Isomorphic Subgraph-based Problem Reduction for Resource Minimal Modulo Scheduling

Patrick Sittel, Nicolai Fiege
University of Kassel
{sittel,fiege}@uni-kassel.de

Martin Kumm
University of Applied Sciences Fulda
martin.kumm@informatik.hs-fulda.de

Peter Zipf
University of Kassel
zipf@uni-kassel.de

***Abstract*—Modulo scheduling is a powerful method to increase throughput in high-level synthesis for digital hardware design. When facing large designs, optimal approaches are likely to time out and heuristics fail to provide satisfying throughput and latency. We propose an isomorphic subgraph-based reduction of the input data-flow graph (DFG) that is applied before scheduling, in order to solve the modulo scheduling problem faster without changing the optimal initiation interval (II) and allocated hardware. Our results show a solving time speedup of 5× on average and up to 102× for large designs. Using the proposed pre-processing step, the II achieved could be reduced by 33% on average for SDC-based modulo schedulers. And in ILP-based scheduling, we could classify 15% more solutions as optimal within the same time compared to solutions provided without applying our transformation.**

***Index Terms*—Modulo Scheduling, High-level Synthesis, Design-space Exploration**

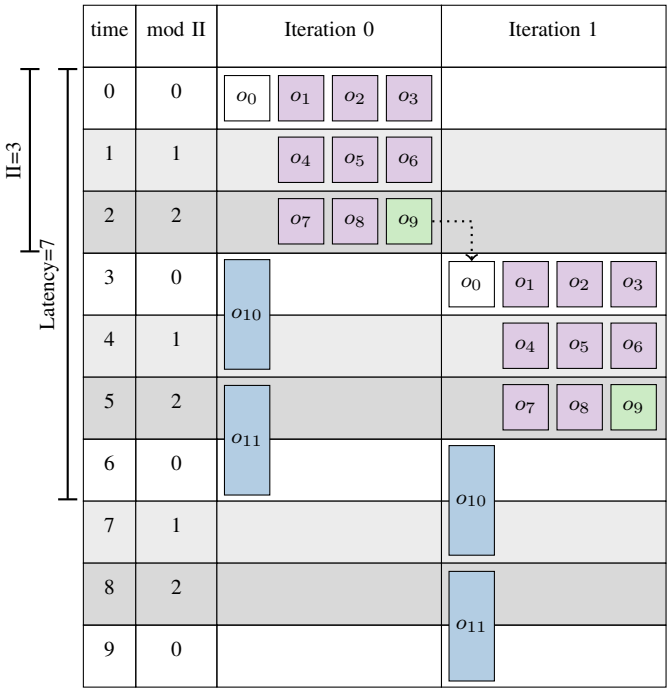
I. Introduction

Computer-aided transformation of algorithmic descriptions into hardware as used in high-level synthesis (HLS) [12], algorithmic synthesis and model-based hardware design has enabled software developers to effectively use field-programmable gate arrays (FPGAs) for accelerating costly computations. A popular approach to increase throughput is the use of loop pipelining, which can be achieved by solving the modulo scheduling problem [17].

The initiation interval (II) of a schedule is the number of clock cycles after which new data, i.e., from the next loop iteration, is inserted. This idea is depicted in Table I that shows the first two iterations of a modulo schedule with II=3 for the DFG shown in Figure 1. The operations that have to be performed are described as o_i. Operations of the same color relate to the same type of operation and therefore the same type of hardware unit. Instead of waiting for iteration 0 to be complete in cycle six (latency = 7), iteration 1 (and all subsequent ones) can already start three cycles after the start of its predecessor, thus increasing throughput significantly.

In steady state operation, the throughput achieved is close to the reciprocal of the II. Therefore, minimizing the II is the first priority objective. Many approaches to solve the modulo scheduling problem for digital hardware design have been proposed using integer linear programming (ILP) [10], [14], system of difference constraints (SDC) [4] and Boolean satisfiability (SAT) [9], [7]. ILP-based approaches are able to identify the optimal II and latency, but tend to timeout in practice when facing large DFGs, resulting in suboptimal IIs. Heuristic formulations that are SDC-based or integrate SAT renounce the ability to minimize latency in order to enable quick identification of optimal IIs. Recent work on heuristic approaches has shown significant improvement on speeding up the solving process. But compared to optimal formulations, the latency achieved has been up to three times worse for small benchmarks with less than 100 vertices [9]. Scalability still represents the main problem in modulo scheduling and we present an approach to reduce problem complexity before scheduling, thus improving solving time for both optimal and heuristic approaches.

TABLE I
A modulo schedule (II=3, latency=7) of the example data-flow graph in Figure 1.



The theoretical minimum II ($\text{II}^{\perp}$) is restricted by resource and inter-iteration dependencies (recurrence) constraints. The minimum recurrence constrained II ($\text{II}^{\perp}_{\text{rec}}$) is inherent to the input problem. In the example DFG in Figure 1, $\text{II}^{\perp}_{\text{rec}}$ is introduced by the edge between o_9 and o_0 that is labeled with a distance of one and represents that o_0 requires the output of

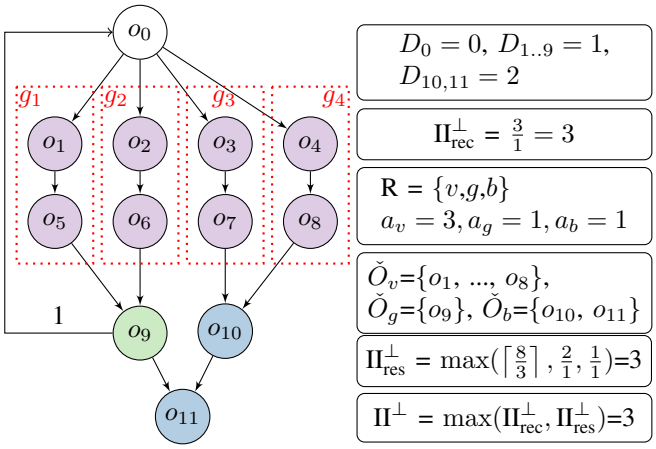


Fig. 1. Example data-flow graph and resource allocation.

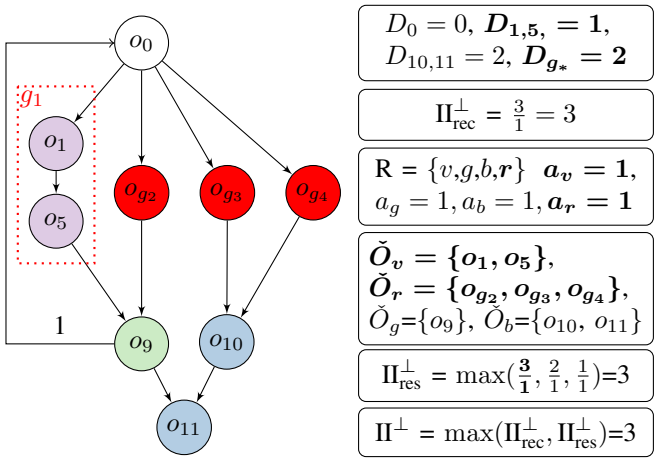


Fig. 2. DFG and allocation from Fig. 1 after the proposed transformation.

o_9 calculated one iteration earlier. The operations $o_{1..9}$ have a delay of one cycle ($D_{1..9} = 1$) and o_0 is an inserted artificial operation with a delay of zero cycles ($D_0 = 0$). Due to the two recurrences $\{o_0, o_1, o_5, o_9\}$ and $\{o_0, o_2, o_6, o_9\}$, we have $\text{II}^{\perp}_{\text{rec}} = \lceil \frac{0+1+1+1}{1} \rceil = 3$ (formal definition is given in Section III-B).

The minimum resource constrained II ($\text{II}^{\perp}_{\text{res}}$) is based on the number of functional units (FUs) allocated and can be used for design space exploration (DSE). In the example in Figure 1, the violet operations o_1 to o_8 are of resource type v ($\check{O}_v$=$\{o_1$, ..., $o_8\}$). Lets suppose for this example, the resource type v has a limit of a_v=3 functional units which leads to $\text{II}^{\perp}_{\text{res}} = 3$. Reducing a_v to two would save one functional unit, but would lead to $\text{II}^{\perp}_{\text{res}} = 4$. Resources g (o_9) and b (o_{10}, o_{11}) have been assigned a limit of one ($a_g = a_b = 1$). Note that at no time slot modulo three, more than the number of allocated hardware units (3/1/1) is scheduled.

The addition of available resources relates to improving throughput (until $\text{II}^{\perp}_{\text{rec}}$ is the limiting factor). Therefore, resource allocation can be used for DSE. This introduces an exponentially growing design space that is often traversed partially using iterative methods or heuristic models [19]. A problem-specific approach is scheduler-driven DSE that discards dominated resource allocations, thus speeding up the process significantly [15].

The main shortcoming of modulo scheduling and DSE are the lack of scalability. An open question that we address by using $\text{II}^{\perp}$ for subgraph combination is how graph reduction techniques can be combined with scheduler-driven DSE to speed up solving time. The contributions of this work are:

i) An isomorphic-subgraph based transformation of the DFG to significantly reduce solving time in modulo scheduling problems without changing the optimal II or adding additional FUs.
ii) A search-space pruning method for isomorphic subgraph mining and combination.
iii) A heuristic for design-space exploration.
iv) An evaluation of the proposed method on real circuits from digital signal processing.

II. Motivational Example

This work proposes a method to reduce the number of resource constrained and unconstrained vertices to be scheduled by using isomorphic subgraphs. A formal definition regarding isomorphic subgraphs is provided in Section V. As an example, four isomorphic occurrences ($g_1 - g_4$) of the same subgraph in the DFG are highlighted by red boxes in Figure 1. Figure 2 shows the graph and resource allocation after the proposed transformation was applied. All changes that were made to the resource model are highlighted in bold.

The proposed graph transformation is applied using a subset of these occurrences ($g_2 - g_4$). The vertex pairs $\{o_2, o_6\}$, $\{o_3, o_7\}$ and $\{o_4, o_8\}$ (in Figure 1) were replaced with vertices that represent the respective subgraph occurrences (see Figure 2). Then, as each subgraph contains two operations of resource type v, the allocated hardware units were reduced by two to $a_v = 1$. To compensate this, one FU of the new resource type r (for red vertices) has been added. The latency of the new operations is two cycles as they consist of two succeeding operations of latency one. In general, the used occurrences have to be scheduled using a non modulo scheduler, e.g. as-soon-as possible (ASAP), before they can be replaced.

By replacing exactly three (and not all four) subgraph occurrences with single vertices and adding a new resource type, we achieve that $\text{II}^{\perp}_{\text{res}}$, $\text{II}^{\perp}_{\text{rec}}$, $\text{II}^{\perp}$ and the allocated hardware units remain unchanged. Note that one unit of the new resource type r is a replacement for exactly two units of resource type v that was removed in the new resource model.

We are able to achieve this by excluding g_1 from the transformation. If g_1 had been used too, all operations of resource type v would have been removed from the resource model. In the case that all four operations of resource type r would have to be implemented, $\text{II}^{\perp}_{\text{res}}$ and $\text{II}^{\perp}$ would increase to four, lowering the optimal throughput achievable. The transformation does not change $\text{II}^{\perp}$ and the number of hardware units allocated of every resource type whenever

$$\#\text{occurrences} \mod \text{II}^{\perp} = 0 \qquad (1)$$

holds. This is true, because (1) ensures 100% resource utilization of all hardware units used in this specific subgraph. This observation is crucial to speed up the identification and combination of isomorphic subgraphs as it allows us to prune large portions of the complete design space.

III. Background

This section provides theoretical background and definitions regarding modulo scheduling and the use of isomorphic subgraphs in the context of modulo scheduler-driven DSE. An overview is provided in Table II.

A. Prerequisites

As is common for scheduling in the context of hardware synthesis, we consider the problem description to be provided as data-flow graph (DFG) $G = (O, E, l)$ where operations $o_i \in O$ are connected by directed edges $(o_i, o_j) \in E$ that represent an integer dependence distance d_{ij}. All forward edges get assigned a distance of zero. Edges that relate to data calculated n iterations before, have a distance of n.

For modelling hardware properties, a set of resource types R (e.g., add, mult,...) that carries all scheduler relevant properties (e.g., latency, blocking time,...) of the respective hardware implementations is used. Via the labeling function l, every o_i is registered to one resource type $\tau \in R$ that implements the respective operation, thus assigning a latency D_i in clock cycles. This latency is used by the modulo scheduler to ensure a valid flow of data. To make the scheduler aware of hardware restrictions, the number of available functional units of type τ can be limited to $\mathrm{FUs}(\tau)$.

In general, it is assumed that every τ is fully-pipelined and able to accept new input data every clock cycle. It follows that at most $\mathrm{FUs}(\tau)$ operations of resource type τ can be scheduled in time steps of the same congruence class (time step modulo II). Resources that are not critical in hardware can be declared unlimited, which simplifies the scheduling problem.

B. Modulo Scheduling

Using a fixed resource allocation, the most common main design goal in modulo scheduling is to find a schedule that enables the smallest II possible. Secondary objectives can be the minimization of latency or required lifetime registers.

The maximum throughput that can be achieved depends both on resource constraints and on recurrences (i.e., edges with dependence distance > 0) in the DFG. The example in Figure 1 has one recurrence with a dependence distance of one and a latency of three cycles, so the II must not be less than $\frac{3}{1}$. This is called the *recurrence-constrained minimum II* [4], written $\mathrm{II}^{\perp}_{\mathrm{rec}}$. In general, we have

$$\mathrm{II}^{\perp}_{\mathrm{rec}} = \max_{i \in \text{ recurrences}} \left\lceil \frac{\text{latency}_i}{\text{distance}_i} \right\rceil \tag{2}$$

where latency_i and distance_i give the latency and distance of the i^{th} recurrence.

Moreover, because this example has eight type v, two type b and one type g operations, the II must also not be less than $8/\mathrm{FUs}(v)$, $2/\mathrm{FUs}(b)$ and $1/\mathrm{FUs}(g)$, where $\mathrm{FUs}(\tau)$ is the number of functional units that can execute operations of type τ. This is called the *resource-constrained minimum II*:

$$\mathrm{II}^{\perp}_{\mathrm{res}} = \max_{\tau \in \text{ resources}} \left\lceil \frac{\#\tau}{\mathrm{FUs}(r)} \right\rceil \tag{3}$$

where $\#\tau$ is the number of operations in the DFG of type τ.

Then, the recurrence-constrained minimum II and the resource-constrained minimum II, as defined in (2) and (3), are used to estimate a lower bound for the II [17]. This minimum II is defined as:

$$\mathrm{II}^{\perp} = \max(\mathrm{II}^{\perp}_{\mathrm{res}}, \mathrm{II}^{\perp}_{\mathrm{rec}}). \tag{4}$$

TABLE II
Notation overview

Notation	Meaning
O	Set of operations
E	Set of edges
R	Set of all Op. types
$\tau \in R$	Op. types (e.g., add, mult, ...)
$l : l(o_i) = \tau$	Labeling function for $o_i \in O \to \tau \in R$
$d_{ij} \in \mathbb{N}_{\geq 0}$	Dependence distance on $o_i \to o_j$
$\check{O} \subseteq O$	Resource-constrained operations
$\mathrm{FUs}(\tau) \in \mathbb{N}_{\geq 1}$	No. of instances of type $\tau \in R$
$\check{O}_\tau \subseteq \check{O}$	Set of resource-constrained operations of type $\tau \in R$, i.e., $\bigcup_{\tau \in R} \check{O}_\tau = \check{O}$
$D_i \in \mathbb{N}_{\geq 0}$	Latency of operation $o_i \in O$
$L \in \mathbb{N}_{\geq 0}$	Maximal latency constraint
$\mathrm{II}^{\perp}_{\mathrm{res}}$	Minimum resource constrained II
$\mathrm{II}^{\perp}_{\mathrm{rec}}$	Minimum recurrence constrained II
$\mathrm{II}^{\perp}$	Minimum II
II^{X}	Candidate II

IV. Related Work

Modulo scheduling is a multi-objective optimization problem, e.g., minimizing both II and latency, where identification of the smallest II possible is considered to be the main objective [14]. To simplify the problem, Rau proposed *iterative modulo scheduling* where the II is given as an input to the solver, and is incremented after each unsuccessful attempt [17]. The iteration starts using an estimated minimum value for the II, and then increments this whenever a solving process fails. The approaches to solve the modulo scheduling problem can be classified into exact ones that are capable of computing optimal solutions regarding II and often a target-dependent secondary objective (e.g., [10], [14]), and heuristic ones that cannot guarantee optimality and are chosen for shorter run times (e.g., [4], [9], [7]).

We introduce a pre scheduling technique that helps to reduce solving time regardless of the used scheduler by reducing the size of the DFG using occurrences of isomorphic subgraphs. In order to apply this approach for resource sharing during HLS, Cong and Jiang proposed a subgraph mining heuristic that restricts the maximum vertex size of patterns [6]. In general, isomorphic subgraph mining is NP-complete [22]. By following the idea of Cong and Jiang, we are able to

significantly reduce the run time of the mining algorithm for our experiments that are discussed in Section VIII.

Sittel *et al.* improve on the work of Cong and Jiang [6] by formulating the problem of subgraph occurrence combination and subgraph-based scheduling for custom hardware design [20]. We follow the authors in their approach of combining occurrences and folding cores by using maximum independent sets. But, we improve on Cong and Jiang by supporting modulo scheduler-driven design space exploration (see Section VI) and on both approaches by including context specific information from resource allocation and modulo scheduling for isomorphic subgraph mining and combination.

To the best of our knowledge, we are the first to introduce a method that uses the concept of isomorphic subgraphs for graph reduction for modulo scheduling and scheduler-driven design-space exploration. Doing this, we are able to improve throughput compared to state-of-the-art approaches that use non modulo schedulers. Additionally, we are the first to utilize $\text{II}^{\perp}$ for subgraph mining to reduce runtime without pruning relevant parts of the design space.

V. Isomorphic Subgraphs

The proposed transformation utilizes embeddings of isomorphic subgraphs into the DFG and applies a graph reduction technique for modulo scheduling problems. This section introduces the used nomenclature and definitions.

Intuitively, an occurrence is an embedding of a pattern into a graph. In Figure 1, four occurrence (g_1, g_2, g_3, g_4) of the pattern that describes two operations of resource type v connected by one edge are highlighted using red boxes.

Definition 1: Let $G = (O, E, l)$ denote the DFG representation of the input model, consisting of a set of operations O, directed edges $e \in E$ and a labeling function l for operations, denoting operator types. Every connected $p = (O_p, E_p, l_p)$ is called a **pattern** occurring in G if there exists a function $\rho : O_p \rightarrow O_G$ mapping the operations of p to the according operations of G such that $(i) \forall\, o \in O_p : l_p(o) = l_G(\rho(o))$ and $(ii) \forall\, (o_i, o_j) \in E_p : (\rho(o_i), \rho(o_j)) \in E_G$. Then every subgraph $\delta^p = (O_{\delta^p} \subseteq O_G, E_{\delta^p} \subseteq E_G, l_G)$ in G such that $(i) \forall\, o \in O_{\delta^p} : l_p(o) = l_G(\rho(o))$ and $(ii) \forall\, (o_i, o_j) \in E_p : (\rho(o_i), \rho(o_j)) \in E_{\delta^p}$ is defined as an **occurrence of p** in G.

A pattern is considered to be frequent in G if it has multiple occurrences, i.e., a user-defined integer threshold, in G [23]. Applying a frequent subgraph mining algorithm, the set of all frequent patterns $\mathcal{P} = \{p_i \mid i = 1, ..., n\}$ and the set of all occurrences $\Delta = \{\Delta^{p_i} \mid p_i \in \mathcal{P}\}$, where Δ^{p_i} contains all occurrences of p_i in G, are identified.

For our transformation, all occurrences used are required to be non overlapping. This is due to the fact that one operation can't be used in different occurrences without changing the algorithm described by the original DFG.

Definition 2: Two occurrences δ_j, δ_k of arbitrary patterns are defined as **compatible** or **conflict free** if and only if the intersection of their node set is empty, i.e. $V_{\delta_j} \bigcap V_{\delta_k} = \emptyset$.

In other words, two occurrences δ_j and δ_k are compatible when they don't share any operations. This can be modelled using an undirected compatibility graph, where occurrences are nodes and compatible occurrences are connected by edges. A clique in such a graph represents a set of compatible occurrences.

Definition 3: The **frequency** S_{min} (or minimum support) of a pattern p in G is defined as the maximum number of compatible occurrences of p in G, i.e. a maximal clique in the compatibility graph.

Following Bringmann and Nijssen [3], we use maximal cliques in order to combine isomorphic occurrences of patterns to reduce the number of operations in the DFG.

Definition 4: Given a pattern p in G and a set of all occurrences Δ^p of p in G, we denote each subset $\Delta^p_k \subseteq \Delta^p$, with $\mid \Delta^p_k \mid \geq 2$, of pairwise compatible occurrences as a **folding core**.

In this work, we show how sets of compatible occurrences (folding cores) can be used for replacing original operations with placeholder operations, thus reducing the number of operations in the DFG, without violating resource constraints or changing $\text{II}^{\perp}$.

Definition 5: The size $N^p_k = \mid \Delta^p_k \mid$ of the k-th folding core Δ^p_k of a pattern p is called the **folding factor** of Δ^p_k.

The folding factor describes how many occurrences of a pattern can be replaced using placeholder operations.

Definition 6: Given any two patterns p_1, p_2 in G and two sets of all occurrences Δ^{p_1}, Δ^{p_2}. Then, two folding cores $\Delta^{p_1}_k$ and $\Delta^{p_2}_l$ are defined as **compatible** if and only if all $\delta^{p_1}_m \in \Delta^{p_1}_k$ and $\delta^{p_2}_n \in \Delta^{p_2}_l$ are pairwise compatible.

In other words, a set of occurrences of different subgraphs can only be used in combination when all occurrences in the set are pairwise conflict free.

Definition 7: A set ϕ of pairwise compatible folding cores is denoted as a **folding core combination**.

It is possible to replace occurrences of a folding core combination with their respective placeholder operations. Whenever more than one operation was replaced, a reduction of the DFG has been achieved. Using exact subgraph mining and combination algorithms, all possible folding core combinations can be identified. But, both problems are NP-complete. To address this, we describe our approach of mining and combining isomorphic subgraph occurrences for modulo scheduling that reduces the design space significantly in Section VII.

VI. Minimal Resource Allocation

In general, given one resource allocation, it is the goal of the modulo scheduler to identify the smallest II possible. But, the exploding number of possible resource allocations makes this approach impractical for design space exploration.

To address this problem, scheduler-driven design space exploration has been proposed as a method for identifying resource allocations that contribute to the Pareto frontier before solving the modulo scheduling problem [15]. The idea is to consider a set of IIs as input and to determine the minimal resource allocations that are able to support the respective IIs. We now describe how the Pareto frontier regarding II

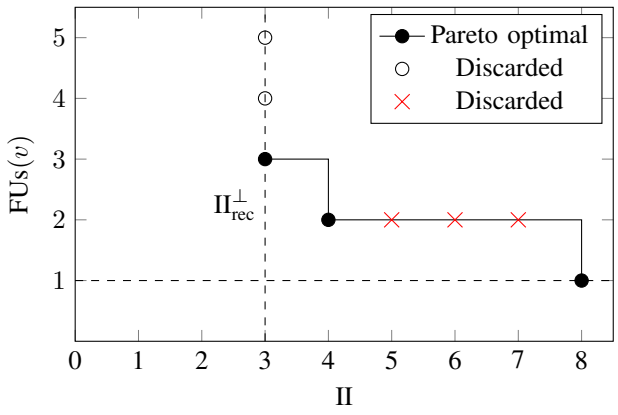


Fig. 3. Sketch of the Pareto frontier for the DFG from Figure 1. The vertical line highlights $\text{II}^{\perp}_{\text{rec}}$. The horizontal line shows the smallest resource allocation possible (one FU) for resource type v. The allocation for the resource types g and b has no influence on the II.

and functional units can be identified before scheduling, while using only a small fraction of all possible resource allocations.

The minimal (or trivial) resource allocation of any limited resource type τ can be calculated as follows [11], [15]:

$$a_\tau^{\text{II}^X} = \left\lceil \frac{|O_\tau|}{\text{II}^X} \right\rceil \quad \forall \tau \in R, \;\; 1 \leq a_\tau^{\text{II}^X} \leq |O_\tau|, \tag{5}$$

where II^X is the II that the desired allocation should support using the minimal amount of FUs over all resource types τ and $|O_\tau|$ is the amount of operations in the DFG that are implemented using τ. In other words, (5) allocates precisely as many FUs as are required to assign every operation in the DFG a time step modulo II^X without violating resource constraints.

Let us consider the example DFG from Figure 1. The design space for this example is depicted in Figure 3. For this problem, we have $\text{II}^{\perp}_{\text{rec}} = 3$ and it follows that $\text{II}^X \geq 3$. We can directly conclude that (5) will allocate exactly one FU of the resource types g and b, because $\lceil \frac{1}{3} \rceil = \lceil \frac{2}{3} \rceil = 1$ and $|O_g| = 1$, $|O_b| = 2$.

Therefore, the design space can be depicted using the allocated FUs(v). All allocations where $a_v^{\text{II}^X} \geq 4$ are discarded from the design space due to (5). The three minimal resource allocations of resource type v that contribute to the Pareto frontier are $a_v^3 = 3$, $a_v^4 = 2$ and the absolute minimal resource allocation $a_v^8 = 1$. Three candidate IIs $\text{II}^X = 5, 6, 7$ can also be discarded as they are dominated by $\text{II}^X = 4$.

To summarize, the identification of the Pareto frontier regarding II and allocated FUs for the example DFG has been reduced to *three* problems using scheduler-driven design space exploration that utilizes (5) for resource allocation. In the following, we derive a method for reducing the number of operations in the DFG such that the determined minimal resource allocation for every candidate II^X remains unchanged in order to reduce modulo schedule solving time. Our approach can be used without scheduler interaction and is therefore applicable for all optimal and heuristic formulations.

Algorithm 1 graphReduction

Require: Input DFG G, initial resource allocation R
Ensure: Reduced DFG G_r, adapted resource allocation R_r
1: $\text{II}^{\perp} \leftarrow$ calcMinII(G, R)
2: $\mathcal{F} = \emptyset$ // a set of folding cores
3: $\mathcal{F} \leftarrow$ subGrMining(G, R, $\text{II}^{\perp}$)
4: $\mathcal{C} \leftarrow$ subGrCombination($\mathcal{F}$, $\text{II}^{\perp}$)
5: $\mathcal{S} = \emptyset$ // a fcc selection
6: $\mathcal{S} \leftarrow$ select($\mathcal{C}, G, R$)
7: $G_r, R_r \leftarrow$ transform($G, R, \mathcal{S}$)
8: **return** G_r, R_r

Algorithm 2 subGrCombination

Require: $\mathcal{F}$, $\text{II}^{\perp}$
Ensure: $\mathcal{C}$
1: $\mathcal{C} = \emptyset$ // folding core combinations
2: $\mathcal{F}_\nabla = \emptyset$ // folding cores
3: **for each** $p_i \in \mathcal{F}$ **do**
4: $\quad \Delta_0^{p_i}$ = clique(Δ^{p_i}, $\text{II}^{\perp}$)
5: $\quad \mathcal{F}_\nabla \leftarrow \mathcal{F}_\nabla \cup \Delta_0^{p_i}$
6: **end for**
7: $\mathcal{C}$ = allMaxCliques($\mathcal{F}_\nabla$)
8: **return** $\mathcal{C}$

VII. Problem Reduction

The input to our transformation is the original DFG and resource allocation. Algorithm 1 describes our transformation as pseudo code. Our approach returns the transformed DFG and modified resource allocation that can be used for modulo scheduling and hardware generation.

As motivated in Section II, our approach to use subgraph mining and combination for modulo scheduling is based on $\text{II}^{\perp}$ that is calculated in line 1 of Algorithm 1 using the methods described in Section III-B. Since $\text{II}^{\perp}$ is derived from recurrence and resource constraints, the input DFG G and the resource allocation R are required.

Next, we mine frequent subgraphs in the DFG and store the findings in a set of folding cores. This is done in line 3. Note that there exists a plethora of isomorphic subgraph mining algorithms in the literature (e.g., [23], [5]) and all of them are applicable for our method. We exclude edges with distance ≥ 1 from the mining process and plan to investigate the usage of the different frequent subgraph mining algorithms in this context in future work.

Note that $\text{II}^{\perp}$ is passed to the frequent subgraph mining algorithm in line 3. In Definition 3, we have introduced the minimum support or frequency of a frequent subgraph. It is a common method to reduce the search space in frequent subgraph mining by setting a certain minimum support [23]. In other words, a lower limit to the amount of times each enumerated subgraph has to occur in the DFG is set. We set the minimum support to $\text{II}^{\perp}$, because - as motivated in Section II - each used subgraph has to be used at least $\text{II}^{\perp}$ times in our transformation. Additionally, we pass the resource

TABLE III
BENCHMARK CHARACTERIZATION

Name	# Ops.	# lim. Ops.	$\text{II}^{\perp}_{\text{rec}}$
sam [18]	125	61	1
iir [16]	544	224	14
radix-2 [13]	736	416	1
cholesky [2]	292	240	1

allocation to the frequent subgraph mining algorithm in order to exclude patterns that would exceed the provided resource limits. Using these two problem specific methods, we reduce the search space significantly without pruning subgraphs that are interesting for our transformation.

The enumerated folding cores are combined and stored in a set of folding core combinations $\mathcal{C}$ in line 4. Our approach for combining occurrences to folding cores and folding cores to folding core combinations is shown as pseudo code in Algorithm 2. For each pattern $p_i \in \mathcal{F}$ a clique of compatible occurrences of size 0 modulo $\text{II}^{\perp}$ is generated in line 4.

Combining occurrences might introduce feedback loops. The `clique` function in line 4 secures that such occurrences are not combined (see [6] or [20]). From there on, we drop occurrences from a maximal clique randomly until this size is acquired to make the combination process as fast as possible and plan to investigate more dedicated methods in future work.

Then, this clique is stored in a new container for folding cores of compatible occurrences $\mathcal{F}_{\nabla}$ in line 5. Following Sittel *et al.*, all max cliques in $\mathcal{F}_{\nabla}$ are returned as a set folding of core combinations $\mathcal{C}$. Every combination $\phi \in \mathcal{C}$ is valid for applying our transformations as depicted in Section II.

Now, we select one combination in line 6 of Algorithm 1. As our goal is to reduce the number of resource constrained operations in the DFG to reduce solving complexity, we assign the following metric to each considered folding core combination ϕ:

$$\phi_m = \frac{N(G)}{N(G) - \sum_{p \in \phi} S^p_{min} \cdot (N(p) - 1)} \quad \forall\, \phi \in \mathcal{C}, \qquad (6)$$

where $N(G)$ is the number of operations in the original DFG, S^p_{min} is the frequency of pattern p and $N(p)$ is the number of operations in each occurrence of pattern p. The denominator of (6) calculates the number of operations that would be in the transformed DFG using ϕ. Therefore, ϕ_m is a metric to measure the possible reduction of operations and one combination with the largest value is selected randomly. We plan to research the effects of choosing different combinations on solving time and latency in future work.

Finally, using the selection $\mathcal{S}$, we generate a copy G_r of G and R_r of R and replace all occurrences described in $\mathcal{S}$ in G_r with new operations and adopt R_r in line 7. Every pattern $p_i \in \mathcal{S}$ is scheduled using a non modulo scheduler, a new hardware resource is generated, all resource limits get updated and all occurrences δ^{p_i} get replaced by replacement nodes such that the data-flow remains unchanged. Then, G_r and R_r are returned and can be used for modulo scheduling and hardware generation.

VIII. EXPERIMENTAL EVALUATION

We evaluated the proposed approach on a set of four test instances that turned out to be very hard to solve using existing modulo schedulers. The benchmarks (open-source [1]) are taken from digital signal processing and embedded computing and are listed in Table III. All models contain a large number of resource limited operations and instance iir contains a mixture of resource constrained operations and recurrences. Due to this, the Moovac formulation [14] was not able to solve these problems. We evaluated the two popular modulo schedulers from Eichenberger and Davidson (ED97) [10] and Modulo SDC (SDC) proposed by Canis *et al.* [4].

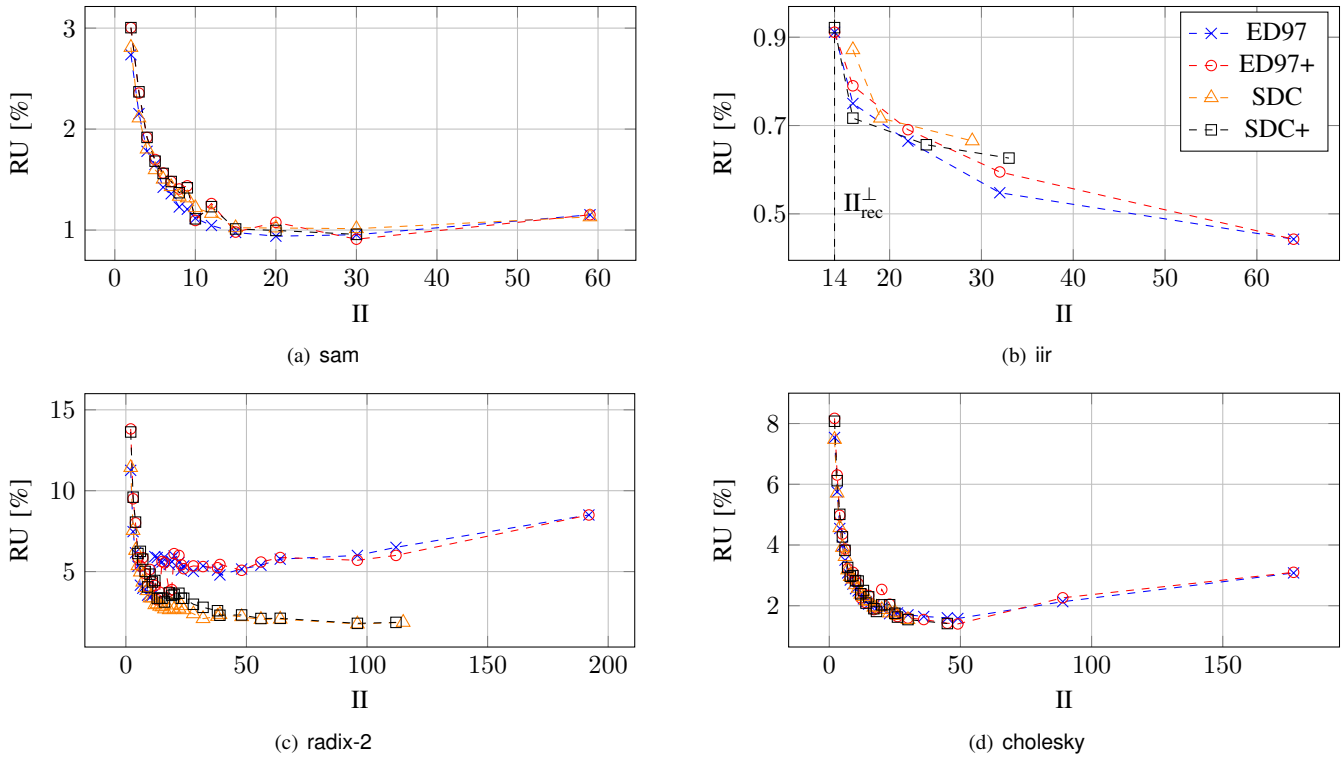
All scheduling problems were constructed such that the final hardware is able to meet a frequency of 250 MHz on the Xilinx Virtex7 xc7v2000t g1925-2G target FPGA. For determining latency information of the operators to achieve the target frequency, we used the open-source core generator *FloPoCo* [8]. The modulo scheduling problems were solved using the open source library `HatScheT` [21] and Gurobi 8.1 in single-thread mode was selected as ILP solver with a timeout of five minutes for each problem. All problems were solved using a server system with Intel Xeon CPU E5-2650 v3 processor operating at 2.3 GHz with 128 GB RAM. The hardware description after scheduling was generated using *Origami HLS* [1] and then synthesized using Vivado v2018.1.

Table IV shows a detailed overview regarding the performed experiments. For all four benchmarks, Pareto-optimal resource allocations were constructed as described in Section VI. The problems were solved using ED97 and SDC. The experiments where the proposed transformation was applied before scheduling are shown as ED97+ and SDC+, respectively.

All four approaches show the same performance for the smallest benchmark sam regarding II. But, latency was increased $1.45\times$ when the scheduler ED97+ was used. This is due to the ILP-based scheduler being able to determine latency optimal solutions for this benchmark. The iir benchmark was hard to solve using the SDC scheduler. Only four out of five problems were solved. Here, the II achieved was $0.46\times$ smaller using our approach of graph reduction, more than doubling the throughput achieved on average. Our approach was very beneficial in this case, because it prevented the SDC scheduler from running out of budget due to recursion calls. Latency was $0.8\times$ smaller on average. Comparing ED97 and ED97+ only very minor changes in latency can be observed. The radix-2 benchmark has the largest number of operations and limited operations in the benchmark set which cause the SDC scheduler to find suboptimal IIs and the ED97 scheduler to find less latency optimal solutions. One problem could not be solved within the five minutes solving time limit by SDC and SDC+. When our approach was applied to radix-2, the II was $0.92\times$ and the latency was $0.88\times$ smaller using SDC+ and ED97+, respectively. Only 23 of 27 problems of the cholesky benchmark were solved using SDC. While the II found by ED97 and ED97+ was better compared to SDC and SDC+, latency was significantly worse. Using

TABLE IV
MODULO SCHEDULING PERFORMANCE FOR TIMEOUT = 5 MINUTES

		ED97 [10]				ED97+ (prop.)				SDC [4]			SDC+ (prop.)			II w.r.t.		Lat. w.r.t.	
instance	#allocs	avg. II	avg. L.	solved	opt.	avg. II	avg. L.	solved	opt.	avg. II	avg. L.	solved	avg. II	avg. L.	solved	ED97	SDC	ED97	SDC
sam [18]	14	13.6	156.9	14	14	13.6	227.6	14	14	13.6	254	14	13.6	251	14	1.00×	1.00×	1.45×	0.99×
iir [16]	5	29.6	171.4	5	5	29.6	174.6	5	5	66.4	175	4	30.6	140	4	1.00×	0.46×	1.02×	0.80×
radix-2 [13]	31	31.2	10126	31	5	31.2	8884	31	12	33.9	91.5	30	31.2	126	30	1.00×	0.92×	0.88×	1.38×
cholesky [2]	27	27.1	5077	27	22	27.1	5161	27	22	45.8	172	23	31.9	198	24	1.00×	0.69×	1.02×	1.15×
average	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1.00×	**0.77**×	1.09×	1.08×
total	77	-	-	77	**46**	-	-	77	**53**	-	-	**71**	-	-	**72**	–	–	–	–



(a) sam (b) iir

(c) radix-2 (d) cholesky

Fig. 4. Synthesis results.

SDC+, one more solution was found and the II was 0.69× smaller on average. The reason is that only 22 of 27 problems were solved to optimality and feasible solutions showed a latency that was 100× to 1000× worse compared to SDC and SDC+ solutions. We conclude that ED97 and ED97+ are good approaches whenever resource constraints are not tight, meaning the implementation is almost parallel or the value of $II^{\perp}$ is close to $II^{\perp}_{rec}$.

Overall, the II found was 0.77× smaller on average when SDC+ was used compared to SDC. Using our approach, the latency observed was 1.09× and 1.08× higher. In total, 53 instead of 46 optimal solutions were found using ED97+ and 72 instead of 71 solutions were found using SDC+.

Figure 4 shows the post place and route synthesis results of all solved scheduling problems. On the x-axis we show the II achieved and on the y-axis we show resource usage in %:

$$\text{RU} = 100 \cdot (0.5 \cdot \frac{\#LUTs \text{ used}}{\#LUTs \text{ available}} + 0.5 \cdot \frac{\#DSPs \text{ used}}{\#DSPs \text{ available}}). \quad (7)$$

For sam, all four approaches performed roughly the same. It is noticeable that the minimal resource allocation for II=59 in our experiments does increase RU compared to the II=30 solution. This is due to larger MUX and register costs. For iir, we can see that ED97 and ED97+ outperform the SDC and SDC+ regarding II and RU. Also, one more solution was identified using the ILP-based scheduler. In addition, SDC+ provides better solutions than SDC regarding II and RU. Scheduling analysis of the radix-2 benchmark in Table IV shows that ED97 and ED97+ solved more problems, but a worse latency was achieved. This propagates to the synthesis results. For II=7 and higher, no latency optimal schedules could be found, RU increases significantly and no Pareto-optimal implementations were found. SDC and SDC+ on the other hand, are able to find modulo schedules that lead to Pareto-optimal implementations up to II=115 and II=112, respectively. Synthesis results of cholesky show that all four approaches perform roughly the same regarding II and RU. But, only ED97 and ED97+ were

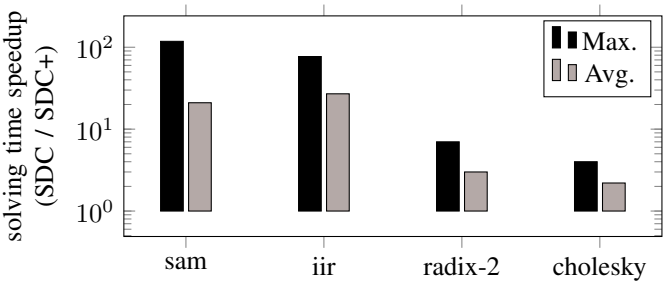


Fig. 5. Scheduler-driven DSE solving time speedup (SDC/SDC+)

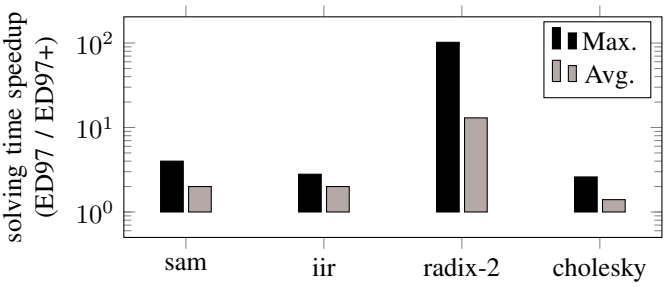


Fig. 6. Scheduler-driven DSE solving time speedup (ED97/ED97+)

able to find solutions for II>30. Note that although ED97 and ED97+ solve more modulo scheduling problems of the cholesky benchmark than SDC and SDC+, latency achieved is way worse. This, combined with the high routing effort required, results in solutions found for II=89 and II=189 being dominated regarding RU by implementations that provide higher throughput.

The maximum and average solving time speedups per benchmark comparing SDC+ with SDC and ED97+ with ED97 are shown in figures 5 and 6, respectively. We limited the maximum operation size for frequent subgraphs to four to reduce subgraph mining time. In future work, we plan to investigate this limit to subgraph size regarding run time and result quality. Due to this and our search space pruning methods described in Section VII, we were able to reduce subgraph mining and combination to < 0.1 seconds for all benchmarks and omit them as they are negligible compared to solving times. We observe solving time speedups of up to 100 for sam (using SDC+) and radix-2 (using ED97). On average, the SDC scheduler benefits more from the proposed transformation regarding solving time. The average solving time speedup observed for the large radix-2 benchmark was 10.5 when ED97+ was used. For each problem, ED97+ and SDC+ outperform ED97 and SDC, respectively.

IX. Conclusion & Outlook

We present a graph transformation for custom hardware generation using modulo scheduling. Our approach reduces graph complexity by replacing isomorphic patterns with single operations. Results show that the II achieved was improved on average by 33% for SDC-based schedulers and more latency optimal solutions were found when an ILP-based scheduler was used. But, a drawback observed was that latency achieved in clock cycles was increased by 8.5% on average over all experiments. On average, modulo scheduling problems were solved $5\times$ faster using the presented approach.

Acknowledgement

We acknowledge the financial support of grant ZI 762/5-1 from the German Research Foundation (DFG).

References

[1] Origami HLS. http://www.uni-kassel.de/go/origami.
[2] Cholesky Decomposition, 2011. http://www.alterawiki.com/wiki/Floating-point_Matrix_Inversion_Example.
[3] B. Bringmann and S. Nijssen. What is frequent in a single graph? In *Pacific-Asia Conf. on Knowledge Discovery and Data Mining*, 2008.
[4] A. Canis, S. D. Brown, and J. H. Anderson. Modulo SDC Scheduling with Recurrence Minimization in High-level Synthesis. In *24th Int. Conf. on Field Programmable Logic and Applications*. IEEE, 2014.
[5] H. Cheng, X. Yan, and J. Han. Mining Graph Patterns. Springer, 2014.
[6] J. Cong and W. Jiang. Pattern-based Behavior Synthesis for FPGA Resource Reduction. In *16th Int. Symp. on FPGAs*. ACM, 2008.
[7] S. Dai and Z. Zhang. Improving Scalability of Exact Modulo Scheduling with Specialized Conflict-Driven Learning. In *56th Annual Design Automation Conference*. ACM, 2019.
[8] F. de Dinechin and B. Pasca. Designing custom arithmetic data paths with FloPoCo. *IEEE Design & Test of Computers*, 2011.
[9] L. de Souza Rosa, C.-S. Bouganis, and V. Bonato. Scaling Up Modulo Scheduling For High-Level Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.
[10] A. E. Eichenberger and E. S. Davidson. Efficient Formulation for Optimal Modulo Schedulers. *ACM SIGPLAN*, 1997.
[11] K. Fan, M. Kudlur, H. Park, and S. A. Mahlke. Cost Sensitive Modulo Scheduling in a Loop Accelerator Synthesis System. In *38th Annual IEEE/ACM Intl. Symp. on Microarchitecture, Barcelona, Spain*, 2005.
[12] D. D. Gajski, N. D. Dutt, A. C. Wu, and S. Y. Lin. *High—Level Synthesis: Introduction to Chip and System Design*. Springer Science & Business Media, 2012.
[13] M. Garrido, J. Grajal, M. Sánchez, and O. Gustafsson. Pipelined Radix-2^k Feedforward FFT Architectures. *Trans. on Very Large Scale Integration (VLSI) Systems*, 2013.
[14] J. Oppermann, A. Koch, M. Reuter-Oppermann, and O. Sinnen. ILP-based Modulo Scheduling for High-level Synthesis. In *Int. Conf. on Compliers, Architectures, and Sythesis of Emb. Systems*. IEEE, 2016.
[15] J. Oppermann, P. Sittel, M. Kumm, M. Reuter-Opperman, A. Koch, and O. Sinnen. Design-Space Exploration with Multi-Objective Resource-Aware Modulo Scheduling. In *25th Int. Europ. Conf. on Parallel and Distibuted Comp.*, 2019.
[16] K. K. Parhi. *VLSI Digital Signal Processing Systems: Design and Implementation*. John Wiley & Sons, 2007.
[17] B. R. Rau. Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops. In *Proc. of the 27th Int.Symp. on Microarchitecture*, pages 63–74. ACM, 1994.
[18] H. Samueli. An Improved Search Algorithm for the Design of Multiplierless FIR Filters with Powers-of-two Coefficients. *Trans. on Circuits and Systems*, 1989.
[19] B. C. Schafer. Probabilistic Multiknob High-level Synthesis Design Space Exploration Acceleration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2015.
[20] P. Sittel, K. Möller, M. Kumm, P. Zipf, B. Pasca, and M. Jervis. Model-based Hardware Design based on Compatible Sets of Isomorphic Subgraphs. In *International Conference on Field Programmable Technology (ICFPT)*. IEEE, 2017.
[21] P. Sittel, J. Oppermann, M. Kumm, A. Koch, and P. Zipf. HatScheT: A Contribution to Agile HLS. In *FSP Workshop 2018; Fifth International Workshop on FPGAs for Software Programmers*. VDE, 2018.
[22] J. L. White, M.-J. Chung, A. S. Wojcik, and T. E. Doom. Efficient Algorithms for Subcircuit Enumeration and Classification for the Module Identification Problem. In *International Conference on Computer Design*. IEEE, 2001.
[23] X. Yan and J. Han. gSpan: Graph-based Substructure Pattern Mining. In *International Conference on Data Mining*. IEEE, 2002.