Improving Scalability of Exact Modulo Scheduling with Specialized Conflict-Driven Learning

Steve Dai[1,2] Zhiru Zhang[2]
[1]School of Electrical and Computer Engineering, Cornell University, Ithaca, NY [2]NVIDIA, Santa Clara, CA
{hd273,zhiruz}@cornell.edu

ABSTRACT

Loop pipelining is an important optimization in high-level synthesis to enable high-throughput pipelined execution of loop iterations. However, current pipeline scheduling approach relies on fundamentally inexact heuristics based on ad hoc priority functions and lacks guarantee on achieving the best throughput. To address this shortcoming, we propose a scheduling algorithm based on system of integer difference constraints (SDC) and Boolean satisfiability (SAT) to exactly handle various pipeline scheduling constraints. Our techniques take advantage of conflict-driven learning and problem-specific specialization to optimally yet efficiently derive pipelining solutions. Experiments demonstrate that our approach achieves notable speedup in comparison to integer linear programming based techniques.

1 INTRODUCTION

As loops abound in high-level software programs, loop pipelining is an important optimization in high-level synthesis (HLS) because it allows different iterations of a loop to be overlapped during execution in a pipelined parallel fashion. Typically enabled by modulo scheduling [5], loop pipelining creates a static schedule for a single loop iteration so that the same schedule can be repeated at a constant *initiation interval (II)*. Because II dictates the achieved throughput of the pipeline, minimizing the II is considered the foremost objective of pipeline scheduling.

While II determines the amount of parallelism, it is inherently limited by inter-iteration dependence (i.e., *recurrence*) between operations in different loop iterations. Figure 1(a) shows the data flow graph (DFG) of a loop that we will referred to throughout this paper. A static schedule for a single iteration is shown in bold in Figure 1(b). Due to the inter-iteration load-after-store dependence (indicated by the dashed arrow in Figure 1(a)) between v_5 and v_0, a subsequent iteration must start at least two cycles after the current iteration as shown in Figure 1(b). Any shorter II causes a dependence violation.

In addition to recurrence, II is also constrained by the available number of resources. Because the schedule for different loop iterations overlap in time, sufficient resources must be allocated to enable parallel execution of operations across iterations. As shown in Figure 1(b), the pipeline execution with `II=2` requires at least two memory read ports. If the same schedule targets `II=1`, at least three read ports are required due to the overlap among load operations.

Because modulo scheduling is not trivial in the presence of both recurrence and resource constraints, there exist a set of heuristics to efficiently solve the problem. For example, iterative modulo scheduling [12] applies a list scheduling like heuristic with backtracking and

DAC '19, June 2–6, 2019, Las Vegas, NV, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6725-7/19/06…$15.00
https://doi.org/10.1145/3316781.3317842

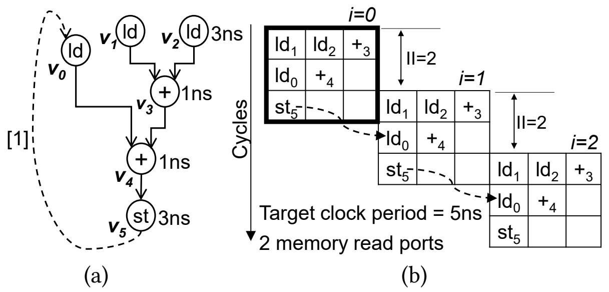

Figure 1: An example modulo scheduling problem — *(a)* DFG of the loop to be scheduled. *(b)* Static schedule for a single iteration is shown in bold. Pipeline executes by repeating the same static schedule with an II of 2 cycles.

has been adapted for loop pipelining in HLS [3]. However, state-of-the-art HLS tools typically employ the more versatile heuristic based on system of integer difference constraints (SDC) to naturally handle operation chaining and various hardware-specific constraints [4, 15]. SDC-based modulo scheduling is rooted in a linear programming formulation and can globally optimize over constraints that can be represented in the integer difference form, including both intra- and inter-iteration dependences. Notably however, resource constraints cannot be exactly modeled within an SDC formulation. As a result, SDC-based modulo scheduling resorts to incremental scheduling of resource-constrained operations on top of SDC to heuristically legalize the schedule under resource constraints. Because resource constraints are not handled exactly, SDC-based modulo scheduling lacks guarantee on achieving the optimal II.

To address this problem, we propose a modulo scheduling formulation that couples SDC with Boolean satisfiability (SAT) to exactly handle both timing and resource constraints for HLS pipelining. Similar to unpipelined scheduling with joint SDC and SAT [8], our proposed approach exploits the efficiency of SDC while leveraging the scalability of SAT to quickly prune away infeasible schedule space and derive an optimal modulo schedule. However, modulo scheduling requires a modified SDC and SAT formulation (Section 3) and calls for a different problem-specific specialization technique (Section 4). Our specific contributions are as follows:

(1) We propose a joint SDC and SAT formulation to exactly encode both resource and timing constraints for HLS pipelining.
(2) We develop an optimal algorithm based on conflict-driven learning to efficiently solve the modulo scheduling problem.
(3) We leverage problem-specific specialization to reduce the problem size and further achieve improved scalability.

2 PRELIMINARIES

A typical HLS tool employs a software compiler (e.g., LLVM, GCC) to compile the input software program into a control data flow graph (CDFG). Within this CDFG, subgraphs corresponding to loops to be pipelined are extracted for modulo scheduling, while the rest are synthesized with unpipelined scheduling techniques [6]. In this paper, we focus on the following HLS modulo scheduling problem:

Given: *(1)* A loop represented by a CDFG with intra- and inter-iteration dependences. *(2)* A set of scheduling constraints which may include resource, latency, and relative timing constraints.

Objective: Generate a modulo schedule that minimizes the II while satisfying all given constraints.

Each operation in the CDFG is associated with a value that indicates the combinational delay of the operation. These delays are used to chain operations into the same cycle based on the target clock period of the problem. In Figure 1(a), we label the combinational delays for the four distinct types of operations in the graph. These delays are used during scheduling to satisfy the clock period constraint denoted in Figure 1(b). Unlike intra-iteration dependence edge, each inter-iteration dependence edge is associated with a distance indicating the number of loop iterations between the occurrences of the dependent operations. In Figure 1(a), the inter-iteration dependence edge in dash has a distance of 1 indicating that v_0 of the next iteration depends on v_5 of the current iteration.

In addition, the problem consists of resource constraints in the form of a resource model containing a set of different resource types (e.g., memory port, floating point multiplier). There exist a finite number of resources of each type in the resource model. If an operation requires any resource from the resource model to execute, we call this operation a *resource-constrained* operation. For example, the schedule in Figure 1(b) is derived based on a resource constraint of two memory read ports (as indicated), which allows at most two load operations in each cycle.

A modulo scheduling solution assigns each operation i to a *time step* t_i which indicates the cycle at which the operation executes. Due to the modulo nature of the scheduling from overlapping different iterations, each time step t_i corresponds to a (modulo) *time slot* s_i, where $s_i = t_i \% II$. For the bolded schedule in Figure 1(b), store operation v_5 is scheduled in the third time step ($t_5 = 2$) with no other operations. However, it is assigned to the first time slot ($s_5 = 0$) along with loads from v_1 and v_2 and addition from v_3. While there can be as many time steps as needed, there can be at most `II` time slots. A modulo reservation table (MRT) indicates the number of each type of resource used by all operations scheduled in each time slot. A feasible modulo scheduling solution requires an MRT in which no resource is oversubscribed in any time slot.

Because modulo scheduling is generally NP-hard under both resource and recurrence constraints, many heuristics such as iterative modulo scheduling [12] have been proposed to quickly derive a solution with a small II. There also exist enumeration-based approaches [1] to exactly solve the problem. Given the state of the field, we focus on describing the best known heuristic and exact modulo scheduling techniques in Sections 2.1 and 2.2.

2.1 SDC-based Formulation

In general, SDC-based scheduling [7] declares a variable t_i to denote the clock cycle (time step) at which operation i in the CDFG is scheduled. Timing constraints, such as dependence and cycle time constraints, can then be represented exactly as the differences of these variables. To handle data dependence for modulo scheduling in particular, SDC creates the following difference constraint

$$t_i - t_j \leq II \cdot Dist_{ij} - L_{ij} \quad (1)$$

where L_{ij} is the minimum latency between operation i and j, and $Dist_{ij}$ is the distance of the dependence. To schedule the DFG in Figure 1(a), for example, we impose the constraint $t_0 - t_4 \leq 0$ to honor the intra-iteration dependence between v_0 and v_4. This ensures that v_4 is scheduled no earlier than v_0. Note that an intra-iteration dependence corresponds to a dependence distance $Dist_{ij} = 0$ in the constraint in Eq. (1). Similar constraints are constructed for other intra-iteration data dependence edges. For the inter-iteration dependence in Figure 1(a), we impose the constraint $t_5 - t_0 \leq II - 1$, where II is the II currently being targeted. Intuitively, this constraint imposes a deadline for the schedule time of v_5 relative to the schedule time of v_0 beyond which v_5 from the current iteration will not execute in-time to produce results needed by v_0 of the following iteration.

To honor the target clock period T_{clk}, SDC identifies the maximum critical combinational delay $D(ccp(v_i, v_j))$ between pairs of operations v_i and v_j and impose the difference constraint in Eq. (2) to ensure pairs of operations whose critical delay exceeds the target clock period are scheduled in different cycles.

$$t_i - t_j \leq -1 \;\; \forall\, (v_i, v_j) \ni D(ccp(v_i, v_j)) > T_{clk} \quad (2)$$

For our example in Figure 1(a), we impose $t_2 - t_5 \leq -1$ to separate $v2$ and $v5$ into different cycles because the critical combinational path from $v2$ to $v5$ exceeds the target clock period of `5ns`. Similar constraints are imposed between v_0 and v_5 as well as v_1 and v_5.

While timing constraints can be handled naturally in SDC, resource constraints are difficult to represent even heuristically because the non-linearity of the MRT requires that operations using the same resource must not only be scheduled in different time steps but also in different time slots. As a result, simple partial ordering constraints in the form of $t_i - t_j \leq -1$ used to produce resource-abiding schedules in unpipelined SDC scheduling [7] fail to honor the complete set of resource constraints in the case of modulo scheduling. To handle resource constraints, SDC-based modulo scheduling [4, 15] rely on stepwise legalization of the non-resource-constrained SDC schedule against the MRT to heuristically derive a solution, resulting in no guarantee on optimality.

2.2 ILP-based Formulation

In place of heuristics, integer linear programming (ILP) can be applied to exactly model the modulo scheduling problem. Eichenberger and Davidson [9] leverages binary variable $a_{i,r}$ to indicate whether operation i is scheduled in time slot r in order to encode the modulo schedule. Timing and resource can then be constrained with this variable. Oppermann et al. [11] improves upon the resource handling capability of Eichenberger and Davidson by using binary variables to represent resource and time slot overlap instead of the actual modulo schedule. In particular, they propose to use binary overlap variable ϵ_{ij} to denote whether operation i's resource instance index is strictly less than j's index, and μ_{ij} to denote whether operation i is executed in a time slot strictly earlier than the time slot of j. These binary variables are in turn constrained with resource index variable r_i, which denotes the index of resource instance used by operation i, and time slot variable s_i, which denotes the modulo time slot of operation i, as in Eichenberger and Davidson. Given these constraints on the consistency of variables ϵ_{ij}, μ_{ij}, r_i, and s_i, resource constraints are satisfied by ensuring that every pair of operations (i, j) use difference resource instances, or are scheduled at different time slots as followed:

$$\epsilon_{ij} + \epsilon_{ji} + \mu_{ij} + \mu_{ji} \geq 1 \quad (3)$$

Even though the ILP formulations handle all constraints exactly and can return a schedule that satisfies a specific II given enough time, ILP is in general NP-hard and difficult to scale. ILP also requires significantly more variables than SDC for encoding the same problem and is too general to exploit problem-specific properties.

3 JOINT SDC AND SAT FORMULATION

Given the tradeoff between scalability and quality in the comparison between SDC and ILP in Section 2, we propose a modulo scheduling algorithm that integrates SDC and SAT to exactly handle various types of constraints and optimally solve the modulo scheduling problem. Borrowing the idea of unpipelined scheduling with joint SDC and

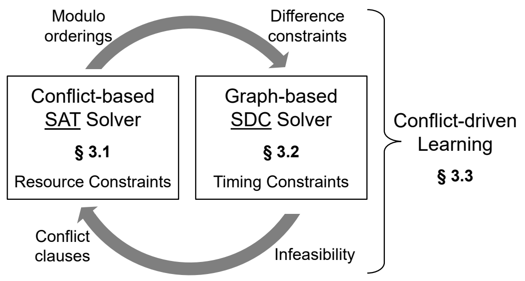

Figure 2: Overall structure of our modulo scheduler.

SAT [8], our proposed formulation leverages SDC to naturally handle timing constraints and SAT to exactly encode resource constraints.

As shown in the high-level diagram in Figure 2, our modulo scheduler is composed of a conflict-based SAT solver coupled with a graph-based SDC solver in a conflict-driven learning loop. On the left, we have a SAT solver that takes advantage of conflict-based search (detailed in Section 3.1) to propose, what we referred to as, modulo orderings that satisfy resource constraints imposed by the MRT. These modulo orderings are converted into difference constraints in SDC and inserted into the SDC problem. On the right, we have an SDC solver that takes advantage of graph-based traversal (detailed in Section 3.2) to check the feasibility of the modulo orderings. Any infeasibility is encoded as a conflict clause in SAT and added to the SAT problem. Given the practical scalability of SAT and the efficiency of SDC, our solver iterates between SAT and SDC (described in Section 3.3) until a feasible solution is found or proven to be non-existent.

3.1 Resource Constraints in SAT

To handle resource constraints, we declare binding variable B_{ik} to denote whether operation i is bound to resource instance k. B_{ik} is true if operation i is bound to resource instance k. By constraining the binding variables with the SAT clause $\sum_k B_{ik} = 1 \ \forall i$, we can ensure that each resource-constrained operation is assigned to exactly one resource instance and that no resource is oversubscribed. With the binding variables, it follows that sharing variable R_{ij} can be derived to denote whether operation i is sharing the same resource instance with operation j as followed:

$$R_{ij} = \bigvee_{k \in T_p} (B_{ik} \wedge B_{jk}) \tag{4}$$

where T_p denotes resource instances of type p. R_{ij} is true if both operations i and j are bound to the same resource instance.

Pipeline scheduling prohibits two operations i and j that share the same instance of resource from being scheduled in the same modulo time slot. In other words, $t_i - t_j \neq kII \quad \forall\ k \in \mathbb{Z}$, which translate to a disjunctive set of constraints $kII < t_i - t_j < (k+1)II$ and $kII < t_j - t_i < (k+1)II$. Therefore, we introduce modulo ordering variables $O_{i\to j,k}$ to represent these constraints as follows:

$$O_{i\to j,k} = True \ \mapsto\ (k-1)II < s_i - s_j < kII \ \ \forall\ k \in \mathbb{Z} \tag{5}$$

$$O_{i\to j,k} = False \ \mapsto\ \emptyset \tag{6}$$

As shown in Eq. (5), assigning $O_{i\to j,k}$ to true maps to the difference constraint where operation i must be scheduled in an earlier cycle than j. Furthermore, their distance must be greater than kII and less than $(k+1)II$ cycles, which means that they are separated apart by k II-intervals. As shown in Eq. (6), assigning $O_{i\to j,k}$ to false maps to an empty set of constraints, indicating that it is not necessary to impose any partial ordering between operations i and j because the particular resource binding does not require any partial ordering. Note that k can be bounded by the length of any non-modulo schedule or the

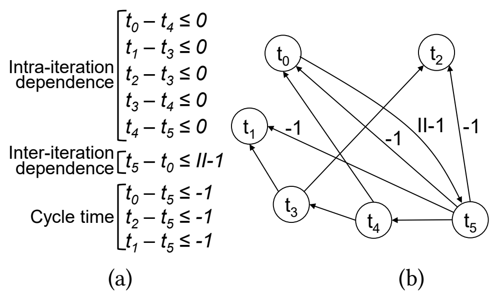

Figure 3: SDC constraints and corresponding SDC constraint graph — *(a)* Timing constraints in SDC. *(b)* SDC constraint graph.

lengths of recurrence cycles. Here we use T to denote the bounded space of k. Ultimately, $O_{i\to j}$ is derived from modulo orderings as:

$$O_{i\to j} = \sum_{k\in T} O_{i\to j,k} \le 1 \tag{7}$$

As shown in Eq. (7), $O_{i\to j}$ is true if operation i is scheduled before j, and they are not in the same time slot.

Given the mapping between SAT and SDC, the following clauses are included to connect the the sharing variables with the modulo ordering variables:

$$R_{ij} \to (O_{i\to j} \vee O_{j\to i}) \tag{8}$$

$$\neg(O_{i\to j} \wedge O_{j\to i}) \tag{9}$$

Eq. (8) indicates that if operations i and j share the same resource instance, operation i must be scheduled either in an earlier cycle or in a later cycle than operation j, but certainly not in the same time slot. Eq. (9) ensures that operation i cannot be simultaneously scheduled both in an earlier cycle and later cycle than j.

3.2 Timing Constraints in SDC

Timing constraints in SDC can be conveniently represented using a constraint graph in which each variable maps to a node and each constraint maps to an edge in the graph. Figure 3(a) shows the set of intra-iteration dependence, inter-iteration dependence, and cycle time constraints in SDC form for our example. These constraints map to the nodes and edges in the constraint graph in Figure 3(b). For each constraint in the integer difference form $t_i - t_j \le C_{ij}$, the constraint graph includes an edge of weight C_{ij} from node j to i. For clarity, note that we have omitted the weights for zero-weight edges.

With this graph-based representation, we can easily derive a feasible schedule, either as soon as possible (ASAP) or as late as possible (ALAP) schedule, by solving a single source shortest path problem. In addition, we can conveniently detect infeasibility of the difference constraints by the presence of negative cycle in the graph. For example, adding the SDC constraint $t_0 - t_2 \le -1$ to the system in Figure 3(a) induces the dashed edge from t_2 to t_0 in Figure 4, creating a negative cycle (shown in bold) that indicates the system is infeasible.

3.3 Conflict-Driven Learning

As shown in Figure 2, SAT and SDC work closely in a loop to handle both resource and timing constraints exactly and efficiently. For each iteration, SAT makes a proposal of modulo orderings that satisfy resource constraints in Eq. (4), (7), (8), and (9) by determining a satisfiable assignment for the modulo ordering variables $O_{i\to j,k}$. SDC constraints used to enforce resource constraints are created based on the mapping in Eq. (5) and appended to the SDC graph. SDC checks the feasibility of the updated graph and returns any feasible solution as the schedule if the graph is feasible. If the graph is infeasible, SDC instead returns the modulo ordering edges involved

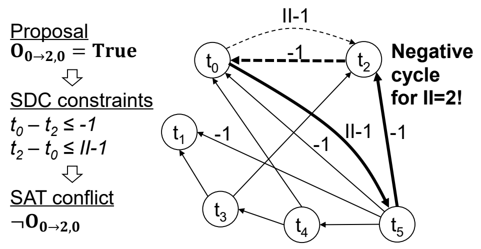

Figure 4: One iteration of conflict-driven learning.

in the negative cycle that causes the infeasibility. The infeasibility is encoded as conflict clause and appended to the SAT problem. In future iterations, SAT will no longer propose any modulo ordering that violates any previously added conflicts. The solver iterates until a feasible solution is found or the SAT search space is exhausted. While there may be multiple feasible solutions, our solver returns the earliest encountered feasible schedule.

Figure 4 illustrates a single iteration of the conflict-driven learning process with our target II=2. In this iteration, assume that SAT proposes a modulo ordering that assigns variable $O_{0\to 2,0}$ to true, which enforces that operation 0 is scheduled before 2, and that the two operations are less than one II apart. The two corresponding SDC constraints are shown on the left. SDC then adds the edges (in dash) corresponding to these constraints to the graph and detects a negative cycle (in bold) that involves one of the two newly added edges. The involvement of this edge in the negative cycle results in the conflict clause on the left, which prevents any modulo ordering with operation 0 scheduled before 2 and within one II apart. SAT will then continue onto the next iteration with a different proposal that satisfies resource constraints but does not involve this previously infeasible edge. As you can see, SAT learns from previously infeasible edges in each iteration to prune the search space. We leverage negative cycle to keep the generated SAT conflict clauses as short as possible because shorter conflicts are able to prune out more of the search space in SAT and result in faster convergence of the solver. If the search space is completely pruned out before finding a feasible solution, the problem is infeasible for the particular II. Otherwise, the problem is feasible where the shortest path solution serves as the feasible schedule.

3.4 Optimization

While we take advantage of conflict-driven learning to derive a modulo schedule that satisfies a particular II, we optimize for the best II by starting with a lower bound value for the II and incrementing it one at a time until the II is feasible or an upper bound value has been reached. Because this is a conventional optimization technique typically employed by HLS tools [3, 6], our conflict-driven learning algorithm is generally applicable regardless of the approach used to establish such upper and lower bounds on II.

4 GRAPH-BASED PROBLEM REDUCTION

In general, the graph to be modulo scheduled can be partitioned into acyclic and cyclic subgraphs. The acyclic subgraphs contain only forward edges, while the cyclic subgraphs contain backward edges in addition forward edges. Because resource constraints essentially delay the execution of resource-constrained operations, resource constraints may cause violation with timing constraints imposed by backward edges. Due to the interaction between backward edges and resource constraints, exactly scheduling the cyclic subgraphs in the presence of resource constraints constitutes the "hard" aspect of the modulo scheduling problem. Subgraphs that are acyclic or not constrained by resource can be scheduled efficiently and exactly with heuristics. Based on this observation, we propose to further

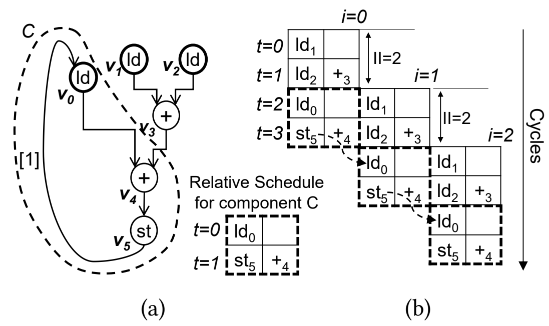

Figure 5: Illustration of graph-based problem reduction — *(a)* C is a complex component that is scheduled first. *(b)* Final schedule.

accelerate the performance of exact modulo scheduler by reducing the complexity of the exact modulo scheduling problem to that of exactly scheduling the cyclic resource-constrained subgraphs.

To enable graph reduction, we rely on the strongly connected component (SCC) algorithm [14] to partition the graph into cyclic and acyclic components. This results in a directed acyclic graph (DAG) of the SCCs of the input graph. Some SCCs form a *trivial subgraph* consisting of only a single node, while others form a *non-trivial subgraph* consisting of multiple nodes. Those with multiple nodes must be cyclic. We refer to cyclic SCCs with resource-constrained nodes as *complex subgraphs* and cyclic SCCs with no resource-constrained nodes as *basic subgraphs*. In Figure 5(a), component C is a complex subgraph because it contains resource-constrained memory operation. Otherwise, it would have been categorized as a basic subgraph. Because they are single nodes, v_1, v_2, and v_3 each constitutes a trivial subgraph regardless of whether they are resource-constrained.

We combine all basic subgraphs without connections between them into a *basic supergraph*. This basic supergraph can be solved exactly with SDC-based modulo scheduling because there is no resource constraint. The scheduling solution of this basic supergraph will satisfy all timing constraints imposed by the edges within the supergraph. Similarly, we combine all complex subgraphs without connections in between into a *complex supergraph*. However, due to the interaction between timing constraints (from backward edges) and resource constraints, this complex supergraph must be solved with an exact technique such as our proposed joint SDC and SAT modulo scheduling algorithm detailed in Section 3. The solution of the complex supergraph will satisfy all timing constraints imposed by edges in the supergraph as well as the resource constraints of the modulo scheduling problem. Regardless of basic or complex supergraph, the schedule generated serves as a *relative schedule* that we can use later to commit the final schedule. Operations satisfy the relative schedule as long as the relative time positions at which operations execute remain unchanged. For example, the relative schedule in Figure 5(a) is satisfied as long as the store and addition operations are executed one cycle after the load operation.

To motivate our subsequent procedure in Algorithm 1 for committing the final schedule based on the relative schedules of basic and complex supergraphs, we identify several interesting properties of the different types of subgraphs. Due to space limitation, we provide an informal proof of each property to illustrate the main ideas. Here $t_G(i)$ denotes the schedule time of operation i in the relative schedule, and $t(i)$ denotes the schedule time in the committed schedule.

Property 1. *Given any time step T and a relative schedule $\{t_G(i), \forall i\}$ of a basic subgraph G, committing every operation i to time step $t(i) = T + t_G(i)$ satisfies all timing constraints imposed by G.*

The relative time positions between operations remain the same in the committed schedule $\{t(i), \forall i\}$ as in the original relative schedule $\{t_G(i), \forall i\}$ for G. Because the original relative schedule satisfies all timing constraints imposed by G, the committed schedule must also satisfy all timing constraints imposed by G.

Property 2. *Given any time step T and a relative schedule $\{t_G(i), \forall i\}$ of a complex subgraph G, there exists an integer constant $\delta : 0 \leq \delta < II$ such that committing every operation i to time step $t(i) = T + t_G(i) + \delta$ satisfies all timing constraints of the subgraph as well as the time slot assignment imposed by the relative schedule.*

Assume that the operations are first committed as in Property 1. If the time slot assignments are satisfied, we have obtained a committed schedule that satisfies both timing constraints and time slot assignments. If the time slot assignments are not satisfied, we can repeatedly increase the time steps of all operations simultaneously by one cycle until the time slot assignments are satisfied. Due to the modulo nature of the schedule, we must be able to find a schedule that satisfies the time slot assignments within II cycles. Any schedule we commit also satisfies the timing constraints of the subgraph because we increase the time steps of all operations by the same constant number of cycles δ and maintain their relative positions in time.

For example, if we would like to commit the relative schedule in Figure 5(a) for $T = 1$, the load for v_0 will be committed to $t(0) = 1$ if $\delta = 0$. This commits v_0 to time slot 1, violating v_0's being scheduled in time slot 0 in the relative schedule. Therefore, the entire relative schedule must be delayed by one cycle to accommodate the time slot assignments. With $\delta = 1$, v_0, v_5, and v_4 will be committed to $t(0) = 2$, $t(5) = 3$, and $t(4) = 3$, where both timing constraints between these operations and their time slot assignments are satisfied.

Property 3. *Given any time step T and that the complex supergraph of the problem has been scheduled and committed to the MRT, the single operation i in each trivial subgraph G can always be committed at some time step $t(i) : T \leq t(i) < T + II$ without violating resource constraints.*

A trivial subgraph contains a single operation. If the operation is not constrained by resource, committing it at any time step will not violate resource constraints. If the operation is constrained by resource, there must be some slot with available resource in the MRT for the operation to be scheduled because enough II slots should have been pre-allocated to satisfy resource constraints. Because a single resource-constrained operation can be scheduled in any time slot in the MRT, committing it will not violate any resource constraints.

In Figure 5, assume v_0, v_5, and v_4 have been committed to slots 0, 1, and 1, respectively, after relatively scheduling the complex subgraph C. Further assume that v_1 is then schedule to $t(1) = 0$, which corresponds to slot 0. With these operations committed, all read port resources in slot 0 of the MRT are subscribed. Because of this, v_2 must be committed to $t(2) = 1$ and slot 1 because slot 0's read ports have been fully subscribed. However, there must be available resource in slot 1 for v_2 because the minimum resource-constrained II of 2 requires at least two modulo time slots in the MRT, each with two read ports available.

Based on the above properties, any operation i in subgraphs (regardless of type) can always be scheduled at some time step $t(i) \geq T$, given a reference time T, without violating timing constraints within the subgraphs or resource constraints of the modulo scheduling problem. Therefore, we can traverse the subgraphs (SCCs) in a topological order (because the graph of SCCs form a DAG) and commit the operations in each SCC to the earliest possible time step (i.e., ASAP). Dependence between subgraphs (manifested by forward edges) determines the earliest time step for which operations in each subgraph can be scheduled. If we consider this earliest time step as T in the previous properties, operations in the subgraph can be committed based on those properties from this reference time step. Committing subgraphs in topological order ensures that timing constraints between subgraphs, all of which must be forward edges, are also fully satisfied. Our exact modulo scheduling algorithm with graph-based problem reduction is listed in Algorithm 1.

Algorithm 1 `ExactModuloSchedulingWithGraphReduction(II)`

```
1: Partition the graph into its SCCs
2: Compute relative schedule for basic supergraphs with SDC
3: Compute relative schedule for complex supergraphs with exact method
4: Update MRT for scheduled resource-constrained nodes
5: for each component in topologically sorted order of SCCs do
6:     if component is a basic subgraph then
7:         Schedule ASAP based on relative schedule
8:     else if component is complex subgraph then
9:         Schedule ASAP based on relative schedule while satisfying time slot assignment
10:    else if component is single resource-constrained node then
11:        Schedule ASAP at time slot with available resource
12:        Update MRT for newly scheduled resource-constrained node
13:    else
14:        Schedule ASAP
15:    end if
16: end for
17: if Any above step is infeasible then II is infeasible end if
```

In our algorithm, Line 7 commits operations in a basic subgraph to the final schedule based on Property 1 to satisfy timing constraints, while Line 9 commits operations in a complex subgraph to the final schedule based on Property 2 to ensure that both timing and resource constraints are honored. If the subgraph turns out to be a single resource-constrained node, Line 11 commits the operation, based on Property 3, to the earliest possible time step whose corresponding time slot has available resource remaining in the MRT. If the subgraph is a single node that is not constrained by resource, it can be scheduled ASAP as shown in Line 14. Our algorithm is essentially an ASAP scheduling scheme subject to resource availability and any prior relative assignment of time steps (of basic and complex subgraphs) and exact assignment of modulo time slots (of complex subgraphs). Figure 5(b) shows the committed schedule for scheduling the graph in Figure 5(a) with Algorithm 1. Note that our graph reduction technique is generally applicable to any exact modulo scheduling techniques, including ILP.

5 EXPERIMENTS

We implement our modulo scheduler in C++, interfaced with the Lingeling SAT solver [2]. We execute our scheduler on an Intel Xeon CPU running at 2.5GHz and evaluate it on a set of 350 benchmark loops from popular HLS benchmark suites CHStone [10] and MachSuite [13]. For experiment purpose, we further classify the benchmarks into *trivial*, *easy*, and *challenging* benchmarks to better evaluate the benefit of our proposed approach. In particular, trivial benchmarks contain no complex component in the graph. Given our graph-based problem reduction technique, these benchmarks do not require exact modulo scheduling to be solved optimally. Therefore, they are not included in our runtime evaluations to avoid skewing the results. For the other benchmarks, we compare the runtimes of our joint SDC and SAT scheduler, with and without graph reduction, against those of state-of-the-art commercial ILP solver CPLEX running the best known ILP-based modulo scheduling formulation [11] (described in Section 2.2). For convenience, we use `ILP` and `SDS+` to refer to the two scheduling techniques, but qualify each technique with `Default` or `Reduced` to indicate whether graph reduction has been applied.

Figure 6 summarizes the runtime speedup of SDS+ on easy benchmarks, which contain complex subgraphs but can solved by Default ILP in less than one second. Each bar represents the runtime speedup against Default ILP for one benchmark. Benchmarks are ordered

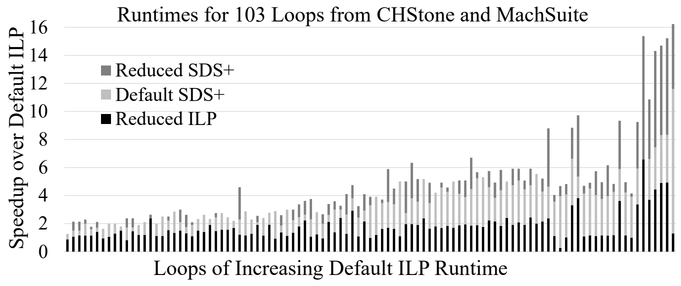

Figure 6: Runtime evaluation on easy loops — Default ILP's runtimes are less than one second. Each color represents the additional speedup achieved over the previous solver against Default ILP.

by increasing Default ILP runtime. Each color represents the additional speedup achieved over the previous solver configuration. For example, black bar shows the runtime speedup of Reduced ILP over Default ILP, while black and light gray together show the speedup of Default SDS+ over Default ILP. Black, light gray, and gray together indicate the speedup of Reduced SDS+ over Default ILP. Figure 6 shows that SDS+, with or without reduction, is consistently more competitive than ILP, with or without reduction. While simply applying the graph-based reduction technique on ILP leads to some degree of speedup, applying SDS+ achieves speedup that grows faster as the difficulty of the problem increases. In addition, Reduced SDS+ can achieve more than one order of magnitude speedup from Default ILP for the most difficult cases in this plot.

In Table 1, we evaluate the runtimes of more challenging loops, which contain complex subgraphs and require more than one second of ILP time. The loops are sorted by the total number of operations before reduction. However, the exact relationship between various metrics and speedup depends on the graph topology and the interaction between resource-constrained nodes and cyclic subgraphs. SDC and SAT complexity is determined by the number of operations in accordance to Section 3. Overall, while Reduced ILP provides marginal speedup from Default ILP, SDS+ (with and without reduction) is able to significantly widen the speedup gap for almost all the loops. With graph reduction, Reduced SDS+ is especially competitive on the more difficult loops like `JPEG19` and `ADPCM2`, achieving over two orders of magnitude of speedup. For `JPEG23`, Reduced ILP's speedup is noticeable because graph reduction decreases the total number of operations by around 86% and the number of resource-constrained operations by 50%. As a result, SDS+ gives no further benefit on top of Reduced ILP in this case. On the other hand, Reduced ILP reaps negligible benefit for `JPEG87` because the graph contains a long recurrence cycle that encapsulates all resource-constrained operations. SDS+'s performance in this case shows that SDS+ is able to handle constraints more efficiently. Noticeably, Reduced SDS+ is able to solve `ADPCM1` and `DFSIN1` for which both ILP and Reduced ILP time out.

While runtime is not strictly proportional to node count, the ILP-based formulations become difficult to solve as the total number of operations increases beyond 400 or the number of resource-constrained nodes goes beyond 50. In these cases, SDS+ demonstrates improved scalability by combining the efficiency of SDC and SAT. Although the ILP formulation by Oppermann et al. [11] used in our experiments has demonstrated improved resource handling capability than previous formulations, it is still more susceptible to scalability issues because both timing and resource constraints are encoded as ILP constraints, which are NP-hard to solve. Instead, SDS+ handles the timing aspect of the problem using polynomial-time SDC and leaves the resource aspect to SAT. In addition, our reduction technique can help prune out nodes to further alleviate scalability issues. For `JPEG19` and `ADPCM2`, graph reduction helps ILP become manageable.

Benchmark	#Ops	Reduced #Ops	Default ILP	Reduced ILP	Default SDS+	Reduced SDS+
FFT_STRIDED2	84 / 18	24 / 18	4.29	1.11 (3.86x)	0.804 (5.33x)	0.091 (47.1x)
JPEG23	135 / 26	18 / 13	2.30	0.190 (12.1x)	0.294 (7.82x)	0.029 (79.3x)
JPEG18	175 / 23	114 / 23	8.60	3.03 (2.84x)	0.512 (16.7x)	0.529 (16.2x)
MD_GRID7	300 / 50	298 / 50	7.14	3.06 (2.33x)	0.285 (25.1x)	0.257 (27.8x)
JPEG87	380 / 37	370 / 37	7.13	6.71 (1.06x)	0.642 (11.1x)	0.361 (19.7x)
JPEG19	476 / 65	465 / 65	TO	397 (>2.27x)	3.41 (>264x)	2.35 (>383x)
JPEG17	942 / 93	615 / 68	156	96.1 (1.62x)	7.29 (21.4x)	6.93 (22.5x)
ADPCM2	710 / 114	295 / 80	TO	562 (>1.60x)	TO	2.01 (>448x)
ADPCM1	777 / 108	466 / 102	TO	TO	31.1 (>28.9x)	11.9 (>75.6x)
DFSIN1	2651 / 74	115 / 74	TO	TO	TO	606 (>10.0x)

Table 1: Runtime evaluation for more challenging loops — `#Ops` shows the total number of operations and number of resource-constrained operations before graph reduction. `Reduced #Ops` shows the same numbers after graph reduction. `TO` indicates timeout after 15 minutes or 10x SDS+ runtime, whichever is greater.

6 CONCLUSIONS

Current pipelining approach relies on fundamentally inexact heuristics with ad hoc priority functions that provide no guarantee on achieving the best throughput. To address this problem, we propose a new modulo scheduling algorithm that combines the efficiency of SDC and scalability to SAT to exactly handle various pipelining constraints. In addition, we identify problem-specific opportunity to further accelerate the performance of our modulo scheduler. Our work aims to improve the scalability of exact modulo scheduling and redefine the tradeoff between scalability and quality.

ACKNOWLEDGEMENTS

We would like to thank researchers at TU Darmstadt and University of Kassel for providing many useful benchmark graphs for this project. This research was supported in part by NSF Awards #1453378, #1618275, Semiconductor Research Corp., and a research gift from Xilinx, Inc.

REFERENCES

[1] Erik R. Altman and Guang R. Gao. Optimal Modulo Scheduling through Enumeration. *Int'l Journal of Parallel Programming*, 1998.

[2] Armin Biere. Lingeling, Plingeling and Treengeling Entering the SAT Competition 2013. *SAT Competition*, 2013.

[3] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. Anderson, S. Brown, and T. Czajkowski. LegUp: High-Level Synthesis for FPGA-Based Processor/Accelerator Systems. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2011.

[4] Andrew Canis, Stephen D. Brown, and Jason H. Anderson. Modulo SDC Scheduling with Recurrence Minimization in High-Level Synthesis. *Int'l Conf. on Field Programmable Logic and Applications (FPL)*, 2014.

[5] Josep M. Codina, Josep Llosa, and Antonio González. A Comparative Study of Modulo Scheduling Techniques. *Int'l Conf. on Supercomputing*, pages 97–106, Jun 2002.

[6] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2011.

[7] Jason Cong and Zhiru Zhang. An Efficient and Versatile Scheduling Algorithm Based on SDC Formulation. *Design Automation Conf. (DAC)*, 2006.

[8] Steve Dai, Gai Liu, and Zhiru Zhang. A Scalable Approach to Exact Resource-Constrained Scheduling Based on a Joint SDC and SAT Formulation. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2018.

[9] Alexandre E. Eichenberger and Edward S. Davidson. Efficient Formulation for Optimal Modulo Schedulers. *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 1997.

[10] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, Hiroaki Takada, and Katsuya Ishii. CHStone: A Benchmark Program Suite for Practical C-Based High-Level Synthesis. *Int'l Symp. on Circuits and Systems (ISCAS)*, 2008.

[11] Julian Oppermann, Andreas Koch, Melanie Reuter-Oppermann, and Oliver Sinnen. ILP-based Modulo Scheduling for High-Level Synthesis. *Intl'l Conf. on Compilers, Architectures and Synthesis of Embedded Systems (CASES)*, 2016.

[12] B. Ramakrishna Rau. Iterative Modulo Scheduling. *Int'l Journal of Parallel Programming*, 1996.

[13] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. MachSuite: Benchmarks for Accelerator Design and Customized Architectures. *Int'l Symp. on Workload Characterization (IISWC)*, 2014.

[14] Robert Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1972.

[15] Zhiru Zhang and Bin Liu. SDC-Based Modulo Scheduling for Pipeline Synthesis. *Int'l Conf. on Computer-Aided Design (ICCAD)*, 2013.