20	105	**6408**	16	4	15	130	**7881**	4	14	4	4	16	2

⋆/$^{\star}_{\star}$ = accepted one/two non-optimal schedule(s) after time budget was depleted

faster than both a state-of-the art heuristic and an established formulation.

Moreover, our extensive experimental evaluation showed that typical HLS kernels exhibit opportunities for loop pipelining, and modulo scheduling can reasonably be applied to a large range of loops. However, our evaluation discovered that there exist problems of very large sizes that can be a challenge for Moovac and its underlying ILP solver. For now, combining Moovac and the Modulo SDC heuristic to operate synergistically leads to an overall scheduling time faster than each individual approach. In the future, we will work on reducing the number of decision variables in the ILP by only scheduling the resource-limited operations and other essential operations, such as the endpoints of backedges. Subgraphs consisting only of resource-unlimited operations will then be replaced by precedence edges with a latency equal to the longest-path latency of the subgraph. After a feasible schedule is found for the reduced problem, all remaining operations can be fit in between the scheduled resource-limited operations by means of a simple ASAP technique.
We also envision to render the incremental search for the smallest feasible II unnecessary by minimising the II within the ILP. Again, we plan to use techniques from the operations research community to allow for efficient solving of the enhanced formulation.

5. ACKNOWLEDGEMENTS

The authors wish to thank Lukas Sommer for his implementation of the Modulo SDC scheduler. This work was partially funded by the EU in the FP7 research project *REPARA* (ICT-609666). The authors would also like to thank Xilinx, Inc. for supporting their work by hardware and software donations.

6. REFERENCES

[1] A. Canis, S. D. Brown, and J. H. Anderson. Modulo SDC scheduling with recurrence minimization in high-level synthesis. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, 2014.

[2] J. M. Codina, J. Llosa, and A. González. A comparative study of modulo scheduling techniques. In *Proceedings of the 16th international conference on Supercomputing - ICS '02*, 2002.

[3] J. Cong and Z. Zhang. An efficient and versatile scheduling algorithm based on SDC formulation. In *Proceedings of the 43rd annual conference on Design automation - DAC '06*, 2006.

[4] I. B. M. Corporation. IBM ILOG CPLEX Optimizer, version 12.6.3. http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/.

[5] A. Davare, J. Chong, Q. Zhu, D. M. Densmore, and A. L. Sangiovanni-Vincentelli. Classification, Customization, and Characterization: Using MILP for Task Allocation and Scheduling. *Systems Research*, 2006.

[6] B. Dupont de Dinechin. Time-indexed formulations and a large neighborhood search for the resource-constrained modulo scheduling problem. In *3rd Multidisciplinary International Scheduling conference: Theory and Applications (MISTA)*, 2007.

[7] A. E. Eichenberger and E. S. Davidson. Efficient formulation for optimal modulo schedulers. In *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation - PLDI '97*, volume 32, New York, New York, USA, 1997.

[8] Y. Hara, H. Tomiyama, S. Honda, and H. Takada. Proposal and Quantitative Analysis of the CHStone Benchmark Program Suite for Practical C-based High-level Synthesis. *Journal of Information Processing*, 17, 2009.

[9] R. A. Huff. Lifetime-sensitive modulo scheduling. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, 1993.

[10] J. Huthmann, B. Liebig, J. Oppermann, and A. Koch. Hardware/software co-compilation with the nymble system. In *Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC), 2013 8th International Workshop on*, 2013.

[11] C.-T. Hwang, J.-H. Lee, and Y.-C. Hsu. A formal approach to the scheduling problem in high level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 10, 1991.

[12] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, CGO*, 2004.

[13] J. Llosa, A. Gonzalez, E. Ayguade, and M. Valero. Swing module scheduling: a lifetime-sensitive approach. In *Parallel Architectures and Compilation Techniques, Proceedings of the Conference on*, 1996.

[14] B. R. Rau. Iterative modulo scheduling. In *Proceedings of the 27th annual international symposium on Microarchitecture - MICRO 27*, 1994.

[15] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. *ACM SIGMICRO Newsletter*, 12, 1981.

[16] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks. MachSuite: Benchmarks for accelerator design and customized architectures. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, 2014.

[17] M. Sivaraman and S. Aditya. Cycle-time aware architecture synthesis of custom hardware accelerators. In *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2002.

[18] S. Venugopalan and O. Sinnen. ILP Formulations for Optimal Task Scheduling with Communication Delays on Parallel Systems. *IEEE Transactions on Parallel and Distributed Systems*, 26, 2015.

[19] T. C. Wilson, N. Mukherjee, M. K. Garg, and D. K. Banerji. An integrated and accelerated ILP solution for scheduling, module allocation, and binding in datapath synthesis. In *VLSI Design, Proceedings of the Sixth International Conference on*, 1993.

[20] Z. Zhang and B. Liu. SDC-based modulo scheduling for pipeline synthesis. In *IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, ICCAD*, 2013.