Dependence Graph Preprocessing for Faster Exact Modulo Scheduling in High-level Synthesis

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

***Abstract*—Modulo scheduling is a key throughput optimisation when compiling for VLIW architectures, which has been applied successfully to high-level synthesis (HLS) of hardware accelerators in the past. However, problem instances in the HLS context usually have larger and denser dependence graphs and may contain many simple operations that are not subject to resource constraints, causing long runtimes with VLIW-centric modulo schedulers.**

We propose a complexity-reduction approach for existing exact modulo schedulers that retains their ability to compute provably optimal schedules, but shortens their runtime on typical HLS instances. The basic idea is to simplify a problem instance's dependence graph by abstracting entire subgraphs of non-critical operations with a single edge, then schedule this reduced problem comprising only the critical operations. A solution obtained for the reduced problem can be easily completed to a solution for the original problem.

Applied to the well-known, originally VLIW-centric, and exact ILP formulation by Eichenberger and Davidson, we show a mean speedup of 4.37x for 21 large instances, which makes it competitive again with the recently proposed, HLS-tailored Moovac formulation. As both formulations show different problem-dependent strengths and weaknesses, these insights are a first step towards an oracle that selects the most promising scheduler for a given problem instance.

I. Introduction

Loop pipelining, i.e. the partially overlapping execution of subsequent loop iterations, is an important throughput-optimising technique. It is enabled by modulo schedulers, which compute start times for the operations in a loop's body, ensuring that all inter-iteration dependences and resource constraints are satisfied when new iterations are started with a fixed number of time steps, called the *initiation interval* (II), apart.

Most of the research in the field targets VLIW processors that typically have only a handful of functional units and a limited number of registers [1], [2]. However, loop pipelining also plays an important role during the automatic generation of FPGA-based accelerators in a high-level synthesis (HLS) environment [3], [4]. Compared to the VLIW context, instances of the modulo scheduling problem (MSP) that arise in an HLS flow are much larger, due to implicit loop-wide if-conversion [5], and have denser dependence graphs, due to the additional edges required to model operator chaining [6], [7] as well as to retain a sequential ordering of the loop's memory accesses where needed. This is apparent in our test MSP set (cf. Section IV): The 21 instances contain 471 operations and 1578 edges (median values). The ratio of the aforementioned additional, non-data-dependence edges ranges from 29% to 90% with a median value of 63%.

On the other hand, HLS generally aims to create spatially distributed computations (employing a dedicated hardware operator for each operation node). This is possible, as many resources, e.g. logic gates, exist in abundance. Only a fraction of the operations needs to be time-multiplexed onto scarce shared resources, such as a limited number of memory ports, or large floating-point operators. The spatial approach thus results in far fewer resource constraints that need to be included in the HLS MSP than in the VLIW MSP. In our test instances, for example, the ratio of the limited operations ranges between 3% and 29% with a median value of 12%.

In a recent study [4], the ILP formulation by Eichenberger and Davidson [8] (abbreviated here as *ED*), which originates from a VLIW compiler context, performed significantly worse on average than the newly proposed Moovac formulation on a set of MSP instances taken from an HLS tool flow. However, given that the prior formulation has received widespread recognition in the community [9], and some instances in [4] appear to be scheduled faster with ED, we suspect that the inferior performance may be partly due to the aforementioned characteristics of HLS MSP.

To this end, we propose a problem-complexity reduction algorithm for HLS MSP instances that yields problem structures which are more similar in size and density to instances expected by a VLIW modulo scheduler. Our work specifically targets *exact* modulo schedulers that are able to compute provably optimal solutions regarding a schedule length minimisation objective. The proposed algorithm retains this property.

The key insight is that only some operations in an MSP instance are *critical* to the exact modelling, e.g. resource-limited operations, while others are not. Non-critical operations can be scheduled in an as-soon-as-possible manner, and thus, subgraphs of non-critical operations may be replaced by a single edge with a delay equal to the minimum delay of the subgraph's longest path. The reduction makes it tractable to discover additional, superfluous dependence edges. Figures 1 - 4 demonstrate the proposed transformation. After scheduling the reduced instance, the start times of the critical operations are fixed in the original problem, and start times for the remaining non-critical operations can be determined in polynomial time.

Problem reduction (or compression) in general has been explored in the Operations Research community to make challenging problems more tractable by excluding non-critical aspects (e.g. [10]). To the best of our knowledge, ours is the first use of the technique to speed-up modulo scheduling.

II. Background

A. Modulo scheduling problem

An instance $\mathcal{I}$ of the modulo scheduling problem (MSP) is defined by:

- the set of operations O. Every operation $i \in O$ has a *delay* (in time steps) D_i.
- the set of dependence edges $E = \{(i \to j)\} \subseteq O \times O$. Edges may carry an edge *delay*[1] d_{ij}. We use the notation $\delta_{ij} = D_i + d_{ij}$ to describe the minimal number of time steps that j can be started after i. Each edge models a precedence relation that has to be satisfied β_{ij} iterations later. We call β_{ij} the edge *distance*, and edges with $\beta_{ij} > 0$ inter-iteration dependences, or shorter, *backedges*, because they point in the opposite direction of the normal data flow. Conversely, we call edges with 0-distance *forward edges*.
- a resource model. We consider a set of resource types $R = \{\mathtt{mem}, \mathtt{dsp}, \dots\}$. There are a_k uniform and fully-pipelined instances of each resource type $k \in R$.
  Operations may require at most one resource type. The set $L_k \subseteq O$ contains all operations using k. $L = \bigcup_{k \in R} L_k$ is the set of all *resource-limited* operations. An operation $i \in L_k$ reserves exactly one k-instance in its start time step.
- a range of candidate initiation intervals $[\lambda^{\perp}, \lambda^{\top}]$.

The solution to an MSP instance consists of an integer initiation interval λ° and integer start times t_i for all $i \in O$ that satisfy the precedence constraints imposed by the dependence edges,

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

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

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

We consider modulo scheduling to be a bi-criteria optimisation problem. The first objective is to find the smallest II for which a feasible schedule exists. In this work, the second objective is to find the schedule with the shortest schedule length $T_\lambda = \max_{i \in O}\{t_i + D_i\}$ for a given interval λ. In practice, the first objective is far more important, as smaller IIs correspond to a higher throughput of the loop accelerator. To this end, it is common to attempt to solve the MSP for several fixed *candidate IIs*, starting with a lower bound $\lambda^{\perp}$ inferred from (1) and (2), and increasing the candidate II until a feasible schedule is found, or an upper bound $\lambda^{\top}$ is reached. Rau [11] presents several ways to compute $\lambda^{\perp}$. The length of any resource-constrained non-modulo schedule can serve as $\lambda^{\top}$, and may be used as a fall-back schedule in case a modulo scheduler does not find a solution with a smaller interval.

a) Running example: The MSP instance in Figure 1 is defined by the set of operations $O = \{1, 2, \dots, 11\}$, the set of edges $E = \{(1 \to 2), (1 \to 3), \dots\}$, and a simple resource model (not shown in the figure) comprising only a single limited resource type $\rho \in R$ that provides one instance ($a_\rho = 1$). We have $L_\rho = \{2, 6, 9\} = L$, and use a grey contour to distinguish these resource-limited operations from the unlimited ones. The operation delay D_i is 1 for all $i \in O$. There are no edge delays, i.e. $d_{ij} = 0$ for all $(i \to j) \in E$, and only a single backedge $(10 \to 3)$ with $\beta_{10\ 3} = 1$ and a dashed line style. All other edges $(i \to j) \in E$ are forward edges with $\beta_{ij} = 0$. The lower bound for the interval $\lambda^{\perp}$ is 5, due to the recurrence spanned by the backedge. We assume a non-modulo scheduler would determine a schedule of length 7, and define $\lambda^{\top}$ accordingly.

B. Exact modulo schedulers

Exact modulo schedulers, in contrast to algorithms that rely on *heuristic* simplifications (e.g. [3], [12], [13]), model all the above aspects of the problem and are therefore able to find provably optimal schedules. They are often defined in terms of a mathematical framework such as integer linear programs (e.g. [4], [8], [14]), or constraint programming (e.g. [15]), and, unfortunately, inherit the frameworks' exponential worst-case runtimes to find a solution. Still, they are a viable option in the context of an HLS system, as logic synthesis and place-and-route often requires multiple hours even on mid-sized FPGA devices anyway. This easily amortises the time spent to determine a high quality (provably optimal) solution by exact modulo scheduling.

However, most exact modulo schedulers were proposed for and evaluated with VLIW-style MSP instances. Using the ED formulation as the representative for this class of schedulers, we showed that HLS-style instances require special attention in the design of a practical exact scheduler [4]. Our Moovac formulation copes well with rather large instances, as the (many) unlimited operations are represented by a single integer decision variable. Similarly, a large number of dependence edges is unproblematic, as each edge results in only one linear constraint (Eq. 5 in [4]).

In contrast, the ED formulation uses II-many binary decision variables per operation, regardless whether the operation is subject to resource constraints or not. The authors achieved a significant speed-up by using II-many, but 0-1-structured[2] constraints to represent dependence edges (Eq. (20) in [8]).

[1]For example, we use edges with a delay of 1 to limit the amount of operator chaining (see Section IV).

[2]Every decision variable occurs only once, and is multiplied by either -1, 0, or 1 [8].

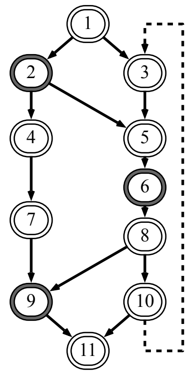

Fig. 1: Example instance

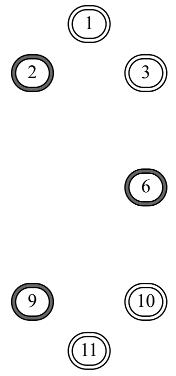

Fig. 2: Reduced instance: critical operations

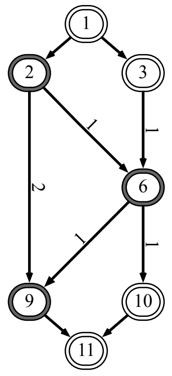

Fig. 3: Reduced instance: constructed edges

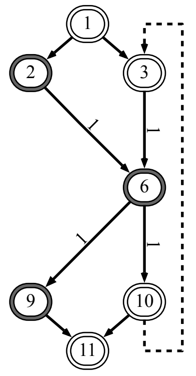

Fig. 4: Reduced instance: edges filtered, backedges added

These scalability difficulties when scheduling large and dense dependence graphs, and HLS-typical candidate IIs, make the ED formulation the ideal subject to demonstrate the strength of our proposed approach.

However, we expect other exact VLIW modulo schedulers, such as the formulations by Dinechin [14] and by Ayala and Artigues [16], to benefit from the proposed complexity reduction as well. A common feature in these formulations is the use of *time-indexed* binary decision variables to model the operations' start times. This design typically results in complex linear constraints involving sums of subsets of these variables. Reducing the number of operations in the problem, and reducing the density of dependence edges therefore gives ample opportunity for speed-up when retrofitting these formulations for use in an HLS setting.

III. Reduced-complexity modulo scheduling

Our reduced-complexity modulo scheduling approach comprises three steps. First, given an HLS-style MSP instance $\mathcal{I}$, we derive a reduced problem instance $\mathcal{I}'$ with the set of critical operations $Q \subseteq O$ and infer a new set of edges $F \subseteq Q \times Q$. Then, $\mathcal{I}'$ is scheduled with an arbitrary modulo scheduler, e.g. the ED formulation. The operation start times obtained from the solution to $\mathcal{I}'$ are fixed in $\mathcal{I}$. This makes scheduling $\mathcal{I}$ tractable, as it is no longer a resource-constrained scheduling problem and can thus be solved in polynomial time.

A. Construction of a reduced problem instance

Let $B = \{e = (u \to v) \mid e \in E \wedge \beta_{uv} > 0\}$ be the set of all backedges in E, $\text{pred}(v) = \{u \mid \exists (u \to v) \in E \text{ with } \beta_{uv} = 0\}$ denote the set of predecessors of an operation v, and analogously, $\text{succ}(v) = \{w \mid \exists (v \to w) \in E \text{ with } \beta_{vw} = 0\}$ denote the set of successors of v.

a) Critical operations: The proposed problem reduction shall guarantee that a feasible/optimal solution to $\mathcal{I}'$ can be completed to a feasible/optimal solution to $\mathcal{I}$.

The feasibility of $\mathcal{I}$ for a candidate interval λ is subject to the interaction of the backedges (which impose deadlines, i.e. latest possible start times) and resource constraints (which may require operations to be scheduled in a later time step than required by their predecessors' finish times), as defined by (1) and (2). Additionally, operations without non-backedge predecessors must be included in Q to capture the earliest start times in the schedule.

The schedule length depends, per definition, on the start times of operations without outgoing forward edges. These operations, in addition to the resource-limited operations, therefore need to be considered in $\mathcal{I}'$ to ensure that an optimal solution for $\mathcal{I}'$ may serve as the base for an optimal solution to $\mathcal{I}$.

With this considerations in mind, we define the set of *critical* operations Q to contain

- all resource-limited operations, L,
- operations at the endpoints of backedges, formally $\{v \mid \exists (u \to v) \in B \vee \exists (v \to w) \in B\}$, and
- operations with either no incoming or no outgoing forward edges, formally $\{v \mid \text{pred}(v) = \emptyset \vee \text{succ}(v) = \emptyset\}$.

Figure 2 depicts the set Q for the running example: operations 2, 6 and 9 are resource-limited, 3 and 10 are the endpoints to the only backedge, and 1 and 11 are the extremal vertices in the dependence graph.

In contrast, the start times of the remaining non-critical operations are determined only by the instance's forward edges when considering schedule length minimisation as the secondary objective. Entire subgraphs of non-critical operations can then be scheduled in an as-soon-as-possible manner between critical operations, given that enough time steps separate the critical operations. As a consequence, such subgraphs can be modelled in $\mathcal{I}'$ by a single edge with the minimally required delay to fit the non-critical operations, as discussed in the next section.

b) Edge construction: The set of edges F is not a subset of E. Rather, it contains forward edges that are constructed based on the results of a data flow analysis that computes $\Delta_v^{\text{IN}}(q)$, i.e. the length of longest path (in terms of operation and edge delays) in E from a critical operation $q \in Q$ to an operation $v \in O$.

The function $\Delta_v^{\text{IN}}(q) : (O \times Q) \to \mathbb{N}_0 \cup \{-\infty\}$ is defined by the data flow equations (3) and (4), where the value $-\infty$ is to be interpreted as "unreachable".

$$\Delta_v^{\text{IN}}(q) = \begin{cases} -\infty & \text{if } \text{pred}(v) = \emptyset \\ \max\limits_{u \in \text{pred}(v)} \Delta_u^{\text{OUT}}(q) + \delta_{uv} & \text{otherwise} \end{cases} \tag{3}$$

$$\Delta_v^{\text{OUT}}(q) = \begin{cases} 0 & \text{if } v = q \\ -\infty & \text{if } v \in Q \land v \neq q \\ \Delta_v^{\text{IN}}(q) & \text{otherwise} \end{cases} \tag{4}$$

The important distinction from a standard longest-path computation is the fact that only paths from the nearest preceding critical operations are considered: the definition of $\Delta_v^{\text{OUT}}(q)$ for a critical operation $v \in Q$ sets the outgoing path length from itself ($v = q$) to 0, and resets the path length concerning all other critical operations to $-\infty$.

After computing $\Delta_v^{\text{IN}}(q)$, we construct new forward edges between the critical operations:

$$\begin{aligned} F_{\text{cons}} = \{(p \to q), d_{pq} = \Delta_q^{\text{IN}}(p) - D_p \mid \\ \forall q \in Q,\ \forall p \in Q \text{ with } \Delta_q^{\text{IN}}(p) > -\infty\} \end{aligned} \tag{5}$$

Specifically, for all $q \in Q$, we add an edge from the preceding critical operations $p \in Q$, for which $\Delta_q^{\text{IN}}(p) > -\infty$, to q. The corresponding edge delay d_{pq} is set to $\Delta_q^{\text{IN}}(p) - D_p$. We subtract D_p, as otherwise we would account for p's delay twice later.

Note that F_{cons} is smaller than the transitive hull of E restricted to Q, which would per definition connect all critical operations with each other. In contrast, in situations where all paths connecting two critical operations q_1, q_2 contain at least one other critical operation, our construction scheme would not add the redundant edge $(q_1 \to q_2)$.

Figure 3 shows F_{cons} corresponding to the running example. Where specified, the edge labels now represent a non-zero edge delay, e.g. $d_{2\ 9} = 2$. For all other edges, the edge delay remains 0.

c) Edge filtering: On the reduced graph (Q, F_{cons}), it is now tractable to compute all-pair-longest-path information $\text{lp}(u, v)$ for pairs of operations $u, v \in Q$ using the Floyd-Warshall algorithm [17], which we use to filter out superfluous direct edges whose implied precedence constraints are satisfied transitively by other edges: an edge $(u \to v) \in F_{\text{cons}}$ is removed iff $\delta_{uv} < \text{lp}(u, v)$.

Finally, we compose the set of edges $F = F_{\text{cons}} \cup B$ in $\mathcal{I}'$ from the newly constructed and filtered forward edges, and the unmodified set of backedges B.

The final, reduced version of the running example is illustrated in Figure 4.

Complexity: In a single pass over E, we precompute the sets B, and $\text{pred}(v), \text{succ}(v)$ for all operations $v \in O$. Collecting all critical operations Q requires visiting all operations in O once. As we only consider forward edges when computing $\Delta_v^{\text{IN}}(q)$, the data flow analysis operates on an acyclic graph and reaches its fix point after handling each operation $v \in O$ (which requires inspecting the $|\text{pred}(v)|$ predecessors) once in topological order. Performing all these preparation steps is in $\mathcal{O}(|O| + |E|)$.

Computing F_{cons} has a worst-case runtime of $\mathcal{O}(|Q|^2)$, while the edge filtering step is in $\mathcal{O}(|Q|^3)$ due to the longest-path calculation. However, as HLS-style MSP instances are characterised by $|Q| \ll |O|$, the resulting runtimes are negligible compared to the runtime of the actual modulo scheduler even for the cubic parts of the reduction approach.

B. Modulo scheduling

The reduced problem instance $\mathcal{I}'$ comprises the sets Q and F, and inherits all other parameters from the original instance $\mathcal{I}$. We obtain a feasible interval λ° and start times t_q for all $q \in Q$ from solving $\mathcal{I}'$ with any modulo scheduler. In case the nested modulo scheduler does not find any solution to $\mathcal{I}'$, our approach will stop here and report the failure. To avoid the obvious name clash, we refer to the computed start times as s_q in the following section.

C. Schedule completion

Lastly, start times for the non-critical operations $i \in O \setminus Q$ have to be computed. To this end, we solve the following ILP, defined for integer variables t_i:

$$\textbf{min } T \tag{6}$$

$$\text{s.t. } t_i + \delta_{ij} \ \leq \ t_j + \beta_{ij} \cdot \lambda^\circ \quad \forall (i \to j) \in E \tag{7}$$

$$t_i = s_i \quad \forall i \in Q \tag{8}$$

$$t_i + D_i \leq T \quad \forall i \in O \tag{9}$$

Together with λ° as computed in the previous step, the start times t_i, $i \in O$, yield a complete solution to the orignal problem instance $\mathcal{I}$.

Complexity: The ILP above can easily be transformed into a system of difference constraints (SDC) [7]. SDCs comprise a special class of ILPs that are optimally solvable in polynomial time, due to the fact that their LP relaxation is guaranteed to produce an integer-valued solution.

IV. Experimental evaluation

We evaluate our reduced-complexity modulo scheduling approach on a set of realistic test instances: C applications from the HLS benchmark suites CHStone [18] and MachSuite [19] are compiled with Clang[3] to LLVM-IR [20]. Nymble [5] constructs per-loop control-data-flow graphs (CDFG), where the original control flow is replaced by multiplexers and predicated operations. Nested loops become special operations in the graph, thus making it possible to modulo schedule all loops in the application instead of being limited to the most deeply nested ones. Nymble also constructs edges to retain the sequential order amongst memory access operations where needed, as determined by LLVM's dependence analysis. We use operator latencies and physical delays from the Bambu

[3]We use LLVM/Clang version 3.3, optimisation preset "-O2" with loop unrolling disabled, and perform exhaustive inlining.

HLS framework's [21] extensive operator library for a Xilinx xc7vx690 device. For each operator, we choose the lowest-latency variant that is estimated to achieve a frequency of at least 250 MHz. The edges to limit operator chaining are constructed with a simple path-based approach similar to [6] and aim to enforce a maximum cycle time of 5 ns. From a total set of 354 loops, we selected 21 unique instances that took more than 10 seconds to schedule with at least one scheduler configuration. The MSP instances use the following resource types: memory load (2 available)/store (1), nested loop (1), integer division (8), floating-point addition (4), FP subtraction (4), FP multiplication (4), other FP operations (2 each).

In the second step (Section III-B) of our approach, we delegate the actual modulo scheduling to the ED formulation, defined by (1), (2), (5) and (20) in [8], and to the Moovac formulation according to Figure 3 in [4]. Both formulations were adapted to minimise the schedule length.

We use CPLEX 12.6.3 to solve the ILPs constructed according to the ED and Moovac formulations, as well as for the ILP used in the schedule construction step (Section III-C). We set a time limit of 3 minutes (minus the time to construct the linear program via the solver's API) per scheduling attempt/candidate II, and try at most 20 candidate IIs per instance, thereby capping its total runtime to one hour. The solver uses deterministic multithreading with up to 8 threads. We ran up to two scheduling jobs concurrently [22] on 2x12-core Intel Xeon E5-2680 v3 systems running at 2.8 GHz with 64 GB RAM. The scheduler runtimes presented here correspond to the best (in terms of the computed II, the schedule length, and lastly, the scheduler runtime) of three runs.

Table I shows the effects of our approach on the problem complexity. The instances are named according to the CHStone/MachSuite benchmark application[4] they originated from, and distinguished by an arbitrary loop ID assigned by the Nymble compiler. The complexity of the original instance $\mathcal{I}$ is specified by the number of operations $|O|$, edges $|E|$, resource-limited operations $|L|$, and backedges $|B|$. Recall that $L \subseteq O$ and $B \subseteq E$. For example, instance aes2::2 has 155 operations, of which 34 are resource-limited, as well as 291 edges in total, of which 26 are backedges. The complexity of the reduced instance is specified accordingly by the number of critical operations $|Q|$, and inferred edges $|F|$. By construction, the sets L and B are carried over unchanged to the reduced problem, and therefore their sizes are not repeated in the table. In the next column, we report the time required for the problem reduction, including all steps of our approach without the actual modulo scheduling with the delegate modulo scheduler.

The remaining columns show the effects of our proposed approach to the size of the ILPs, i.e. the number of decision variables $N_{\mathcal{X}}$ and the number of linear constraints $C_{\mathcal{X}}$, when applying either the ED[5] or the Moovac formulation to an instance $\mathcal{X}$. We present the absolute numbers for the original instance $\mathcal{I}$, and relative to that, the quantities for the reduced instance $\mathcal{I}'$. For example, the ILP according to the ED formulation for aes2::2 has roughly 3600 decision variables initially. After the problem reduction, only 26% remain, i.e. the ILP for the reduced instance requires about 936 variables. Overall, ED formulation ILPs are reduced to 15% of variables and 22% of constraints (excluding dfsin::1), whereas Moovac ILPs retain over 83% of their original sizes (geometric means).

In Table II, overall runtimes to schedule the original instances with and without our proposed reduced-complexity approach are shown alongside the computed initiation intervals λ° and the corresponding schedule lengths T_{λ°. Asterisks denote which parts of the solution were proven to be optimal. We distinguish four cases: "(*,*)" solutions are optimal according to both objectives, i.e. a minimum length schedule was found for the smallest feasible II. In a "(*,)" solution, the determined II is proven to be optimal, but the solver was not able find or prove the optimal schedule length. This typically means that the solver depleted its 3 min time budget, at whose end we accept any feasible solution. We only know that we have a feasible solution "(,)" if at least one scheduling attempt was aborted due to the time limit without finding a solution, and conservatively classified as infeasible. In the worst case, no solution was found for any candidate II, for which we note "-" in the table.

The results clearly show that our reduced-complexity modulo scheduling approach significantly speeds-up scheduling with the ED formulation. The accumulated runtime to schedule the 21 test instances is improved by 18%. The per-instance speed-ups are never below 1x, and range up to 14x, with a geometric mean of 4.37x. As a side effect, the solution quality for the largest instances is improved as well: the solver is able to turn feasible solutions into optimal ones (e.g. jpeg::19), or finding feasible solutions at all (e.g. jpeg::17). Compared to solver runtimes, the time spent in the problem reduction and schedule completion is negligible.

The distribution of the IIs in our set of test instances (cf. Table II) plays an important role in explaining these substantial performance gains. Considering that each operation in the ED formulation is represented by II-many binary variables, and each edge is expressed by II-many linear constraints, it is obvious (and evident in Table I) that any reduction in the input problem complexity will have a huge impact on the size of the constructed and solved ILP.

The accumulated runtime with the Moovac formulation improves marginally, but the per-instance speed-ups ranging from 0.2x to 1.33x indicate that Moovac is not amenable to be accelerated by our problem reduction approach. We attribute the small positive gains to the already efficient modelling of non-critical operations and edges in the Moovac formulation, as documented in Table I. Most of the performance regressions

[4]As both suites contain an application named aes, we refer to MachSuite's version as aes2.

[5]As the number of variables and constraints in the ED formulation is dependent on the candidate II, we present here the ILP complexity for the last (= successful) scheduling attempt.

TABLE I: Complexity-reduction results

	Original instance $\mathcal{I}$				Reduced instance $\mathcal{I}'$			ILP complexity: ED				ILP complexity: Moovac			
Instance	$\|O\|$	$\|E\|$	$\|L\|$	$\|B\|$	$\|Q\|$	$\|F\|$	time [s]	$N_{\mathcal{I}}$	$N_{\mathcal{I}'}/N_{\mathcal{I}}$	$C_{\mathcal{I}}$	$C_{\mathcal{I}'}/C_{\mathcal{I}}$	$N_{\mathcal{I}}$	$N_{\mathcal{I}'}/N_{\mathcal{I}}$	$C_{\mathcal{I}}$	$C_{\mathcal{I}'}/C_{\mathcal{I}}$
fft_strided::2	84	290	24	65	29	112	0.02	6.5k	35%	22.1k	40%	0.4k	85%	1.1k	84%
aes2::2	155	291	34	26	41	92	0.03	3.6k	26%	6.3k	32%	1.5k	92%	4.6k	96%
aes::2	177	377	29	25	34	81	0.02	8.7k	19%	18.1k	22%	0.7k	79%	1.9k	84%
gsm::3	194	313	18	12	43	120	0.03	2.3k	22%	3.4k	36%	0.5k	72%	1.3k	86%
aes2::4	225	683	62	158	69	285	0.05	7.9k	31%	22.6k	42%	3.8k	96%	12.4k	97%
aes::15	236	439	32	2	37	81	0.03	3.5k	16%	6.0k	18%	1.4k	85%	4.1k	91%
md_grid::7	300	577	24	53	35	115	0.02	14.4k	12%	27.0k	20%	1.1k	75%	3.1k	85%
jpeg::87	380	1574	35	102	40	171	0.04	15.6k	11%	61.1k	11%	1.6k	78%	5.4k	74%
jpeg::63	382	1577	35	102	40	171	0.04	15.7k	10%	61.2k	11%	1.6k	78%	5.4k	74%
jpeg::47	383	1578	35	102	40	171	0.04	15.7k	10%	61.2k	11%	1.6k	78%	5.4k	74%
jpeg::59	471	1919	50	223	55	308	0.05	23.6k	12%	91.8k	16%	2.4k	83%	8.2k	80%
jpeg::19	476	1609	69	363	74	489	0.07	26.2k	16%	85.3k	30%	4.8k	92%	16.2k	93%
jpeg::41	480	1956	50	224	55	309	0.05	24.5k	11%	95.5k	16%	2.4k	82%	8.3k	80%
adpcm::2	710	1478	100	82	105	388	0.12	37.6k	15%	76.2k	26%	9.3k	93%	30.6k	96%
adpcm::1	777	1746	95	141	106	396	0.11	50.5k	13%	110.8k	21%	8.4k	92%	27.5k	95%
blowfish::1	789	2558	107	42	118	517	0.12	90.7k	15%	289.1k	20%	18.6k	96%	63.8k	97%
jpeg::17	942	5047	106	1041	111	1210	0.15	105.5k	11%	553.3k	23%	9.5k	91%	33.9k	89%
mips::1	1076	4441	65	1155	76	1595	0.09	34.4k	6%	132.1k	32%	5.7k	82%	19.9k	86%
aes::12	1367	3065	205	532	210	911	0.27	128.5k	15%	283.3k	29%	45.8k	97%	156.5k	99%
aes::4	1374	2816	205	402	212	785	0.31	129.2k	15%	260.7k	28%	46.3k	97%	158.0k	99%
dfsin::1	2651	38642	67	1222	76	1402	0.10	$N_{\mathcal{I}'} = 16.7\text{k}, C_{\mathcal{I}'} = 305.3\text{k}$ †				6.7k	62%	52.3k	29%

† Timeout during ILP construction for original instance.

TABLE II: Scheduling results

	ED (standard)			ED (reduced)			ED	Moovac (standard)			Moovac (reduced)			Moovac
Instance	time [s]	λ°	T_{λ°	time [s]	λ°	T_{λ°	speed-up	time [s]	λ°	T_{λ°	time [s]	λ°	T_{λ°	speed-up
fft_strided::2	45.04	*74	*77	13.04	*74	*77	3.45	0.54	*74	*77	0.80	*74	*77	0.67
aes2::2	26.77	*20	*21	4.70	*20	*21	5.69	37.24	*20	*21	27.00	*20	*21	1.38
aes::2	18.60	*46	*48	2.92	*46	*48	6.37	2.97	*46	*48	3.35	*46	*48	0.89
gsm::3	0.45	*9	*20	0.25	*9	*20	1.82	36.34	*9	*20	180.03	*9	20	0.20
aes2::4	1277.49	32	33	703.53	32	33	1.82	2520.00	33	33	2700.04	34	33	0.93
aes::15	2.04	*12	*18	0.29	*12	*18	6.99	180.00	*12	18	180.02	*12	18	1.00
md_grid::7	15.80	*45	*45	2.20	*45	*45	7.19	6.20	*45	*45	5.75	*45	*45	1.08
jpeg::87	16.26	*38	*47	1.22	*38	*47	13.33	0.28	*38	*47	0.48	*38	*47	0.59
jpeg::63	16.33	*38	*47	1.22	*38	*47	13.41	0.25	*38	*47	0.48	*38	*47	0.52
jpeg::47	15.88	*38	*47	1.27	*38	*47	12.51	0.28	*38	*47	0.48	*38	*47	0.58
jpeg::59	42.08	*47	*55	3.12	*47	*55	13.47	1.01	*47	*55	1.10	*47	*55	0.92
jpeg::19	180.00	*52	61	18.27	*52	*54	9.85	1.53	*52	*54	4.65	*52	*54	0.33
jpeg::41	48.20	*48	*56	3.40	*48	*56	14.17	1.05	*48	*56	1.21	*48	*56	0.87
adpcm::2	180.00	*50	132	39.59	*50	*85	4.55	180.00	*50	94	180.10	*50	92	1.00
adpcm::1	1620.00	62	94	758.32	58	72	2.14	1440.00	61	81	1080.11	59	73	1.33
blowfish::1	3600.00	-	-	3600.12	-	-	1.00	3600.00	-	-	3571.82	-	-	1.01
jpeg::17	1080.00	-	-	86.08	*104	*105	12.55	8.93	*104	*105	11.47	*104	*105	0.78
mips::1	1077.93	29	34	398.66	26	38	2.70	2160.00	35	34	1980.08	34	34	1.09
aes::12	3600.00	-	-	3600.27	-	-	1.00	3600.00	-	-	3600.29	-	-	1.00
aes::4	3600.00	-	-	3600.31	91	92	1.00	3600.00	-	-	3600.27	-	-	1.00
dfsin::1	3240.00	-	-	3240.10	-	-	1.00	24.28	*201	*216	24.89	*201	*216	0.98
sum [min]	328.38			267.98				290.02			285.91			
geomean							4.37							0.80

λ° = computed initiation interval, T_{λ° = corresponding schedule length. Asterisks (*) denote optimality.

occur on reduced instances that are already scheduled in very short runtimes in their original versions, and can be considered negligible in practice. However, instances gsm::3 and aes2::4 incur a loss of solution quality. We believe that this is caused by a phenomenon called *performance variability* which is inherent to ILP solvers. In a nutshell, even small structural changes to linear programs may significantly influence the solver runtime [23]. As removing redundant constraints and decision variables is at the core of our approach, it is susceptible to this effect.

V. Conclusion and outlook

We presented a complexity-reduction approach intended to be used in conjunction with exact modulo schedulers whose internal structure does not scale well with the size and density of dependence graphs typical for HLS-style MSP instances. Applied to the ILP formulation by Eichenberger and Davidson, we were able to unconditionally speed up all instances in our benchmark set, and obtain better quality solutions for the largest loops.

With our new preprocessing of the dependence graphs, the ED and Moovac formulations are much closer performancewise on HLS-typical MSP instances. However, the two schedulers still exhibit different strengths and weaknesses, which make each of them better suited for a certain set of MSP instances. We plan to capitalise on that by building an oracle to decide a-priori which scheduler is most promising for a given MSP.

Our experiment also suggests that none of the quantities in Table I individually will be sufficient to predict whether a given instance constitutes a hard problem for the formulations. To this end, we plan to develop a configurable MSP generator to investigate further which additonal structural properties have a significant influence on the scheduler runtimes, and to guide the development of the aforementioned oracle.

Acknowledgment

Calculations for this research were conducted on the Lichtenberg high performance computer of TU Darmstadt.

References

[1] 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 2002, New York City, NY, USA, June 22-26, 2002*. ACM, 2002, pp. 97–106.

[2] M. Ayala, A. Benabid, C. Artigues, and C. Hanen, "The resource-constrained modulo scheduling problem: an experimental study," *Comp. Opt. and Appl.*, vol. 54, no. 3, pp. 645–673, 2013.

[3] A. Canis, S. D. Brown, and J. H. Anderson, "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, 2014, pp. 1–8.

[4] J. Oppermann, A. Koch, M. Reuter-Oppermann, and O. Sinnen, "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, 2016, pp. 1:1–1:10.

[5] J. Huthmann, B. Liebig, J. Oppermann, and A. Koch, "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, 2013, pp. 1–8.

[6] C. Hwang, J. Lee, and Y. Hsu, "A formal approach to the scheduling problem in high level synthesis," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 10, no. 4, pp. 464–475, 1991.

[7] J. Cong and Z. Zhang, "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*. ACM, 2006, pp. 433–438.

[8] A. E. Eichenberger and E. S. Davidson, "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*. ACM, 1997, pp. 194–205.

[9] A. E. Eichenberger, E. S. Davidson, and S. G. Abraham, "Author retrospective for optimum modulo schedules for minimum register requirements," in *ACM International Conference on Supercomputing 25th Anniversary Volume*. ACM, 2014, pp. 35–36.

[10] P. J. Billington, J. O. McClain, and L. J. Thomas, "Mathematical programming approaches to capacity-constrained MRP systems: review, formulation and problem reduction," *Management Science*, vol. 29, no. 10, pp. 1126–1141, 1983.

[11] B. R. Rau, "Iterative modulo scheduling," *International Journal of Parallel Programming*, vol. 24, no. 1, pp. 3–65, 1996.

[12] R. A. Huff, "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*. ACM, 1993, pp. 258–267.

[13] J. Llosa, E. Ayguadé, A. González, M. Valero, and J. Eckhardt, "Lifetime-sensitive modulo scheduling in a production environment," *IEEE Trans. Computers*, vol. 50, no. 3, pp. 234–249, 2001.

[14] B. D. D. Dinechin, "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*, 2007, pp. 144–151.

[15] A. Bonfietti, M. Lombardi, L. Benini, and M. Milano, "CROSS cyclic resource-constrained scheduling solver," *Artif. Intell.*, vol. 206, pp. 25–52, 2014.

[16] M. Ayala and C. Artigues, "On integer linear programming formulations for the resource-constrained modulo scheduling problem." Archive ouverte HAL, Tech. Rep. LAAS no. 10393, 2010. [Online]. Available: https://hal.archives-ouvertes.fr/hal-00538821

[17] R. W. Floyd, "Algorithm 97: Shortest path," *Commun. ACM*, vol. 5, no. 6, p. 345, 1962.

[18] 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," *JIP*, vol. 17, pp. 242–254, 2009.

[19] B. Reagen, R. Adolf, Y. S. Shao, G. Wei, and D. M. Brooks, "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, 2014, pp. 110–119.

[20] C. Lattner and V. S. Adve, "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, 2004, pp. 75–88.

[21] C. Pilato and F. Ferrandi, "Bambu: A modular framework for the high level synthesis of memory-intensive applications," in *23rd International Conference on Field programmable Logic and Applications, FPL 2013, Porto, Portugal, September 2-4, 2013*. IEEE, 2013, pp. 1–4.

[22] O. Tange, "Gnu parallel - the command-line power tool," *;login: The USENIX Magazine*, vol. 36, no. 1, pp. 42–47, Feb 2011. [Online]. Available: http://www.gnu.org/s/parallel

[23] A. Lodi and A. Tramontani, "Performance variability in mixed-integer programming," in *Theory Driven by Influential Applications*. INFORMS, 2013, pp. 1–12.