A Scalable Approach to Exact Resource-Constrained Scheduling Based on a Joint SDC and SAT Formulation

Steve Dai, Gai Liu, Zhiru Zhang
School of Electrical and Computer Engineering, Cornell University, Ithaca, NY
{hd273,gl387,zhiruz}@cornell.edu

ABSTRACT

Despite increasing adoption of high-level synthesis (HLS) for its design productivity advantage, success in achieving high quality-of-results out-of-the-box is often hindered by the inexactness of the common HLS optimizations. In particular, while scheduling forms the algorithmic core to HLS technology, current scheduling algorithms rely heavily on fundamentally inexact heuristics that make ad hoc local decisions and cannot accurately and globally optimize over a rich set of constraints. To tackle this challenge, we propose a scheduling formulation based on system of integer difference constraints (SDC) and Boolean satisfiability (SAT) to exactly handle a variety of scheduling constraints. We develop a specialized scheduler based on conflict-driven learning and problem-specific knowledge to optimally and efficiently solve the resource-constrained scheduling problem. By leveraging the efficiency of SDC algorithms and scalability of modern SAT solvers, our scheduling technique is able to achieve on average over 100x improvement in runtime over the integer linear programming (ILP) approach while attaining optimal latency. By integrating our scheduling formulation into a state-of-the-art open-source HLS tool, we further demonstrate the applicability of our scheduling technique with a suite of representative benchmarks targeting FPGAs.

ACM Reference format:
Steve Dai, Gai Liu, Zhiru Zhang. 2018. A Scalable Approach to Exact Resource-Constrained Scheduling Based on a Joint SDC and SAT Formulation. In *Proceedings of 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, February 25–27, 2018 (FPGA '18),* 10 pages.
https://doi.org/10.1145/3174243.3174268

1 INTRODUCTION

The breakdown of Dennard scaling has led to the rapid growth of specialized hardware accelerators to meet the ever more stringent performance and energy requirements. However, great performance-per-watt comes at the cost of enormous development effort. With the traditional register-transfer-level (RTL) design flow, designers must constantly wrestle with low-level hardware description languages (HDLs) and manually explore a large multidimensional solution space. With the RTL design methodology, it is difficult to re-target multiple design points because the timing and micro-architecture are essentially fixed by design.

As the process of RTL optimization becomes unequivocally difficult, if not already unsustainable, high-level synthesis (HLS) has emerged as a promising alternative to the RTL design methodology for tackling the design productivity gap [19]. HLS raises the abstraction of input from HDL to software programming language by providing the capability to automatically synthesize untimed high-level software programs into cycle-accurate RTL implementations. Lower design complexity and faster simulation speed enable shorter time-to-market, which is especially relevant in today's rapidly-evolving technology landscape. Most recently, HLS has successfully accelerated the design of complex and realistic applications [22, 30, 37] as well as system-on-chip [1]. The productivity advantage has led to growing adoption of commercial and open-source HLS tools, including Vivado HLS [7]and LegUp [4].

Because HLS transforms an untimed, possibly sequential, description with no concept of clock into a timed parallel implementation with registers, scheduling has been recognized as one of the most important problems in HLS. Scheduling extracts parallelism from the input high-level program and determines the clock cycle at which different computation and communication operations should be executed. With exclusive control on timing at the front-end of the hardware flow, scheduling is in a unique position to influence the micro-architecture and quality of the generated hardware. Nevertheless, finding an optimal schedule is intractable in general, and thus necessitates a tradeoff between optimality and efficiency.

For example, HLS traditionally solves the classic resource-constrained scheduling problem, which minimizes latency given a limited number of functional units of each type. It is an NP-hard problem which can be optimized exactly with integer linear programming (ILP). However, it is typically approximated using heuristics for better scalability. One heuristic is list scheduling, a constructive algorithm that sorts ready operations based on an established priority and schedules them one clock cycle at a time considering resource availability [27]. It is a fast local optimization algorithm for minimizing latency under resource constraints, albeit sub-optimally. State-of-the-art HLS tools typically employ the more versatile scheduling heuristic based on system of integer difference constraints (SDC) [8]. SDC-based scheduling is rooted in a linear programming formulation and can globally optimize over design constraints that can be represented in the integer difference form (e.g., cycle time constraints, latency constraints). Notably however, resource constraints must be heuristically transformed into integer difference form to be considered. As a result, SDC-based scheduling is unable to optimally handle resource constraints.

While scheduling heuristics are fast and scalable, they are fundamentally inexact with no guarantee on optimality. First, scheduling heuristics are designed to consider only a restrictive set of constraints and are unable to handle more complex scheduling problems. Second, they lack the ability to perform global optimization and may miss valuable optimization opportunities that can otherwise be discovered by exact techniques. In some cases, these challenges introduce a quality-of-results (QoR) gap whose severity remains unknown to both the designer as well as the tool itself. This gap may be exacerbated as the quantity and variety of constraints increase for HLS to accommodate emerging application domains.

To address these challenges, we propose a scheduling formulation based on SDC coupled with Boolean satisfiability (SAT) to exactly

FPGA '18, February 25–27, 2018, Monterey, CA, USA

ACM ISBN 978-1-4503-5614-5/18/02…$15.00
https://doi.org/10.1145/3174243.3174268

model a rich set of scheduling constraints. Inspired by satisfiability modulo theory (SMT) [13], our proposed approach exploits the efficiency of SDC while leveraging the scalability of modern SAT solvers to quickly prune away infeasible schedule space and derive optimal schedule. Our scheduling technique aims to push the limit on what is practically scalable with exact scheduling as well as the variety of constraints that can be efficiently encoded and solved. Our specific contributions are as follows:

(1) We propose a novel resource-constrained scheduling formulation, which combines SDC and SAT problems, to exactly and efficiently encode both resource and timing constraints in HLS.
(2) We devise an exact yet fast resource-constrained scheduling algorithm for HLS based on conflict-driven learning by leveraging the efficiency of SDC and scalability of modern SAT solvers.
(3) We employ problem-specific knowledge to specialize our scheduling algorithm to enable optimization and incremental scheduling techniques that further improve scalability.
(4) We apply our specialized scheduler within the open-source HLS tool LegUp to efficiently synthesize high-quality RTL for a range of representative benchmarks targeting FPGAs.

The rest of this paper is organized as follows: Section 2 provides background on scheduling and relevant theories, as well as motivation for our approach; Section 3 details our scheduling formulation; Section 4 describes our specialized conflict-driven scheduler; Section 5 presents experimental results; Section 6 provides related work and additional discussions, followed by conclusions in Section 7.

2 PRELIMINARIES

A typical HLS flow employs a software compiler (e.g., LLVM, GCC) to compile the input high-level program into a control data flow graph (CDFG) on which scheduling is then performed. In this paper, we focus on the resource-constrained scheduling problem, which is also a classic optimization problem in operation research. In the context of HLS, the problem is described as follows:

Given: *(1)* A CDFG $G(V_G, E_G)$ where V_G represents the set of operations in the CDFG and E_G represents the set of edges; *(2)* A set of scheduling constraints, which may include dependence constraints, resource constraints, cycle time constraints, and relative timing constraints.

Objective: Construct a minimum-latency schedule so that every operation is assigned to at least one clock cycle while satisfying all scheduling constraints.

We illustrate the three types of scheduling formulation using the data flow graph (DFG) in Figure 1(a). As our running example, we would like to schedule the DFG targeting a clock period T_{clk} of `5ns`. We assume that each `add` or `store` operation incurs a delay of `1ns`, and each load operation incurs a delay of `3ns`. We further assume that only two memory read ports are available, so at most two `load` operations can be scheduled within the same cycle. `add` and `store` operations are unconstrained.

2.1 SDC-Based Formulation

SDC is a system of inequality constraints in the integer difference form $x_i - x_j \leq b_{ij}$, where b_{ij} is an integer, and x_i and x_j are variables. The system is feasible if there exists a solution that satisfies all inequalities in the system. Because of the restrictive form of the constraints, SDC can be solved efficiently. For SDC-based scheduling [8], a schedule variable s_i is declared for each operation i in the CDFG to denote the clock cycle at which operation i is scheduled. All SDC scheduling constraints are then expressed in the integer difference form so that the system consists of a totally unimodular constraint matrix over which an optimal integer solution can be guaranteed in polynomial time. For resource-constrained scheduling, we minimize

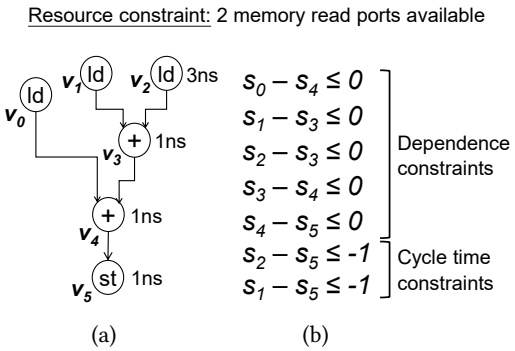


Figure 1: Motivational and running example for this paper — *(a)* DFG for our example. Delay of each operation type is indicated next to the corresponding node. Resource constraint denotes that only two memory read ports are available. No resource constraints are imposed on `add` or `store` operations. *(b)* Dependence constraints and cycle time constraints corresponding to the DFG for a target clock period of `5ns`.

the objective l such that $l > s_i \ \forall i$, where l represents the latency of the design.

To handle data dependence, SDC creates the following difference constraint for each data edge from operation i to operation j in G.

$$s_i - s_j \leq 0 \tag{1}$$

In our example, because there is an edge from node v_0 to node v_4, SDC will impose the difference constraint $s_0 - s_4 \leq 0$ to ensure that v_4 is scheduled no earlier than v_0. Similar constraints are constructed for other data dependence edges. To honor the target clock period T_{clk}, SDC identifies the maximum critical combination delay $D(ccp(v_i, v_j))$ between pairs of operations i and j and constructs the following different constraint to ensure that the combinational path with total delay exceeding the target cycle time T_{clk} must be partitioned into $\lceil D(ccp(v_i, v_j))/T_{clk} \rceil$ number of clock cycles.

$$s_i - s_j \leq -(\lceil D(ccp(v_i, v_j))/T_{clk} \rceil - 1) \tag{2}$$

In our example, because the maximum critical delay from v_2 to v_5 ($D(ccp(v_2, v_5)) = 6ns$) exceeds the target clock period of `5ns`, SDC will impose the constraint $s_2 - s_5 \leq -1$ to ensure that v_5 is scheduled at least one cycle after v_2. Similar constraints are imposed for combinational paths from v_1 to v_5 and v_0 to v_5. The aforementioned dependence and cycle time constraints are indicated in Figure 1(b).

While SDC is able to model timing constraints exactly, it must heuristically transform resource constraints into the integer difference form by imposing a particular heuristic linear ordering on the resource-constrained operations. This process separates resource-constrained operations appropriately into different cycles to ensure that sufficient resources are available to execute operations scheduled within the same cycle. The linear ordering consists of a set of precedence relationships between pairs of resource-constrained operations i and j represented in the form of

$$s_i - s_j \leq -L_i \tag{3}$$

where L_i denotes the latency (in cycles) of operation i. Although the linear ordering results in a legal schedule that satisfies all resource constraints, the schedule is likely sub-optimal because the linear ordering is devised heuristically. There are many possible such legal linear orderings, some resulting in better schedules than others. However, SDC can simply pick one particular linear ordering heuristically and without knowledge of whether it is optimal.

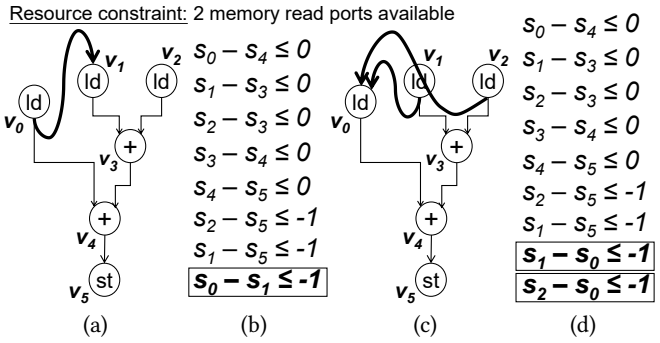


Figure 2: Partial ordering edges are heuristically imposed on the DFG, and subsequently in the SDC, to satisfy the resource constraints — Partial ordering edges are shown in bold, and corresponding difference constraints are boxed. *(a)-(b)* represent a different combination of partial ordering edges than *(c)-(d)*. Minimum latency differs depending on the particular combination.

For our example, SDC must impose partial orderings among the resource-constrained `load` operations because only two memory read ports are available for the three `load` operations (v_0, v_1, and v_2). On one hand, SDC can impose an edge from v_0 to v_1 as shown in bold in Figure 2(a) to separate v_0 and v_1 into different cycles so that each cycle has at most two `load` operations. With this heuristic partial ordering, the DFG requires at least three cycles to execute due to the critical path delay from v_0 to v_5. Given the target clock period of `5ns`, v_0 and v_1, each of which incurs a delay of `3ns`, must be scheduled in separate cycles given the partial ordering edge between them. v_5 cannot be scheduled in the same cycle as v_1 because there is no slack remaining in the clock cycle after scheduling v_3 and v_4. On the other hand, if SDC instead imposes an edge from v_1 to v_0 and another edge from v_2 to v_0 as shown in bold in Figure 2(c), the DFG can achieve a better latency of only two cycles while ensuring that each cycle has at most two `load` operations. In Figure 2, corresponding SDC constraints are shown in (b) and (d), respectively, with appended partial ordering ("resource") constraints boxed.

From this example, we see that it is necessary to enumerate all possible combinations of partial orderings and solve an SDC for each combination of imposed "resource" edges to find the optimal (minimum-latency) schedule. However, attempting all combinations is not scalable in the general case for an arbitrary number of resource-constrained operations. For this reason, SDC heuristically imposes one particular partial ordering without guarantee of optimality and proceed with solving the scheduling problem without regards to the effect of any sub-optimality on the solution.

2.2 ILP-Based Formulation

Applying ILP in the context of resource-constrained scheduling problem has been a well-studied topic [24]. ILP is a linear program with linear objective and constraints in which all variables are restricted to be integers. For the ILP-based formulation, we focus on the special case of 0-1 ILP in which all variables are binary. The formulation declares a binary variable x_{it} to denote whether operation i starts at clock cycle t, where i and t are integers bounded by the total number of operations and maximum allowable latency, respectively. With these binary variables, the start time s_i of operation i can be expressed as

$$s_i = \sum_{t=0}^{L-1} t \cdot x_{it} \tag{4}$$

where L denotes the maximum latency. Because s_i is analogous to the corresponding schedule variable in SDC, dependence constraints in ILP can be equivalently represented as the difference between pairs of schedule variables as in Eq. 1. For our example, we can safely assume a maximum start time equal to the number of operations $N = 6$. It follows that we declare variables $\{x_{00}, x_{01}, x_{02}, x_{03}, x_{04}, x_{05}\}$ for operation v_0 and denote that $s_0 = \sum_{t=0}^{6-1} t \cdot x_{0t}$. Variables are similarly declared and derived for operations v_1 to v_5. The objective is same as that defined in Section 2.1 for the SDC formulation.

Unlike in SDC, resource constraints can be encoded exactly as linear constraints in ILP. To ensure that the number of active operations of type r in clock cycle t does not exceed the number of available type-r resources a_r, the ILP formulation imposes the resource constraint

$$\sum_{i:RT_i=r} \sum_{t'=t-L_i}^{t} x_{it} \leq a_r \tag{5}$$

where RT_i and L_i denote the resource type and latency of operation i, respectively. For our example, the ILP formulation needs to impose the constraints $\sum_{i=0}^{2} x_{it} \leq 2$ for each clock cycle t because only two memory ports are available. These constraints apply to the resource-constrained `load` operations v_0, v_1, and v_2 (i.e., $i = 0, 1, 2$). The second summation is omitted because the latency of `load` operation is zero-cycle in our example. The ILP formulation also requires the following unique start time constraint for each operation i to ensure that operation i starts at only one particular clock cycle.

$$\sum_{t} x_{it} = 1 \tag{6}$$

While modern ILP solvers can handle problems of non-trivial size, ILP is in general NP-hard and difficult to scale. In comparison to SDC for scheduling, ILP requires significantly more variables for encoding the same problem and cannot take advantage of special matrix structure to efficiently solve the problem.

2.3 SAT-Based Formulation

SAT stands for the Boolean satisfiability problem, which determines if there exists an assignment of the Boolean variables that satisfies a Boolean formula. A SAT problem consists of a set of Boolean clauses, all of which must be satisfied by some assignment of the Boolean variables for the problem to be satisfiable. The problem is unsatisfiable otherwise. In general, a SAT-based scheduling formulation [16] uses Boolean variable x_{it} to denote whether operation i starts at clock cycle t, and employs Boolean variable u_{it} to denote whether operation i is active at clock cycle t. Dependence and resource constraints can be expressed as clauses of these variables.

Modern SAT solvers perform systematic search based on variations of the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [12] of *decide*, *propagate*, and *backtrack*. These solvers recursively *decide* the value (true or false) of an unassigned variable, *propagate* the effects of this decision using deduction rules, and *backtrack* if conflicts dictate that a different value should be attempted for the variable. In particular, conflict-driven SAT solvers complements DPLL with extra features to achieve significant improvement in efficiency. Extra features may include clause learning, non-chronological backtracking, adaptive branching, unit propagation, and random restart [36]. Although SAT remains a well-known NP-complete problem, SAT procedures based on the DPLL algorithm have demonstrated scalability with hundreds of thousands of variables and clauses [23]. In the domain of design automation, SAT has been successfully applied to solve problems in hardware/software model checking, test pattern generation, equivalence checking, etc.

However, it is interesting to note that although the scheduling problem can be encoded completely in SAT, the encoding is often too large and too inefficient even considering the capability of modern SAT solvers [26]. Moreover, SAT is only concerned with whether the problem is satisfiable and does not inherently support optimization of an objective, such as minimizing latency.

3 JOINT SDC AND SAT SCHEDULING

In resource-constrained scheduling, there has always been an inherent tension between scalability and quality. On one hand, heuristic scheduling is fast and scalable, but generates sub-optimal QoR. On the other hand, exact scheduling creates optimal QoR, but is slow and difficult to scale. As described in Section 2, the SDC heuristic achieves fast runtime but generates sub-optimal schedule because resource constraints cannot be represented exactly with integer difference constraints. The ILP-based formulation can model both timing and resource constraints exactly but is not scalable in general. As a result, resolving the tension between scalability and quality is key to achieving both global optimization and fast runtime.

To this end, we propose a scheduling algorithm that integrates SDC and SAT to exactly handle different types of constraints and optimally solve the resource-constrained scheduling problem defined in Section 2. To achieve global optimization, our algorithm leverages SDC to represent constraints that can be readily expressed in the integer difference form and employs SAT to encode constraints that do not naturally fall under the SDC framework. A joint SDC and SAT formulation allows us to leverage the advantages of SDC and SAT while exactly encoding both timing and resource constraints.

Figure 3 shows the high-level structure of our scheduler, mainly composed of a conflict-based SAT solver integrated with a graph-based SDC solver. On the left, the SAT solver takes advantage of conflict-based search (detailed in Section 3.1) to quickly propose partial orderings that satisfy the resource constraints. These partial orderings are converted to SDC constraints and appended to the SDC problem. On the right, the SDC solver leverages a graph-based algorithm (detailed in Section 3.2) to efficiently check the feasibility of the proposed partial orderings. Any infeasibility will be encoded as a conflict clause in SAT and appended back into the SAT problem. The solver iterates between SAT and SDC until it finds a feasible solution or proves that such solution does not exist.

Because a particular binding (set of partial orderings) proposed by SAT may not be consistent with the given SDC timing constraints, it is necessary to communicate any SAT binding decision to the SDC so that constraints in SDC and SAT are jointly considered. At the same time, any infeasibility must be communicated back from SDC to SAT so that SAT can learn from the mistakes of its previous proposals and make better proposals in the future. This process of conflict-driven learning is key to enabling accelerated convergence of our proposed scheduler. It is important to note that despite the benefits of conflict-driven learning, the problem remains NP-hard. Nevertheless, our approach demonstrates better efficiency and scalability than ILP. While our approach is inspired by and bears resemblance to SMT, we will discuss the key differences in Section 6.

3.1 SAT for Resource Constraints

As shown in Figure 3, our algorithm leverages SAT to model the resource constraints based on which partial orderings are proposed. In our formulation, let binding variable B_{ik} denote whether operation i is bound to resource instance k. We employ one binding variable to denote the binding of each resource-constrained operation to each resource instance. For our example, operations v_0, v_1, and v_2 are resource-constrained `load` operations, each of which can be bound to one of two memory read ports (i.e., $k = 0, 1$). Therefore,

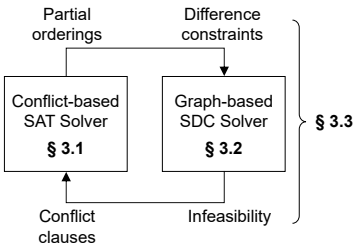


Figure 3: Overall structure of our scheduler — Composed of a SAT solver integrated with an SDC solver to enable conflict-driven learning. This solver checks the feasibility of a particular latency. Latency optimization (Section 3.4) is built on top of this solver.

we declare $\{B_{00}, B_{01}, B_{10}, B_{11}, B_{20}, B_{21}\}$ for the different operation-resource pairs. By adding the appropriate clause $\sum_k B_{ik} = 1\ \forall i$ to enforce that each operation is bound to exactly one resource, the binding variables are responsible for assigning each operation to a resource instance without exceeding the resource availability.

Based on the definition of binding variable, a sharing variable R_{ij} can be derived to denote whether operation i is sharing the same resource with operation j. For each pair of operations (i, j) mapped to the same type of resource,

$$R_{ij} = \bigvee_{k \in T} (B_{ik} \wedge B_{jk}) \tag{7}$$

where T denotes the set of resources of the particular type. R_{ij} is true if both operations i and j are bound to the same resource instance by the binding variable. With R_{ij}, we can then define the partial ordering variable $O_{i \to j}$ to denote whether operation i is scheduled in an earlier cycle than operation j. $O_{i \to j}$ maps to integer difference constraint in SDC between i and j as follows:

$$O_{i \to j} = True \ \mapsto \ s_i - s_j \leq -1 \tag{8}$$

$$O_{i \to j} = False \ \mapsto \ \emptyset \tag{9}$$

As shown in Eq. (8), assigning $O_{i \to j}$ to true dictates that operation i must be scheduled in an earlier cycle than operation j and therefore maps to the difference constraint $s_i - s_j \leq -1$. As shown in Eq. (9), assigning $O_{i \to j}$ 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 no particular partial ordering is required by the proposed resource binding. Given the mapping between SAT and SDC, we include the following partial ordering clauses in SAT for each pair of operations (i, j) mapped to the same type of resource.

$$R_{ij} \rightarrow (O_{i \to j} \vee O_{j \to i}) \tag{10}$$

$$\neg (O_{i \to j} \wedge O_{j \to i}) \tag{11}$$

Eq. (10) indicates that if operation i and j shares the same resource instance, it implies that operation i must be scheduled either in an earlier cycle or in a later cycle than operation j. Eq. (11) ensures that operation i cannot be simultaneously scheduled both in an earlier cycle and later cycle than operation j.

Figure 4(a) shows the partial ordering clauses for our problem where a pair of clauses is specified for every combination of resource-constrained `load` operations (v_0, v_1, and v_2). Among other types of clauses described, only the partial ordering clauses are shown because they contain the partial ordering variables to be mapped to SDC. In this figure, for example, the first clause indicates that if v_0 and v_1 share the same resource instance, v_0 must be scheduled either in an earlier cycle or in a later cycle than v_1, and not both. A similar line of logic follows with the other clauses in the figure. SAT clauses like these (e.g., Eq. (7), (10), (11)) can be translated into conjunctive normal form commonly accepted by SAT solvers. Subsequently, the resulting assignments of $O_{i \to j}$ and $O_{j \to i}$ satisfying these clauses

will be mapped to integer difference constraints or lack thereof in SDC based on Eq. (8) and (9). For instance, $O_{0\to1}$ assigned to true will be mapped to $s_0 - s_1 \leq -1$.

3.2 SDC for Timing Constraints

As shown in Figure 3, our algorithm uses SDC to solve the difference constraints, which consist of incoming partial ordering constraints from SAT and the original set of timing constraints (e.g., dependence and cycle time constraints) of the problem previously shown in Figure 1(b) and reproduced for convenience in Figure 4(b). From Figure 4(b), we see the difference constraints can be conveniently represented using a constraint graph where each variable maps to a node and each constraint maps to an edge. The constraint graph contains edges to represent dependence constraints and cycle time constraints. Inequalities whose right-hand side is `0` represent dependence constraints, while those whose right-hand side is `-1` represent cycle time constraints, both described in Section 2.1. For each of these constraints in integer difference form $s_u - s_v \leq d_{u,v}$, the constraint graph includes an edge of weight $d_{u,v}$ from node v to u. For clarity, weights are omitted for zero-weight edges.

By representing SDC as a constraint graph, we can detect infeasibility of the difference constraints by the presence of negative cycle in the graph. This property will be useful for checking whether the proposed partial orderings from SAT are consistent with the given SDC timing constraints. In addition, the negative cycle serves as a certificate of any inconsistency between the proposed resource binding and given timing constraints. In Section 3.3, we will describe how we leverage the negative cycle to provide feedback from SDC to SAT for enabling conflict-driven learning. Furthermore, we can obtain a feasible schedule, either as late as possible (ALAP) or as soon as possible (ASAP) schedule, by solving a single source shortest path problem on the graph. ASAP schedules all operations to the earliest possible clock cycle, and ALAP schedules all operations to the latest possible clock cycle given a latency constraint.

In our solver, it is necessary to detect whether the addition of each partial ordering edge induces a negative cycle in the constraint graph. However, it is wasteful to solve the entire SDC with all nodes and edges for each edge added when only a small part of graph is affected by the addition. Doing so cuts directly into the bottom line of our scheduler because SDC is a crucial component of conflict-driven learning. Quick propagation and convergence of the scheduler rely on having a highly efficient SDC solver and a method to quickly identify any negative cycle in the constraint graph. To accelerate the process of conflict identification in SDC, we propose to leverage an efficient incremental algorithm for maintaining a feasible solution and detecting negative cycle for a dynamically changing SDC constraint graph [28].

To enable incremental SDC solving, our scheduler initializes with a feasible solution (shortest path solution) of the original graph (without partial ordering edges). For each edge added to the constraint graph or each tightened edge weight, the algorithm traverses only the affected subgraph and update the distances of only affected nodes. This incremental update guarantees that the updated node values continue to maintain a feasible solution. Because the algorithm is essentially applying Dijkstra's algorithm to modify only affected edges and nodes, the addition (or tightening) of a constraint incurs a marginal time complexity $O(\Delta e + \Delta v \log \Delta v)$, where Δe and Δv denote the number of affected edges and nodes, respectively. The algorithm is able to delete or relax an edge in constant time. Because deletion or relaxation results in a less constrained system, the current feasible solution remains feasible.

Using the incremental SDC algorithm, our scheduler inserts one edge at a time until the constraint graph becomes infeasible. The algorithm detects such infeasibility when the distance of the source node of the inserted edge is updated during the traversal of the affected subgraph. This indicates a negative cycle in the affected subgraph because the distances of the nodes will continue decrease as long as we continue to traverse the subgraph. At this point, our algorithm traces backward on the predecessors along the shortest path computed by Dijkstra's algorithm to extract the edges involved in the negative cycle. Our algorithm then reports partial ordering edges in the negative cycle back to SAT because SAT is concerned with resource-related partial orderings. Other edges represent hard constraints and are not influenced by SAT.

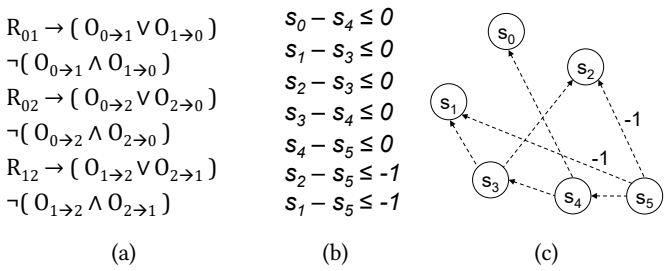


Figure 4: Constraints for our running example — *(a)* Resource constraints in SAT. *(b)* Timing constraints in SDC. *(c)* Corresponding SDC constraint graph.

3.3 Conflict-Driven Learning

As shown in Figure 3, SAT and SDC interact closely within a feedback loop to enable conflict-driven learning. For each iteration of the loop, SAT proposes partial orderings that satisfy the SAT clauses described by Eq. (10) and (11). These partial orderings are converted to SDC constraints based on Eq. (8) and (9) and appended to the SDC problem. SDC then checks the feasibility of the proposed partial orderings and report any infeasibility as a conflict clause back to the SAT.

We illustrate the power of conflict-driven learning in Figure 5 using our running example. Here we would like to determine if the DFG in Figure 1(a) can be scheduled within two cycles. The corresponding SAT formulation for resource constraints is reproduced on the top of Figure 5(a), while the initial SDC constraint graph for timing constraints is shown on the bottom. As the solver progresses, resource-related edges mapped from the partial ordering variables will be added to the constraint graph in a manner similar to that of timing constraints described in Section 3.2. It is important to note that the constraint graph contains a latency edge of weight `1` from s_0 to s_5 to indicate a maximum allowable clock cycle index of `1` for our target two-cycle schedule starting with cycle `0`.

To solve the feasibility problem of determining whether the graph can be scheduled within two cycles, SAT starts with an initial proposal of the assignment of the partial ordering variables as shown on the top of Figure 5(b). For clarity, we show only partial ordering variables that are assigned to `True` because they are the ones that will influence the constraint graph. On the bottom of the figure, SDC adds the corresponding edges (shown with solid lines) proposed by SAT into the constraint graph. With these additional edges, SDC detects a negative cycle (shown in bold) among the initial edges and the partial ordering edge from $O_{0\to1}$. SDC then reports the conflict back to SAT using the conflict clause $\neg O_{0\to1}$ to ensure that any partial ordering involving v_0 before v_1 should no longer be proposed by SAT. As shown in Figure 5(c), after the conflict clause is added to the SAT problem, SAT makes a different proposal based on the updated set of clauses. In this case, SDC detects a different negative cycle involving the edge proposed by $O_{0\to2}$ and adds the

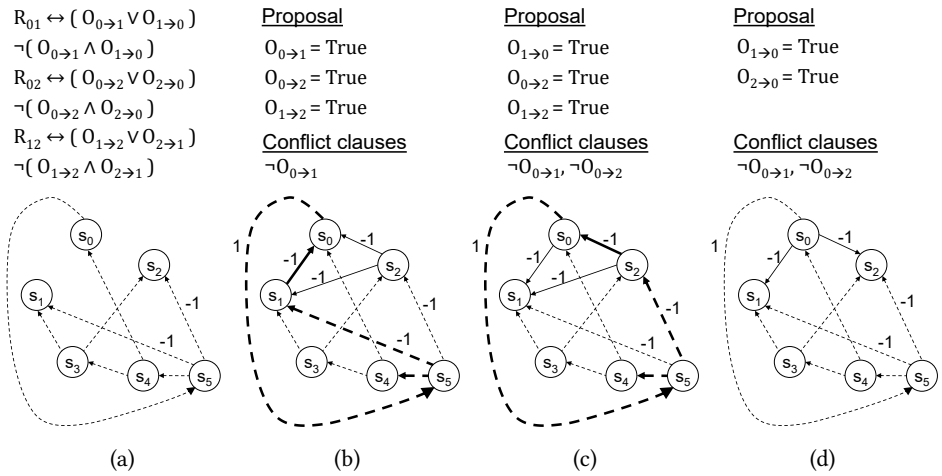


Figure 5: Illustration of conflict-driven learning with SDC and SAT using our running example from Figure 1 — *(a)* Resource constraints in SAT on the top and initial SDC constraint graph on the bottom. *(b)-(d)* The progression of joint SAT and SDC scheduling. Corresponding partial ordering proposals by SAT are shown on the top. For conflict clauses, ¬ denotes negation of the SAT variable. For constraint graphs, dashed lines represent hard constraints. Solid lines represent partial ordering constraints proposed by SAT. Bold lines trace negative cycles.

conflict clause $\neg O_{0\to2}$ to the SAT. During conflict-driven scheduling, a negative cycle indicates that the resource binding proposed by SAT is inconsistent with the (hard) timing constraints of the problem. No schedule is able to achieve the desired latency while satisfying both the timing constraints and the proposed resource binding. As a result, a different resource binding needs to be attempted.

Based on the feedback up until this point from SDC, conflict clauses dictate that any schedule with v_0 before v_1 or v_0 before v_2 will be infeasible and need not be attempted. Notice that these conflict clauses are short, allowing SAT to prune out a large search space because it no longer needs to propose any combination involving these infeasible orderings. Shorter conflict clauses lead to a larger search space that can be pruned and therefore faster propagation and convergence for our scheduler. As such, it is crucial to derive conflict clauses that are as short as possible. Negative cycle satisfies this property because it is guaranteed to be an irreducibly inconsistent set of constraints [34]. It is a minimal set of inconsistent constraints in which the removal of any edge in the negative cycle will also remove the negative cycle in its entirety.

With two short conflict clauses, SAT has a much better understanding of the search space. As shown in Figure 5(d), SAT now makes a proposal whose corresponding edges no longer generate any negative cycle in the constraint graph. Because the constraint graph is now feasible, SDC returns a feasible solution that satisfies all timing and resource constraints. For efficiency, our scheduler uses the shortest path distances of the constraint graph as the feasible solution because the shortest path has already been computed in the process of detecting negative cycle.

3.4 Minimizing Latency

Because SAT has its root in decision problems, we have so far limited our discussion to checking the feasibility of a particular latency value. To minimize latency as in the case of resource-constrained scheduling, we propose to perform binary search over the range of possible latency values based on an initial upper and lower bound. During the binary search, we solve a series of feasibility problems as described in Section 3.3, each of which returns either a feasible solution or a proof that the problem is infeasible. A feasible answer allows our scheduler to decrease the upper bound, while an infeasible answer requires increasing the lower bound. The binary search terminates when the upper and lower bounds coincide.

Because the convergence of the scheduler depends on the number of latency values the binary search needs to process, we propose to leverage specialized knowledge we can obtain for the scheduling problem to establish upper and lower latency bounds to reduce the range of latency values that need to be searched. Specifically, we propose to leverage the original SDC heuristic scheduling algorithm [8] for upper bounding to establish a good initial solution that has already globally optimized over a subset of constraints. Furthermore, we propose to apply the resource-aware lower bounding algorithm [29] (described later in Section 4.1) to establish a lower bound so that the scheduler does not waste time exploring too many unmeaningful latency values. While the upper and lower bounds are not necessarily tight, they provide a good starting point from which exact scheduling can initialize.

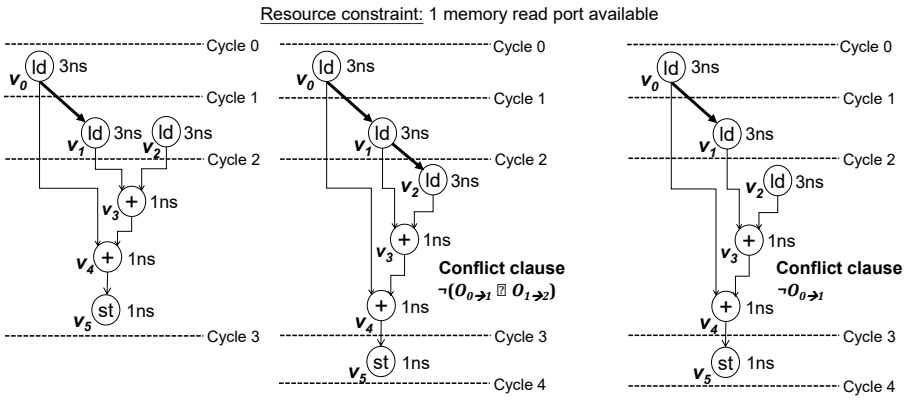
4 SCHEDULER SPECIALIZATION

As mentioned in Section 3.4, it is possible to extract knowledge we have specific to the resource-constrained scheduling problem to further reduce the search space and improve runtime. In this section, we describe how we leverage various heuristics to specialize our scheduler for the scheduling problem. These techniques maintain the exactness of the algorithm and the optimality of the solution.

4.1 Resource-Aware Lower Bounding

Resource-aware lower bounding applies a greedy algorithm to solve a relaxed version of the resource-constrained scheduling problem [29]. While the algorithm eliminates dependence constraints for the relaxation, it uses the ASAP schedule to determine the earliest clock cycle each operation can be scheduled and minimizes the tardiness of each operation in respect to the ALAP schedule. The greedy algorithm selects the operation with minimum ALAP value and assigns it to the earliest clock cycle based on the ASAP schedule and resource constraints. This process continues until all operations have been scheduled. The resulting lower bound is determined by adding the maximum tardiness (in cycles) among all operations to the critical path latency for the entire design, which considers only dependence.

While we have discussed in Section 3.4 the application of resource-aware lower bounding to establish tighter lower bound in optimization, the same exact algorithm can be helpful for accelerating the propagation for conflict-driven learning described in Section 3.3. Recall that partial ordering edges are inserted one-by-one into the SDC constraint graph until the graph becomes infeasible. The fewer the number of inserted partial ordering edges, the shorter the conflict clause and larger the search space that can be pruned by SAT based on the conflict clauses. In addition to detecting negative cycle, our scheduler can also incrementally determine the lower bound



(a) SDC

(b) Lower bounding

Figure 6: Illustration of the advantage of lower bounding over SDC in conflict-driven learning — Assume one memory read port and $T_{clk} = 5ns$. Actual DFGs, instead of constraint graphs, are shown in these figures. *(a)* SDC requires two "resource" edges (in bold) to determine that the DFG requires at least 4 cycles. *(b)* The lower bounding algorithm requires only one edge to determine the same 4-cycle latency because it pushes v_2 to the next cycle due to resource constraint.

upon the insertion of each new edge. After identifying the first edge that results in an infeasible system, our scheduler uses the deletion filtering algorithm [6] to remove previously added edges that do not contribute to the infeasibility. An edge does not contribute to the infeasibility if the graph remains infeasible even after the edge has been removed. The remaining set of edges then compose an irreducibly inconsistent set of constraints. Because the lower bounding algorithm is aware of the limited resource availability, it is actually able to prove infeasibility, in certain cases, with fewer partial ordering edges than SDC which has no sense of resource constraints other than those imposed by partial ordering. As such, lower bounding improves solution space pruning during conflict-driven learning.

We illustrate one such case in Figure 6 with the same DFG as in Figure 1(a). Here we would instead like to determine if the DFG can be executed within three cycles, assuming one memory read port and a target clock period of 5ns. To separate the resource-constrained load operations (v_0, v_1, and v_2) into different cycles due to the availability of only one read port, let's further assume that partial ordering edges are added in the order corresponding to partial ordering variables $\{O_{0\to1}, O_{1\to2}\}$. In Figure 6, note that edges are shown within the DFG instead of the constraint graph. With the first partial ordering edge from v_0 to v_1 in Figure 6(a), SDC is unable to rule out the feasibility of executing the DFG in three cycles. Because SDC is unaware of the number of available read ports, it schedules v_2 in the same cycle as v_1. Only with the second edge from v_1 to v_2, as shown in Figure 6(a), does SDC pushes v_2 to the next cycle and realize that the DFG requires at least four cycles. Because the DFG cannot complete in three cycles with the two edges, SDC will return the conflict clause $\neg(O_{0\to1} \wedge O_{1\to2})$ to reflect the irreducibly inconsistent set of two edges. SDC requires both partial ordering edges (the complete resource binding) to decide infeasibility.

With resource-aware lower bounding, however, it is possible to determine that the DFG requires at least four cycles after adding only the first partial ordering edge from v_0 to v_1. As demonstrated in Figure 6(b), the algorithm does not attempt to schedule v_2 in the same cycle as v_1 even without the second edge, because the algorithm is aware that only one read port can be used in each cycle. As a result, v_2 is pushed to the next cycle, increasing the latency to at least four cycles. With lower bounding, the scheduler generates a more concise conflict clause $\neg O_{0\to1}$ for this example, which enables more effective pruning of the search space in SAT. Lower bounding is able to determine infeasibility with only a partial resource binding, thus resulting in speedup.

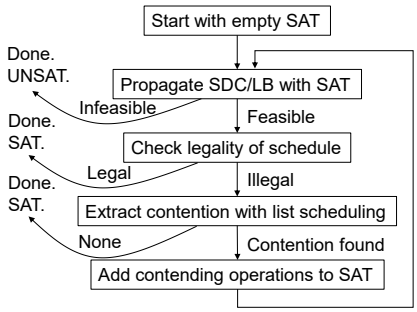


Figure 7: Incremental learning flow — Starts with no resource constraints and incrementally imposes resource constraints on operations that have encountered resource contention in previous iterations of the loop in this flow.

4.2 Incremental Learning

Because the proposed SAT formulation in Section 2.3 includes variables for all resource-constrained operations, conflict-driven learning described in Section 3.3 considers all resource-constrained operations equally. In reality, however, some operations tend to be located in congested region of the schedule and must compete for a very limited number of resources within a limited number of time steps. Other operations do not fall in the congested region and can be freely scheduled. The congested region constitutes the problematic part of the schedule because there are more operations that need to be scheduled than the number of available resources for these operations. As a result, it would be more effective to emphasize our SAT's resource constraints over operations that are likely to encounter resource contention and allow non-contending operations to be scheduled by SDC's (hard) timing constraints only. This approach attempts to reduce the size of the NP-hard part of the problem and leverages SDC as much as possible in finding a feasible schedule.

To implement this idea, we propose an incremental learning mode for our scheduler. Incremental learning leverages problem-specific knowledge to specifically target operations that are likely to cause resource contention. The flow of incremental learning mode is shown in Figure 7. Based on this flow, the scheduler starts with an empty SAT formulation, with no resource constraints initially. The scheduler then performs conflict-driven learning by propagating SDC and/or lower bounding (denoted as LB in the figure) with SAT. If the SDC graph reports a negative cycle, the problem is not satisfiable even with only timing constraints. In this case, the solver returns unsatisfiable and terminates. If the SDC graph does not detect any negative cycle, which is the more likely scenario, the scheduler checks the legality of the schedule against resource constraints. If the schedule is legal, the scheduler returns with the feasible schedule. If the

Table 1: Runtimes are reported in seconds for our proposed joint SDC and SAT scheduling (SDS for short) compared to default ILP scheduling using CPLEX and CBC — `%variables`: percentage of variables in non-incremental mode activated in incremental mode; speedup of `Non-incremental` and `Incremental` shown respectively in parentheses against `CPLEX` and `CBC`. `TO`: timeout after 300 seconds. `n/a`: not applicable. `Optimal Latency`: optimal latency in clock cycles for each benchmark and represents the latency achieved by both SDS scheduler and default ILP scheduling. `LegUp Latency`: latency achieved by LegUp using SDC-based scheduling heuristic.

Benchmark	# Operations	Runtime for SDS Scheduling (sec)		Runtime for Default ILP Scheduling (sec)		Optimal Latency	LegUp Latency
		Non-incremental	Incremental (% Variables)	CPLEX	CBC		
ARAI	44	0.01	0.01 (39.5%)	0.12 (12x, 12x)	1.18 (118x, 118x)	8	9
PR	52	0.02	0.01 (31.3%)	0.86 (43x, 86x)	3.70 (185x, 370x)	12	14
WANG	54	0.01	0.01 (8.29%)	0.86 (86x, 86x)	12.2 (1220x, 1220x)	12	14
LEE	58	0.01	0.01 (3.02%)	0.26 (26x, 26x)	2.88 (288x, 288x)	12	14
MCM	74	0.54	0.34 (10.4%)	6.19 (11x, 18x)	24.6 (46x, 72x)	15	16
DIR	76	0.14	0.01 (6.18%)	1.51 (11x, 151x)	11.5 (82x, 1550x)	14	15
HONDA	105	0.02	0.02 (0.95%)	9.06 (453x, 453x)	104 (5200x, 5200x)	27	33
CHEM	349	TO	1.42 (0.12%)	TO (n/a, n/a)	TO (n/a, n/a)	85	89
U5ML	857	0.01	0.01 (0.00%)	20.8 (2080x, 2080x)	TO (n/a, n/a)	261	264

schedule is illegal, likely in the initial iterations of this flow because no resource constraints have been considered, the scheduler will extract the contending operations with a list scheduling like heuristic. During the extraction process, the heuristic attempts to reorder operations to remove resource contention. If the heuristic succeeds in removing all resource contention, the scheduler also returns a feasible schedule. If contention remains, however, the scheduler adds the clauses of those contending operations to the SAT and repeats the flow starting with another iteration of conflict-driven learning.

The ultimate goal of incremental learning is to dramatically reduce the search space and improve runtime by using well-known heuristics (e.g., list scheduling, SDC-based scheduling) to direct the search toward the more difficult region of the schedule. Nevertheless, it is important to emphasize that these heuristics are used simply to guide the solver in a more promising path toward the solution and should in no way jeopardize the exactness of the scheduler. When incremental learning returns satisfiable, it always provides a legal schedule in regards to both timing and resource constraints and satisfies the given latency bound. Incremental learning is performed for different latency bounds in the binary search manner described in Section 3.4 to determine the schedule with the optimal latency.

5 EXPERIMENTS

We implement our proposed scheduler (detailed in Section 4) in C++ interfaced with LLVM compiler and Lingeling SAT solver [2]. We execute our scheduler on an Intel Xeon CPU running at 2.50GHz, and evaluate it on a set of compute-intensive benchmarks listed in Table 1. These benchmarks include a chemical plant controller and a number of DSP algorithms such as discrete cosine transforms. We constrain the scheduling process such that these benchmarks contain a large portion of resource-constrained operations useful for stress-testing our scheduler.

Our first set of experiments aim to compare the runtimes of our scheduler against those of state-of-the-art commercial and open-source ILP solvers. A comparison of runtime results between our joint SDC and SAT scheduling (SDS for short) and default ILP scheduling is shown in Table 1. For our SDS scheduler, we provide results for the scheduler running in non-incremental mode and in incremental mode. `Non-incremental` column provides results from applying conflict-driven learning from Section 3.3 with the full set of SAT variables. `Incremental` column provides results from applying incremental learning from Section 4.2 by selectively targeting a subset of SAT variables. For default ILP scheduling, the formulation presented in Section 2.2 is solved in CPLEX [9], a state-of-the-art commercial ILP solver, as well as in CBC [14], a best-in-class open-source ILP solver. Speedup values achieved by non-incremental and incremental modes against each ILP solver are shown respectively in parentheses in the corresponding columns.

Table 2: Runtimes in seconds for different combinations of resource constraints on multiplier and memory port — Results are shown for SDS scheduling in incremental mode.

Benchmark	# Operations	Runtime for Incremental Scheduling (sec)				
		1 mult 1 port	2 mult 2 port	3 mult 3 port	4 mult 4 port	6 mult 6 port
ARAI	44	0.01	0.01	0.01	0.01	0.01
PR	52	0.01	0.01	0.01	0.01	0.02
WANG	54	0.01	0.01	0.01	0.01	0.02
LEE	58	0.01	0.01	0.01	0.01	0.02
MCM	74	0.05	0.34	0.01	0.13	0.07
DIR	76	0.02	0.01	0.01	0.01	0.01
HONDA	105	0.01	0.03	0.04	0.09	0.24
CHEM	349	1.49	1.42	1.10	2.92	4.33
U5ML	857	0.01	0.01	0.01	0.01	0.01

Based on the results in Table 1, SDS scheduler running in non-incremental mode is faster than the open-source ILP solver by around two orders of magnitude and sometimes three orders of magnitude in all cases except `CHEM` for which both solvers time out. In non-incremental mode, SDS scheduler can also beat the commercial ILP solver by at least one order of magnitude, and up to two or three orders of magnitude for the same set of benchmarks. These results demonstrate the effectiveness of setting upper and lower latency bounds and exploiting negative cycle and lower bounding in propagation to quickly prune out the entire search space. It is interesting to note that benchmark `U5ML` achieves a low runtime because it is much more constrained by timing than by resource. Timing constraints dictate that its latency cannot be further reduced regardless of resource assignment.

With incremental mode enabled, Table 1 shows that SDS scheduler is able to complete the previously difficult benchmark `CHEM` and locate the optimal solution while both the commercial and open-source solvers struggle and time out. At the same time, incremental mode also improves the runtime of other benchmarks by various degrees. The improvement from incremental mode stems from the fact that only a small fraction of operations are actually involved in resource contention. Based on Table 1, mostly less than 10% of the SAT variables are needed to resolve resource constraints and converge to an optimal solution. The percentage becomes small for large benchmarks. With problem-specific knowledge, we specifically target contending operations to achieve significant speedup.

Table 2 shows the runtime in seconds for different combinations of constraints on the number of multipliers and memory ports. In general, an increase in the number of resources leads to additional SAT binding variables while the number of SAT partial ordering variables remains unchanged. The overall increase in the number of SAT variables may lead to longer runtime for the SAT solver. However, increasing the number of resources also loosens the resource constraints and decreases the number of partial ordering edges that

Table 3: Experiments on synthesizing CHStone benchmarks targeting the Intel Cyclone V FPGA at a clock period of `10ns` — `#Ops`: number of operations in the program. `#States`: number of states in the generated schedule for each function; benchmarks achieving state reduction with SDS are highlighted in bold. `CP`: achieved clock period in `ns`. `ALM`, `LUT`, `FF`, `DSP`, and `RAM`: number of corresponding resources used on the target device. `Runtime`: time in seconds taken to solve the SDS scheduling problem.

Benchmark	#Ops	SDC-Based Scheduling							SDS Scheduling							
		#States	CP	ALM	LUT	FF	DSP	RAM	Runtime	#States	CP	ALM	LUT	FF	DSP	RAM
ADPCM	850	25, 58	11.2	5316	8948	9851	122	7	0.03	25, **54**	12.5	5549	9166	9894	146	7
AES	812	37, 25, 17, 46	10.6	5313	7817	9568	0	10	0.05	37, 25, 17, **42**	11.5	5506	8147	9755	0	10
BLOWFISH	687	74, 36	7.5	2209	3330	4035	0	29	0.02	**70**, 36	7.8	2402	3709	4582	0	29
DFADD	361	4, 4	7.3	1439	1770	2124	0	1	0.01	4, 4	7.4	1442	1778	2105	0	1
DFDIV	361	65	9.7	3179	4383	6776	48	2	0.01	65	9.6	3170	4385	6769	48	2
DFMUL	279	5	9.6	1125	1601	1494	32	1	0.01	5	9.7	1126	1630	1492	32	1
DFSIN	1067	4, 9, 65	10.2	8584	10677	14568	82	5	0.05	4, 9, 65	9.6	8541	10594	14557	82	5
GSM	966	7	10.2	3256	4747	5204	54	7	0.03	7	10.6	3233	4697	5154	62	7
JPEG	2255	36, 9, 6, 7, 9, 7, 53	13.4	17066	28087	21211	87	83	0.11	36, 9, 6, 7, 9, 7, 53	13.7	17020	28112	21016	87	83
MIPS	346	5	11.7	1036	1468	928	6	4	0.01	5	11.8	1029	1468	947	6	4
MOTION	284	4, 7	8.4	5577	8257	8495	0	6	0.01	4, 7	9.0	5729	8339	8985	0	6
SHA	314	11, 11	6.1	1350	1596	2650	0	20	0.01	11, 11	6.3	1375	1603	2687	0	20

needs to be inserted into the SDC (when the solver is running in incremental mode). The resulting set of SDC constraints are more likely to be consistent, making it easier for SDC to return a feasible solution after fewer iterations in propagation. Table 2 shows that SDS scheduler running in incremental mode remains scalable as the number of resources in the constraints increases.

To demonstrate the applicability of SDS, we further integrate SDS scheduler into LegUp [4], a state-of-the-art open-source HLS tool. We leverage LegUp's front-end to compile the input program into a CDFG and extract the relevant scheduling (e.g., timing, resource) constraints. SDS scheduler schedules the CDFG based on the constraints extracted from LegUp and returns the generated schedule to LegUp for post-scheduling processing and RTL generation. For experiments, we synthesize a set of applications from the CHStone benchmark suite [15] targeting the Intel Cyclone V FPGA at a clock period of `10ns`.

Using LegUp, we compare the QoR of the synthesized hardware produced by SDC-based scheduling against the QoR of hardware produced by our SDS scheduler. For each benchmark, Table 3 reports the total number of operations of the program, runtime of SDS scheduling, as well as the key quality metrics post place-and-route generated by SDC-based scheduling and our SDS scheduler. Table 3 shows that our SDS scheduling approach achieves QoR comparable to that of SDC-based scheduling. On average, we observe small increase in clock period with small reduction in resource usage. Because most of the CHStone benchmarks are not dominated by resource constraints, they do not benefit from reduction in the number of states with the exception of ADPCM, AES, and BLOWFISH. Nevertheless, these experiments demonstrate that the SDS scheduling approach is practical for real-life applications of non-trivial size. We note that the achieved clock period exceeds the target clock period for several benchmarks regardless of the scheduling approach applied. We believe this is a result of inaccurate delay estimation in HLS tools instead of an artifact of our proposed scheduling approach. Table 4 demonstrates that ADPCM, AES, BLOWFISH, and DFMUL can achieve further state reduction after we tighten the resource constraints in LegUp to one memory port and one multiplier.

Table 4: Benchmarks achieving further state reduction after tightening resource constraints — Results are shown for one memory port and one multiplier. Same notations are followed as in Table 3.

Benchmark	SDC-Based Scheduling		SDS Scheduling		
	#States	CP	Runtime	#States	CP
ADPCM	31, 64	12.0	0.04	**26**, **60**	11.3
AES	37, 49, 33, 55	10.4	0.05	37, 49, 33, **47**	10.5
BLOWFISH	118, 57	8.2	0.02	**108**, 57	8.5
DFMUL	7	9.4	0.02	**6**	9.6

6 RELATED WORK AND DISCUSSIONS

Resource-constrained scheduling has been the subject of extensive study, resulting in a line of heuristics, including Hu's Algorithm, List Scheduling, and Force-Directed Scheduling, to solve the problem efficiently. Iterative metaheuristics, such as simulated annealing and ant colony optimization, have also been demonstrated as viable options [24]. Because resource-constrained scheduling maps naturally to a constraint satisfaction problem consisting of logical connectives of linear constraints, it can also be solved with modern SMT solvers, which integrate specialized (linear) solvers with propositional satisfiability search techniques to achieve conflict-driven learning [13]. In particular, a subset of SMT solvers focus on determining the satisfiability of a Boolean combination of difference constraints [35]. These solvers take advantage of an graph-based algorithm to efficiently explore the search space.

Our scheduler is inspired by the concept of SMT and employs a graph-based algorithm to perform conflict-based learning to quickly prune out the infeasible search space. However, unlike generic SMT solvers in which SAT assumes a principal role in driving the underlying theory solver, our solver treats SAT and the underlying theory as equal partners. Notably, our underlying theory is able to influence the subset of SAT clauses that need to be included at each iteration of the feedback loop and determine the appropriate problem that needs to be solved by SAT. In addition, our solver makes heavy use of well-established heuristics specific to the resource-constrained scheduling problem to significantly improve the efficiency of propagation. These problem-specific knowledge provides supports for the key features of our solver, including optimization, resource-aware lower bounding, and incremental learning described in Section 4.

Branch-and-bound style pruning is another popular approach for solving the resource-constrained scheduling problem [5, 25]. This type of approach divides the problem into sub-problems and computes the lower and upper bounds of each sub-problem. A sub-problem is solved optimally when the lower and upper bounds coincide. While these branch-and-bound style schedulers employ problem-specific knowledge from lower and upper bounding to reduce overall scheduling time, our scheduler applies conflict-driven learning tightly coupled with various scheduling heuristics (including upper and lower bounding) to achieve additional runtime improvement. Our approach combines the power of conflict-driven learning and problem-specific knowledge to realize significant speedup. While previous schedulers are designed to work with only resource-constrained scheduling problems, our proposed joint SDC and SAT formulation allows more expressive encoding of a rich set of constraints. With a combination of SAT and SDC, our approach

provides the flexibility to make tradeoffs among different constraints and select the encoding most suitable for each type of constraints.

While this work focuses on HLS, the proposed scheduling approach can equally apply to resource-constrained scheduling problems in many other fields of study. Moreover, our scheduling framework is designed to generalize to a wide range of constrained scheduling problems with a variety of constraints. For example, the framework can be extended to consider constraints arising from various forms of pipeline scheduling [3, 10, 32, 39], which are also typically handled by heuristics for efficiency. In addition, recent interest in dynamically scheduled HLS [11, 18, 20, 21, 33] necessitates a tradeoff between runtime hardware overhead and performance that may not be easily optimized. A scheduling formulation with SAT will enable modeling of the hardware resource overhead so it can be co-optimized during scheduling. Our scheduling approach can also be extended to handle cross-layer HLS optimizations, such as mapping-aware scheduling [31, 40] and place-and-route aware HLS [41], as well as low-power optimizations in HLS [17, 38]. Because many constraints cannot be anticipated by heuristics, the gap to optimality is expected to only widen. Efforts in exact scheduling is therefore crucial for handling a rich set of current and future constraints.

7 CONCLUSIONS

Current HLS scheduling algorithms rely on inexact heuristics that make ad hoc local decisions and cannot accurately and globally optimize over a rich set of constraints. To provide guarantee on QoR out-of-the-box, we propose an exact scheduling approach based on a joint SDC and SAT formulation to precisely handle a variety of scheduling constraints. We develop a specialized scheduler based on conflict-driven learning and problem-specific knowledge to efficiently solve the resource-constrained scheduling problem. By pushing the boundary of what is practically scalable, our scheduler demonstrates orders-of-magnitude improvement in runtime over current exact scheduling approach. Given the flexibility of SAT, we envision that our approach can be effectively applied to a wide range of constrained scheduling problems. As ongoing research, we are further enhancing the proposed scheduler to handle pipeline scheduling and enable more intelligent static optimization techniques for dynamically scheduled HLS.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their insightful comments. This research was supported in part by DARPA Award HR0011-16-C-0037, a DARPA Young Faculty Award, NSF Awards #1337240, #1453378, #1618275, Semiconductor Research Corporation, and a research gift from Xilinx, Inc.

REFERENCES

[1] T. Ajayi et al. Celerity: An Open-Source RISC-V Tiered Accelerator Fabric. *Hot Chips: A Symp. on High Performance Chips*, 2017.
[2] Armin Biere. Lingeling, Plingeling and Treengeling Entering the SAT Competition 2013. *SAT Competition*, 2013.
[3] 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.
[4] A. Canis et al. LegUp: High-Level Synthesis for FPGA-Based Processor/Accelerator Systems. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2011.
[5] Mingsong Chen, Saijie Huang, Geguang Pu, and Prabhat Mishra. Branch-and-Bound Style Resource Constrained Scheduling using Efficient Structure-Aware Pruning. *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2013.
[6] John W. Chinneck and Erik W. Dravnieks. Locating Minimal Infeasible Constraint Sets in Linear Programs. *ORSA Journal on Computing*, 1991.
[7] 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.
[8] Jason Cong and Zhiru Zhang. An Efficient and Versatile Scheduling Algorithm Based on SDC Formulation. *Design Automation Conf. (DAC)*, 2006.
[9] IBM ILOG CPLEX. V12.6: User's Manual for CPLEX. *International Business Machines Corporation*, 2015.
[10] Steve Dai, Mingxing Tan, Kecheng Hao, and Zhiru Zhang. Flushing-Enabled Loop Pipelining for High-Level Synthesis. *Design Automation Conf. (DAC)*, 2014.
[11] Steve Dai, Ritchie Zhao, Gai Liu, Shreesha Srinath, Udit Gupta, Christopher Batten, and Zhiru Zhang. Dynamic Hazard Resolution for Pipelining Irregular Loops in High-Level Synthesis. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2017.
[12] Martin Davis, George Logemann, and Donald Loveland. A Machine Program for Theorem-Proving. *Communications of the ACM*, 1962.
[13] Leonardo De Moura and Nikolaj Bjørner. Satisfiability Modulo Theories: Introduction and Applications. *Communications of the ACM*, 2011.
[14] John Forrest. CBC User Guide. *IBM Research*, 2005.
[15] 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.
[16] Andrei Horbach. A Boolean Satisfiability Approach to the Resource-Constrained Project Scheduling Problem. *Annals of Operations Research*, 2010.
[17] Wei Jiang, Zhiru Zhang, Miodrag Potkonjak, and Jason Cong. Scheduling with Integer Time Budgeting for Low-Power Optimization. *Asia and South Pacific Design Automation Conf. (ASP-DAC)*, 2008.
[18] Lana Josipovic, Philip Brisk, and Paolo Ienne. From C to Elastic Circuits. *Asilomar Conf. on Signals, Systems, and Computers*, 2017.
[19] Yun Liang, Kyle Rupnow, Yinan Li, Dongbo Min, Minh N. Do, and Deming Chen. High-Level Synthesis: Productivity, Performance, and Software Constraints. *Journal of Electrical and Computer Engineering*, 2012.
[20] Gai Liu, Mingxing Tan, Steve Dai, Ritchie Zhao, and Zhiru Zhang. Architecture and Synthesis for Area-Efficient Pipelining of Irregular Loop Nests. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2017.
[21] Junyi Liu, Samuel Bayliss, and George A. Constantinides. Offline Synthesis of Online Dependence Testing: Parametric Loop Pipelining for HLS. *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2015.
[22] Xinheng Liu, Yao Chen, Tan Nguyen, Swathi Gurumani, Kyle Rupnow, and Deming Chen. High Level Synthesis of Complex Applications: An H.264 Video Decoder. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2016.
[23] Sharad Malik and Lintao Zhang. Boolean Satisfiability from Theoretical Hardness to Practical Success. *Communications of the ACM*, 2009.
[24] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1994.
[25] M. Narasimhan and J. Ramanujam. A Fast Approach to Computing Exact Solutions to the Resource-Constrained Scheduling Problem. *ACM Trans. on Design Automation of Electronic Systems (TODAES)*, 2001.
[26] Robert Nieuwenhuis. SAT and SMT are Still Resolution: Questions and Challenges. *Automated Reasoning*, 2012.
[27] Alice C. Parker, Jorge T. Pizarro, and Mitch Mlinar. MAHA: A Program for Datapath Synthesis. *Design Automation Conf. (DAC)*, 1986.
[28] Ganesan Ramalingam, Junehwa Song, Leo Joskowicz, and Raymond E. Miller. Solving Systems of Difference Constraints Incrementally. *Algorithmica*, 1999.
[29] Minjoong Rim and Rajiv Jain. Lower-bound Performance Estimation for the High-Level Synthesis Scheduling Problem. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 1994.
[30] Nitish Kumar Srivastava, Steve Dai, Rajit Manohar, and Zhiru Zhang. Accelerating Face Detection on Programmable SoC Using C-Based Synthesis. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2017.
[31] Mingxing Tan, Steve Dai, Udit Gupta, and Zhiru Zhang. Mapping-Aware Constrained Scheduling for LUT-Based FPGAs. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2015.
[32] Mingxing Tan, Bin Liu, Steve Dai, and Zhiru Zhang. Multithreaded Pipeline Synthesis for Data-Parallel Kernels. *Int'l Conf. on Computer-Aided Design (ICCAD)*, 2014.
[33] Mingxing Tan, Gai Liu, Ritchie Zhao, Steve Dai, and Zhiru Zhang. ElasticFlow: A Complexity-Effective Approach for Pipelining Irregular Loop Nests. *Int'l Conf. on Computer-Aided Design (ICCAD)*, 2015.
[34] J.N.M. Van Loon. Irreducibly Inconsistent Systems of Linear Inequalities. *European Journal of Operational Research*, 1981.
[35] Chao Wang, Franjo Ivančić, Malay Ganai, and Aarti Gupta. Deciding Separation Logic Formulae by SAT and Incremental Negative Cycle Elimination. *Logic for Programming, Artificial Intelligence, and Reasoning*, 2005.
[36] Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. *Int'l Conf. on Computer-Aided Design (ICCAD)*, 2001.
[37] X. Zhang, X. Liu, A. Ramachandran, C. Zhuge, S. Tang, P. Ouyang, Z. Cheng, K. Rupnow, and D. Chen. High-Performance Video Content Recognition with Long-Term Recurrent Convolutional Network for FPGA. *Int'l Conf. on Field Programmable Logic and Applications (FPL)*, 2017.
[38] Zhiru Zhang, Deming Chen, Steve Dai, and Keith Campbell. High-Level Synthesis for Low-Power Design. *IPSJ Transactions on System LSI Design Methodology (T-SLDM)*, 2015.
[39] Zhiru Zhang and Bin Liu. SDC-Based Modulo Scheduling for Pipeline Synthesis. *Int'l Conf. on Computer-Aided Design (ICCAD)*, 2013.
[40] Ritchie Zhao, Mingxing Tan, Steve Dai, and Zhiru Zhang. Area-Efficient Pipelining for FPGA-Targeted High-Level Synthesis. *Design Automation Conf. (DAC)*, 2015.
[41] Hongbin Zheng, Swathi T. Gurumani, Kyle Rupnow, and Deming Chen. Fast and Effective Placement and Routing Directed High-Level Synthesis for FPGAs. *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2014.