# Modulo SDC Scheduling with Recurrence Minimization in High-Level Synthesis

Andrew Canis, Stephen D. Brown, and Jason H. Anderson
ECE Department, University of Toronto, Toronto, ON, Canada
{acanis, brown, janders}@eecg.toronto.edu

***Abstract*—**

**Loop pipelining is a high-level synthesis scheduling technique that overlaps loop iterations to achieve higher performance. However, industrial designs often have resource constraints and other constraints imposed by cross-iteration dependencies. The interaction between multiple constraints can pose a challenge for HLS modulo scheduling algorithms, which, if not handled properly can lead to a loop pipeline schedule that fails to achieve the minimum possible initiation interval. We propose a novel modulo scheduler based on an SDC formulation that includes a backtracking mechanism to properly handle multiple scheduling constraints and still achieve the minimum possible initiation interval. The SDC formulation has the advantage of being a mathematical framework that supports flexible constraints that are useful for more complex loop pipelines. Furthermore, we describe how to specifically apply associative expression transformations during modulo scheduling to restructure recurrences in complex loops to enable better scheduling. We compared our techniques to existing prior work in modulo scheduling in HLS and also compared against a state-of-art commercial tool. Over a suite of benchmarks, we show that our scheduler and proposed optimizations can result in a geomean wall-clock time reduction of 32% versus prior work and 29% versus a commercial tool.**

## I. Introduction

Over the past decades, Field-Programmable Gate Arrays (FPGAs) have continued to march along the path of Moore's law by growing exponentially denser. This growth has made the daunting task of programming these large devices increasingly difficult for hardware engineers. High-Level Synthesis (HLS) is one proposed solution to simplify FPGA design by allowing an engineer to specify an application in a high-level language, such as `C`, that can be synthesized automatically into hardware.

In many `C` applications, the majority of run time is spent executing critical loops. This paper focuses on an important high-level synthesis scheduling technique called *loop pipelining*, which exploits parallelism across loop iterations to generate hardware pipelines. Loop pipelining increases parallelism and hardware utilization, creating circuits similar to hand-coded hardware architectures.

Loop parallelism is limited by cross-iteration dependencies between operations in the loop called *recurrences*. Recurrences can prevent the next iteration of a loop from starting in parallel until data from a prior iteration has been computed, for instance an accumulation across iterations. The second limitation is due to user-imposed resource constraints, e.g. only allowing one floating point adder in the design. These constraints can significantly impact the final loop pipeline throughput.

State-of-the-art HLS scheduling uses a mathematical framework, called a System of Difference Constraints (SDC) to describe constraints related to scheduling [1]. The SDC framework is flexible and allows the specification of a wide range of constraints such as data and control dependencies, relative timing constraints for I/O protocols, and clock period constraints. Although loop pipelining has been well studied in HLS, until recently, the SDC approach had not been applied to scheduling loop pipelines due to non-linearities caused by describing the resource constraints in modulo scheduling. Recent work in [2] has extended the SDC framework to handle loop pipelining scheduling by using step-wise legalization to handle resource constraints. This new SDC approach offers compelling advantages over prior methods of modulo scheduling by providing the same mathematical framework for a wide range scheduling constraints. However, there are issues applying this approach to more complex loops, particularly the class of loops that contain a combination of recurrences and resource constraints.

The focus of this paper is a novel algorithm for modulo loop scheduling using the SDC framework. Our contribution is a new modulo scheduling algorithm that uses backtracking to handle complex loops with competing resource and dependency constraints, as can be expected in commercial hardware designs. This new scheduling approach significantly improves the performance of loop pipelines compared to prior work by scheduling pipelines at their optimal minimum initiation interval even when the loops have complex constraints. Furthermore, our scheduler is based on the SDC formulation allowing for a flexible range of user constraints. We also describe how to apply well-known algebraic transformations to the loop's data dependency graph using operator associativity to reduce the length of recurrences. These associative transformations have already been widely applied for balancing the tree height of expression trees in HLS. However, a loop containing recurrences must be restructured differently to minimize the length of the loop recurrences. This idea has been previously studied in the DSP domain but to the author's knowledge, has not yet been widely applied in HLS.

The remainder of this paper is organized as follows: Section II presents related work. Section III gives an overview of loop pipelining and introduces a motivating example. Section IV describes our modulo SDC scheduling algorithm. Section V discusses our data dependency restructuring transformations to reduce loop recurrence cycles. Section VI presents an experimental study and Section VII draws conclusions.

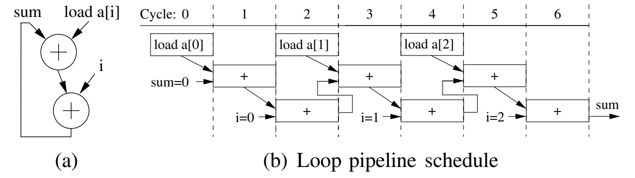
## II. Related Work

In high-level synthesis, scheduling is the task of ordering operations into clock cycles, or control steps, such that all program data dependencies and resource constraints are satisfied. In the presence of resource constraints, scheduling is an NP-hard problem that can be solved with integer linear programming [3] or approximated by various heuristics [4], [5]. The work in [1] has formulated the scheduling problem as a linear programming problem in which constraints are specified in a flexible system of difference constraints (SDC) form. However, the authors did not describe how to support loop pipelining in their formulation.

Loop pipelining can be performed using *software pipelining*, a compiler technique traditionally aimed at Very Long Instruction Word (VLIW) processors [6]. VLIW processors can execute multiple instructions in the same clock cycle allowing them to exploit instruction-level parallelism. Software pipelining uncovers instruction-level parallelism between successive iterations of a loop, and reschedules the instructions to exploit these opportunities. One common software pipelining heuristic is called *Iterative Modulo Scheduling* (IMS) [7], which has been adapted for loop pipelining in high-level synthesis by PICO [8], C-to-Verilog [9], and also by LegUp [10]. Iterative modulo scheduling combines list-scheduling, backtracking, and a modulo reservation table to reorder instructions from multiple loop iterations into a pipelined schedule. IMS, in its original form [7], did not consider HLS operator chaining, as chaining is not applicable to VLIW architectures. The authors of the HLS tool PICO [11] studied the impact of adding chaining capability to IMS, which is non-trivial and requires adding an approximate static timing analysis to the inner loop of the algorithm. However, they focused on area improvements assuming a fixed initiation interval and did not consider the impact of chaining on loop recurrences.

Recently, a heuristic using SDC-based scheduling to perform modulo scheduling was proposed [2]. This work used an SDC-based scheduling formulation with an objective function to minimize register pressure and compared the register usage to swing modulo scheduling [12]. Their scheduling algorithm is similar to the one proposed in this paper but uses a greedy heuristic to choose operations to be scheduled, prioritizing operations that minimize the impact on operations still to be scheduled. We take an alternative approach by abandoning any infeasible partial schedules and then backtracking by attempting other possible scheduling combinations.

The work in [13] presents a method for incrementally reducing the height of an expression tree from O($n$) to O(log$n$) in high-level synthesis. They focus on incrementally reducing the height of the longest path to eventually balance the height of the entire expression tree. However, they did not consider applying the method to recurrences during loop pipelining. Tree height restructuring has also been investigated for software pipelining in the Cydra compiler [14], however they targeted a rotating register ILP processor without considering a flexible high-level synthesis architecture. The work in [15] presents an approach of using algebraic transformations to restructure a data flow graph, while performing retiming, given input and output arrival times for each node in a streaming application. Their work is the most similar to the transformations discussed in this paper, but we focus on the specifics of how to apply these transformations to modulo scheduling in HLS.




(a) (b) Loop pipeline schedule

Fig. 1. Loop pipelining with a recurrence

## III. Motivation

### A. Loop Pipelining Overview

Modulo scheduling rearranges the operations from one iteration of the loop into a schedule that can be repeated at a fixed interval without violating any data dependencies or resource constraints. This fixed interval between starting successive iterations of the loop is called the *Initiation Interval* (II) of the loop pipeline. The best pipeline performance and hardware utilization is achieved with an II of one, meaning that successive iterations of the loop begin every cycle, analogous to a MIPS processor pipeline. If the first iteration of the pipelined loop takes $T$ cycles to complete, then the total number of cycles required to complete the loop is $T + (N - 1) \times II \approx N \times II$, where $N$ is the number of iterations. Consequently, minimizing the initiation interval can significantly improve pipeline performance.

Loop recurrences can increase the initiation interval required for a feasible pipeline schedule. Fig. 1(a) illustrates the data dependency graph of a loop performing an accumulation across iterations: $sum = sum + a[i] + i$, where every addition takes one clock cycle to complete. In this case, $sum$ in the current iteration has a loop-carried dependency on the $sum$ calculated in the previous iteration, therefore the loop contains a recurrence, indicated by the cycle in the data flow graph. Consequentially, after loop pipelining, the schedule shown in Fig. 1(b) shows that each successive loop iteration can begin every two cycles. The recurrence prevents us from the ideal case of scheduling a loop iteration every clock cycle.

In general, we can calculate the minimum recurrence constrained initiation interval (recMII) in the following manner: for every loop recurrence $i$, we take the clock cycle latency along the path, $delay_i$, and divide by the dependency distance of the recurrence, $distance_i$, and round up. The dependency distance is the number of iterations separating the destination operation from the source operation of the recurrence back edge. The recMII is calculated by taking the maximum over all recurrences in the dependency graph: $recMII = \max_i \lceil delay_i / distance_i \rceil$.

Resource constraints can also limit the minimum initiation interval. For instance, if we schedule a loop pipeline with three multiply operations but with only one multiplier unit in the datapath, we must wait three cycles before starting each new loop iteration. In general, we calculate the resource constrained minimum II ($resMII$) by taking every resource type, $i$, and calculating the number of operations in a loop iteration using that resource, $\#ops_i$, divided by the number of functional units available, $\#FU_i$, and round that up to the nearest integer. We take the maximum over all resource types to give us: $resMII = \max_i \lceil \#ops_i / \#FU_i \rceil$. Many resources are typically unconstrained in HLS, for instance adders, in contrast to general purpose processors which have a fixed number of functional units.

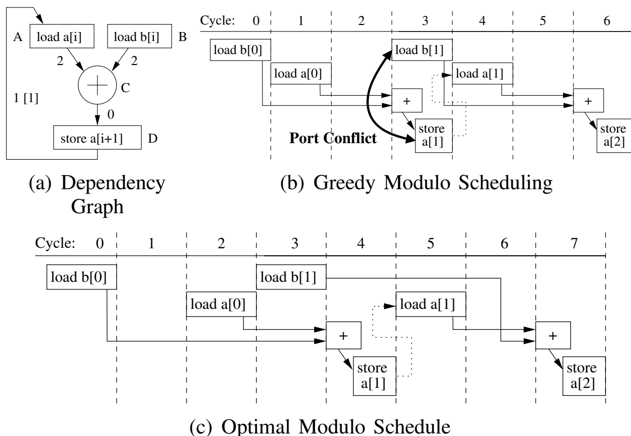
The modulo scheduling algorithm begins by calculating a lower bound on the initiation interval called the minimum II (MII). Any legal schedule must have an II greater than or equal to the MII, but the MII is optimistic and may not be feasible. We calculate the MII by taking the maximum of both the resource constrained MII (resMII) and the recurrence constrained MII (recMII), $MII = max(resMII, recMII)$.

The SDC formulation models the scheduling problem as a Linear Programming (LP) problem with variables modeling the scheduled start and end time of each operation. The form of all SDC constraints are: $s_i - s_j \leq C$, where $s_i$ and $s_j$ are linear programming variables representing operation start or end control steps and $C$ is a constant integer. In this form, the constraints matrix is totally unimodular by construction, which guarantees an integer solution to the linear programming problem. Given a feasible SDC, we can quickly and incrementally determine whether adding a new constraint would make the new system infeasible [16]. This incremental approach works by maintaining a constraint graph, with a vertex for every SDC variable and an edge, $s_j -> s_i$, for every constraint with a weight of $C$. If there are any negative cycles in the constraint graph then the SDC system is infeasible. We can construct data dependency constraints in the SDC formulation to enforce cycle latencies between operations. The linear programming objective function is usually to minimize the start time of each operation to achieve an As-Soon-As-Possible (ASAP) schedule: $min \sum_i s_i$.

### B. Greedy Modulo Scheduling Example

In this section, we will illustrate greedy modulo scheduling for a loop containing both cross-iteration dependencies that cause recurrences in the loop data flow graph and also resource constraints. A greedy modulo scheduling algorithm will not achieve an optimal schedule with the minimum possible initiation interval if we schedule an operation to a particular time step that later turns out to be wrong. Therefore, greedy scheduling is highly dependent on our chosen priority ordering function.

We present the loop data dependency graph given in Fig. 2(a). We have labeled the operations $A$, $B$, $C$, and $D$ for convenience. The directed edges in the graph represent data dependencies between operations and the edge labels indicate the required clock cycle latency between operations.



(a) Dependency Graph
(b) Greedy Modulo Scheduling
(c) Optimal Modulo Schedule

Fig. 2. SDC Modulo Scheduling for II=3

We assume memory latencies of two cycles for a load and one cycle for a store, and we allow the adder to be chained with zero latency. The back edge from node $D$ to node $A$ represents a cross-iteration data dependency with a dependency distance of one (next iteration) labeled in square brackets. The total delay along the recurrence is three cycles, therefore the recMII is three ($\lceil delay/distance \rceil = \lceil 3/1 \rceil = 3$). We assume one memory port giving a resMII of three ($\lceil \#ops/\#FU \rceil = \lceil 3/1 \rceil = 3$).

Modulo scheduling specifies that an operation scheduled at time $t$ will be repeated every $II$ clock cycles. Given resource constraints, we keep track of available resources using a table, where each row tracks a resource and each column is an available time slot. When we schedule an operation at time $t$, we reserve a single time slot in column $t \bmod II$ of the table and in the appropriate resource row. Consequently the table is called the *modulo reservation table* (MRT) and has II time slot columns. Returning to the example, the minimum II is three and the MRT has three time slots available for the memory in Fig. 2(a).

First, we will attempt to greedily modulo schedule the loop. We will schedule operations prioritized in order of *perturbation*, a typical priority function [2], which gives precedence to operations that will most impact the schedule when moved. Therefore, the order of precedence is $B$ (affecting $C$, $D$, and $A$), followed by $A$, and then $D$. First we schedule $B$ into time step 0, and reserve the single memory port for that time. Next, we attempt to schedule $A$ into time step 0 but the memory port is occupied by $B$, so we schedule $A$ into the next time step at time 1. However, after scheduling both loads, when we attempt to schedule the store operation, we will have a memory port conflict, as shown in Fig. 2(b). In this case, the greedy approach fails to achieve the minimum initiation interval of three. There is now no feasible place to schedule the store operation due to the recurrence constraint and the previously scheduled loads. At this point, we must give up and increase the initiation interval to four and try again. However, we can avoid this suboptimal greedy solution by unscheduling one of the load operations and backtracking to find the schedule shown in Fig. 2(c). This schedule is optimal and achieves the minimum initiation interval of three. Generally, a greedy modulo scheduler is only guaranteed to yield an optimal schedule with an II equal to the minimum II if the loop has only (1) simple recurrence circuits involving a single operation, or (2) if each operation in the loop is pipelined with II=1 (no multi-cycle operations), in all other cases greedy scheduling may fail to find the optimal solution [7].

## IV. Modulo SDC Scheduler

In this section, we describe our novel *Modulo SDC Scheduler*. We begin with a candidate II based on the pre-calculated minimum II and increment the II when we fail to find a feasible schedule. Given an II, we can use SDC-based scheduling to quickly give us the control step for every operation in the loop. An advantage we gain from the SDC formulation is the support for operator chaining and frequency constraints. To support modulo scheduling, we modify the SDC constraints that specify dependencies between operations by adding an additional term to account for loop recurrences. For two dependent operations $i \rightarrow j$, the constraint becomes:

$$end_i - start_j \leq II \times distance(i, j) \quad (1)$$

Here $start_j$ is the starting cycle time of operation $j$, and $end_i$ is the cycle time when the output of operation $i$ is available. The dependency distance, which is the number of loop iterations separating the dependency, is given by $distance(i, j)$. If there is no loop-carried dependency then the distance will be zero and this constraint will reduce into a standard SDC data dependency constraint. The loop initiation interval, $II$, is fixed for each iteration of the algorithm. We also add SDC timing constraints between operations to enforce a frequency constraint during scheduling and to prevent excessive chaining from lowering the desired clock period.

Unfortunately, resource constraints during modulo scheduling cannot be modeled using the SDC-based linear programming formulation due to the non-linearity of the modulo reservation table. Therefore, we apply an iterative backtracking approach to legalize the SDC modulo schedule. First, we ignore all resource constraints and then we incrementally assign each resource-constrained operation to a particular control step in the schedule, depending on availability in the modulo reservation table (MRT). In some cases, after fixing one or more resource constrained operations, the schedule will no longer be feasible, in which case we backtrack by unscheduling the tentatively scheduled operations and resuming our attempts.

**Algorithm 1** MODULO SDC SCHED$(II, budget)$

1: Schedule without resource constraints to get $ASAP$ times
2: $schedQueue \leftarrow$ all resource constrained instructions
3: **while** $schedQueue$ not empty **and** $budget \geq 0$ **do**
4: $I \leftarrow$ pop $schedQueue$
5: $time \leftarrow$ scheduled time of $I$ from SDC schedule
6: **if** scheduling $I$ at $time$ has no resource conflicts **then**
7: Add SDC constraint: $t_I = time$
8: Update modulo reservation table and prevSched for I
9: **else**
10: Constrain SDC with GE constraint: $t_I \geq time + 1$
11: Attempt to solve SDC scheduling problem
12: **if** LP solver finds feasible schedule **then**
13: Add $I$ to $schedQueue$
14: **else**
15: Delete new GE constraint
16: BACKTRACKING$(I, time)$
17: Solve the SDC scheduling problem
18: **end if**
19: **end if**
20: $budget \leftarrow budget - 1$
21: **end while**
22: **return** success if $schedQueue$ is empty otherwise fail

Algorithm 1 gives the pseudocode for our iterative algorithm. The input to this function is the initiation interval and a budget, which will be described shortly. First, we schedule the loop without resource constraints and we save the ASAP time for each operation. Next we initialize a queue of all resource constrained operations. We take the first operation out of the queue, which could be a priority queue based on height [7] or perturbation [2] but neither is required due to backtracking. However, having a good priority function will reduce the execution time of the algorithm. Next, we check the MRT for resource conflicts at the time step given by the SDC scheduler. In the first iteration, the SDC time step will be identical to the ASAP time calculated earlier. However, as we add constraints to the SDC formulation, the SDC time steps may begin to diverge from the ASAP times. If there are no MRT resource conflicts then we tentatively assign the operation to that time step by adding an equality constraint to the SDC formulation and we update the MRT and the previous scheduled time for I (lines 7–8). Otherwise, we try to reschedule with that operation constrained to a greater time step (lines 10–11). If we find a feasible schedule then we add this instruction back into the queue for later scheduling (lines 12–13). If we cannot find a feasible schedule (lines 15–17), then we backtrack by unscheduling one or more already scheduled resource constrained instructions and then schedule the current instruction. This process is continued until either a legal schedule is discovered with all resource constrained instructions fixed to a specific time slot or when a budgeted number of while loop iterations have occurred, upon which we consider the current fixed II to be infeasible and increment the II. The budget parameter is equal to the $budgetRatio \times numInstructions$, where we have found empirically that a $budgetRatio = 6$ (as was also found by [7]) works well to avoid excessive backtracking.

**Algorithm 2** BACKTRACKING$(I, time)$

1: **for** $minTime =$ASAP time of $I$ **to** $time$ **do**
2: SDC schedule with $I$ at $minTime$ ignoring resources
3: **break if** LP solver finds feasible schedule
4: **end for**
5: $prevSched \leftarrow$ previous scheduled time for $I$
6: **if** no $prevSched$ **or** $minTime \geq prevSched$ **then**
7: $evictTime \leftarrow minTime$
8: **else**
9: $evictTime \leftarrow prevSched + 1$
10: **end if**
11: **if** resource conflict scheduling $I$ at $evictTime$ **then**
12: $evictInst \leftarrow$ instr. at $evictTime \bmod II$ in MRT
13: Remove all SDC constraints for $evictInst$
14: Remove $evictInst$ from modulo reservation table
15: **end if**
16: **if** dependency conflict scheduling $I$ at $evictTime$ **then**
17: **for all** $S$ in already scheduled instructions **do**
18: Remove all SDC constraints for $S$
19: Remove $S$ from modulo reservation table
20: Add $S$ to $schedQueue$
21: **end for**
22: **end if**
23: Add SDC constraint: $t_I = evictTime$
24: Update modulo reservation table and prevSched for I

Algorithm 2 gives the pseudocode for our backtracking stage, which takes as input an operation $I$ to be scheduled at control step $time$. First, we find a valid time slot while ignoring resource constraints but considering data dependencies (lines 1–4). Because we ignore resource constraints of the partial SDC schedule, we will always find a minimum time slot and break out of the loop on line 3. In lines 5–10, we ensure forward progress by storing the previous scheduled time (updated on line 24 or line 8 of Algorithm 1) of each operation to prevent attempting a time step before that point. This prevents two operations from displacing each other back and forth during backtracking. We remove any resource conflicts at the candidate scheduling time by unscheduling the tentatively scheduled operations found in the MRT at that slot (lines 11–15). In some cases, the previous scheduling time pushes

TABLE I. ALGORITHM EXAMPLE (II=3)

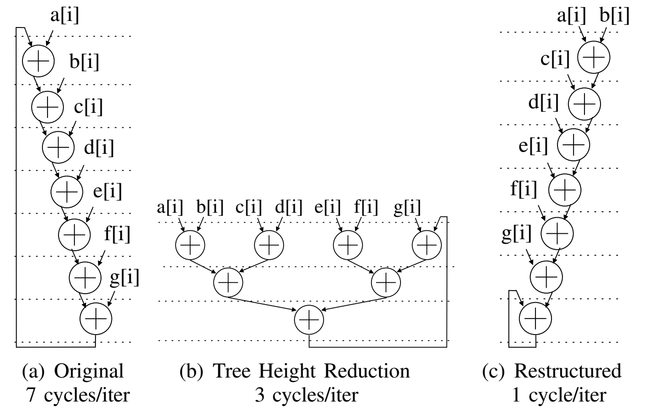
| Iter | MRT Slot | | | SDC Time | | | Sch. Time | | | I | Description |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | B | A | D | B | A | D | | |
| 1 | B | | | 0 | 0 | 2 | 0 | | | B | Sched. $t_B = 0$ |
| 2 | B | | | 0 | 1 | 3 | 0 | | | A | Conflict. $t_A \geq 1$ |
| 3 | B | A | | 0 | 1 | 3 | 0 | 1 | | A | Sched. $t_A = 1$ |
| 4 | D | A | | 0 | 1 | 3 | | 1 | 3 | D | Evict $B$. $t_D = 3$ |
| 5 | D | A | | 1 | 1 | 3 | | 1 | 3 | B | Conflict. $t_B \geq 1$ |
| 6 | D | B | | 1 | 1 | 3 | 1 | | 3 | B | Evict $A$. $t_B = 1$ |
| 7 | | | A | 0 | 2 | 4 | | 2 | | A | Evict All. $t_A = 2$ |
| 8 | B | | A | 0 | 2 | 4 | 0 | 2 | | B | Sched. $t_B = 0$ |
| 9 | B | D | A | 0 | 2 | 4 | 0 | 2 | 4 | D | Sched. $t_D = 4$ |

forward the schedule time of an operation such that there is also a data dependency conflict at the candidate time. In this case, we unschedule all other operations to ensure forward progress and add these operations back into the queue to be rescheduled (lines 16–22). Finally, we schedule the operation at the new time step by updating the MRT and previous scheduled time for I, then we add an equality constraint to the SDC formulation.

### A. Detailed Scheduling Example

In this section, we walk through the exact steps of our scheduling algorithm for the loop data flow graph previously provided in Fig. 2(a). We begin Algorithm 1 by performing SDC scheduling without resource constraints, giving us the ASAP times: $t_A = 0, t_B = 0, t_C = 2, t_D = 2$. Here, we assume $schedQueue$ is prioritized by perturbation [2], giving precedence to operations that will most impact the schedule when moved—although this is not required. The queue contains $B$ (affecting $C$, $D$, and $A$), followed by $A$, and then $D$. We skip $C$ because adders are not resource constrained. Table I provides record keeping for the end of each iteration (first column) of the algorithm. The "MRT slot" column lists the operations reserved in each time slot of the memory MRT, the "SDC Time" column gives the operation control steps under the current SDC constraints, "Sch. Time" gives the tentatively scheduled time of each operation (blank if not scheduled), "I" gives the current instruction $I$, and "Description" summarizes what occurred during the iteration.

In the first iteration, we pop $B$ off the queue and find no resource constraints at time 0, so we add the SDC constraint $t_B = 0$ and reserve MRT slot 0. Next iteration, we try to schedule $A$ but find a resource conflict with $B$, so we update the SDC with $t_A \geq 1$ and re-solve the linear program (LP). Next, we schedule $A$ at time 1 and reserve MRT slot 1. In iteration 4, we try to schedule $D$ at time 3 but MRT slot 0 ($3 \bmod 3 = 0$) is unavailable. We constrain $t_D \geq 4$ and re-solve but the SDC constraints are infeasible due to the recurrence with $A$. At this point a greedy algorithm would give up and increment the II, as shown in Fig. 2(b). Instead, we call BACKTRACKING(D, 3), where we calculate D's $minTime$ to be 3. Therefore, we evict $B$ from the MRT at slot 0, and we can now schedule $D$ at time 3. Next iteration, we find a resource conflict scheduling $B$ at time 0, so we add the constraint $t_B \geq 1$. In iteration 6, we try $B$ at time 1 but there is still a resource conflict, and $t_B \geq 2$ is not feasible due to the recurrence. We call BACKTRACKING(B, 1) and get $minTime = 0$ but $B$ has already been previously scheduled at time 0, so we schedule $B$ at time 1 and kick out $A$ from the MRT. In iteration 7, we have a resource conflict scheduling $A$ at time 1, $t_A \geq 2$ is infeasible, so we call BACKTRACKING(A, 1). A has been previously scheduled at time 1, so we schedule at time 2 which conflicts with the recurrence so we evict all other operations. The algorithm continues as shown in Table I until we find a valid modulo schedule for $II = 3$ with $t_A = 2, t_B = 0, t_D = 4$. At this point the SDC scheduled time for operations without resource constraints is also valid, in this case $t_C = 4$ (the addition).




(a) Original 7 cycles/iter (b) Tree Height Reduction 3 cycles/iter (c) Restructured 1 cycle/iter

Fig. 3. Dependency graph restructuring

## V. LOOP RECURRENCE OPTIMIZATION

Data flow graph transformations have been well-studied in prior work [15], [14], [13]. We propose a targeted manner of applying these transformations specific to HLS modulo scheduling. The goal of these transformations is to reduce the length of loop recurrence cycles in the loop data dependency graph and improve the achievable initiation interval. To illustrate, we will consider a loop that accumulates the sum of seven arrays over all array indices: $sum = sum + a[i] + b[i] + c[i] + d[i] + e[i] + f[i] + g[i]$. Fig. 3(a) shows the default data dependency graph assuming left-to-right associativity. The dotted lines in the figure indicate control steps after scheduling, where arrows that cross the dotted line require registers. For this example, we assume that operator chaining is not allowed, that is, every addition takes one clock cycle to complete. In this case, $sum$ in the current iteration has a loop-carried dependency on the $sum$ calculated in the previous iteration (a dependency distance of one). The loop recurrence spans across seven addition operations, having a path delay of seven clock cycles. Therefore the minimum initiation interval of this loop pipeline is seven cycles ($recMII = \lceil 7/1 \rceil = 7$).

The typical approach in HLS is to balance the expression tree. For instance, we could use the tree height reduction algorithm from [13] to obtain the height balanced tree in Fig. 3(b). We have now reduced the path length of all inputs to three cycles improving the minimum initiation interval to three. While this loop pipeline is more than twice as fast as Fig. 3(a), the minimum initiation interval is still constrained by the loop recurrence.

In our proposed approach, we restructure the expression tree to incur the least latency along the loop recurrence. By targeting loop recurrences, we can focus on improving the minimum initiation interval and consequently the loop pipeline performance. First, we find all operations in the graph that are contained within a loop recurrence. To determine all recurrences in a loop's data dependency graph, we solve the

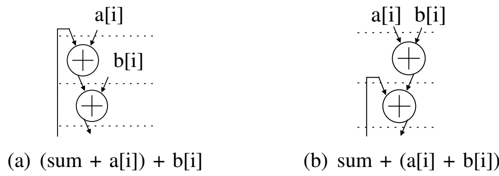



Fig. 4. Incremental Associativity Transformation

equivalent problem of finding all elementary cycles in the graph. An elementary cycle is a path through a graph where the first and last vertices are identical and no other vertex appears twice. All elementary cycles in a graph can be found in polynomial time [17], and each cycle corresponds to a loop recurrence. If the graph contains multiple recurrences, we rank the recurrences by their respective impact on the initiation interval. The rank of each recurrence is found by calculating the recMII of the recurrence in isolation and then ranking the recMII values from high (most critical) to low. Each operation in a recurrence inherits this ranking.

Next, we apply transformations incrementally to the graph to reduce the path length of recurrences. For example, Fig. 4(a) shows the first two addition operations from the original data dependency graph Fig. 3(a), corresponding to the expression: $(sum + a[i]) + b[i]$. The left operand of the first addition is part of the loop recurrence that we wish to improve. We use associativity to restructure these two operations into an algebraically equivalent expression, $sum + (a[i] + b[i])$, as shown in Fig. 4(b). This transformation has reduced the length of the recurrence by one cycle. In general, if we consider additions, an associative transformation involves two two-operand operations that form a recurrence: $late = lateParent + earlyParent$, and $curOp = late + early$. Here $lateParent$ and $late$ are the critical edges along which the recurrence occurs. In this case, we use the associative property of addition to transform this into: $curOp = lateParent + (earlyParent + early)$. In this new expression, we have removed one addition operation from the recurrence, leaving only $lateParent$.

Repeating this associative transformation incrementally, we eventually obtain the restructured data dependency graph in Fig. 3(c). In this new graph, instead of balancing the height of the expression tree, our transformations have actually lengthened some paths in the data dependency graph in order to shorten the recurrence path. The loop recurrence now consists of only one addition, therefore the new loop pipeline has an initiation interval of one (assuming no resource constraints). Due to the improvement in the initiation interval, this new pipeline will have approximately seven times the throughput of the original loop in Fig. 3(a) (II reduced from 7 to 1). Based on this example, we can conclude that the structure of the data dependency graph is critical for obtaining high performance loop pipelines. These transformations are particularly effective for recurrences containing multi-cycle operations, for example floating point operations, which can cause long recurrence lengths and are unaffected by operator chaining.

## VI. Experimental Study

We implemented our new backtracking SDC modulo scheduler in the open-source high-level synthesis tool LegUp [10], which is built within the LLVM compiler framework [18]. LegUp supports loop pipelining of loops where the loop bounds are known before the loop begins execution. The loop body can contain multi-cycle operations such as floating point or memory operations. Cross-iteration dependencies are supported with conservative alias analysis. We also implemented Zhang's greedy modulo SDC scheduler [2] in LegUp for comparison.

We evaluated our approach using five $C$ benchmarks containing a loop with the initiation intervals limited by both loop recurrences and resource constraints, all of which are synthesizable by LegUp and the commercial tool. Table II provides a summary of the properties of each benchmark. All benchmarks contain a tree of operations with a recurrence. The "Balanced Restructuring" column gives the recurrence minimum II (recMII), the resource MII (resMII), and the combined minimum II (MII) for the default case of balanced expression tree restructuring. The "Restructuring" column gives the same metrics but after restructuring, as described in Section V. The "Operations" column gives the number of additions, floating point additions, multiplications, divisions, and memory operations in the loop body. The "Constraints" column gives the constraint on the number of functional units for adders, floating point adders, multipliers, and memory ports, with an X indicating no constraint. Although we restricted memories to two ports in these benchmarks, memory is spread across multiple independent block RAMs that can be accessed in parallel. The "Distance" column gives the dependency distance of the cross-iteration dependency in the loop. The "Total Instr" column gives the total number of LLVM instructions in the loop, most of which represent binary operations, to measure the scheduling complexity of each benchmark. In Table II, we can see that restructuring improved the recMII of *faddtree* from 26 to 13. This was caused when we restructured a floating point addition with a latency of 13 away from the loop recurrence.

The benchmarks include golden input and output test vectors, allowing us to synthesize the circuits with a built-in self-test. We used these test vectors to simulate the circuits in ModelSim and verify correctness. We targeted the Stratix IV [19] FPGA (EP4SGX530KH40C2) on Altera's DE4 board [20] using Quartus II 11.1SP2 to obtain area and FMax metrics. Quartus timing constraints were configured to optimize for the highest achievable clock frequency.

We benchmarked against a state-of-the-art commercial HLS tool configured to target a commercial FPGA similar to Stratix IV. We used the default commercial tool options, which include standard expression tree balancing. We configured LegUp to use functional units with identical latency as the commercial tool. We imposed a target clock period constraint of 3ns (333MHz) on both HLS schedulers.

In our study, we consider four scenarios for comparison: (1) A commercial HLS tool (Comm), (2) Zhang's recently published greedy modulo SDC scheduler [2] (Zha), (3) Our proposed backtracking SDC modulo scheduler (Back), and (4) Our scheduler combined with data dependency graph associative restructuring (Back+R). Table III gives speed performance results for these four scenarios. The "Initiation Interval" column is the scheduled II of the loop pipeline. The "Cycles" column is the total number of cycles required to complete the benchmark. The "FMax" column provides the $FMax$ of the circuit given by the Quartus. The "Time" column gives the circuit wall-clock time: $Cycles \cdot (1/FMax)$. Ratios in the table compare the geometric mean (geomean) of the column

TABLE II. BENCHMARK LOOP PROPERTIES

| | Balanced Restructuring | | | Restructuring | | | Operations | Constraints | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Benchmark** | **recMII** | **resMII** | **MII** | **recMII** | **resMII** | **MII** | **+/fadd/*/%/[]** | **+/fadd/*/%/[]** | **Distance** | **Total Instr** |
| faddtree | 26 | 23 | 26 | 13 | 23 | 23 | 0/21/0/0/22 | X/1/X/X/2 | 1 | 80 |
| adderchain | 2 | 2 | 2 | 2 | 2 | 2 | 40/0/0/0/26 | X/X/X/X/2 | 1 | 92 |
| multipliers | 2 | 2 | 2 | 2 | 2 | 2 | 6/0/2/0/10 | X/X/2/X/2 | 1 | 30 |
| dividers | 2 | 2 | 2 | 2 | 2 | 2 | 11/0/0/4/13 | X/X/X/X/2 | 1 | 72 |
| complex | 3 | 3 | 3 | 3 | 3 | 3 | 16/0/7/2/27 | X/X/3/X/2 | 9 | 98 |

TABLE III. SPEED PERFORMANCE RESULTS

| | Initiation Interval | | | | Cycles | | | | Fmax (Mhz) | | | | Time ($\mu$s) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Benchmark** | **Comm** | **Zha** | **Back** | **Back+R** | **Comm** | **Zha** | **Back** | **Back+R** | **Comm** | **Zha** | **Back** | **Back+R** | **Comm** | **Zha** | **Back** | **Back+R** |
| faddtree | 36 | 34 | 26 | 23 | 1539 | 1439 | 1128 | 1045 | 257 | 229 | 248 | 233 | 5.99 | 6.28 | 4.55 | 4.48 |
| adderchain | 4 | 3 | 2 | 2 | 372 | 297 | 209 | 209 | 270 | 239 | 194 | 234 | 1.38 | 1.24 | 1.08 | 0.89 |
| multipliers | 3 | 3 | 2 | 2 | 292 | 294 | 206 | 207 | 540 | 485 | 545 | 511 | 0.54 | 0.61 | 0.38 | 0.41 |
| dividers | 4 | 3 | 2 | 2 | 261 | 230 | 152 | 152 | 236 | 261 | 270 | 270 | 1.11 | 0.88 | 0.56 | 0.56 |
| complex | 4 | 5 | 3 | 3 | 454 | 550 | 382 | 382 | 269 | 211 | 232 | 232 | 1.69 | 2.61 | 1.65 | 1.65 |
| **Geomean** | **5.9** | **5.4** | **3.6** | **3.5** | **456** | **437** | **309** | **305** | **299** | **271** | **277** | **281** | **1.53** | **1.61** | **1.11** | **1.09** |
| **Ratio** | **1** | **0.92** | **0.62** | **0.6** | **1** | **0.96** | **0.68** | **0.67** | **1** | **0.91** | **0.93** | **0.94** | **1** | **1.05** | **0.73** | **0.71** |

to the respective geomean in the commercial tool.

The results show that our backtracking modulo SDC scheduling approach can have a significant impact on loop pipelines with resource constraints combined with recurrences. Based on our experiments, the commercial tool is using a greedy modulo scheduler for loop pipelining because their schedule cannot achieve the minimum II for these benchmarks. Consequently, our approach achieved a geomean reduction in II by 38% versus the commercial tool. Furthermore, with restructuring we were able to improve this to a 40% reduction in geomean II. We also see that our backtracking approach achieves an average geomean improvement of 33% over Zhang's greedy approach.

Backtracking SDC modulo scheduling reduced the geomean cycle time by 32% versus the commercial tool and by 29% versus Zhang. The cycle time improvement was less than the reduction in II due to time spent outside the loop pipeline in these benchmarks. The geomean FMax decreased by 7% versus the commercial tool when applying our approach, due to better balanced expression restructuring by the commercial tool. When restructuring along recurrences, we chain fewer operations, causing the geomean FMax to increase by 2%. Overall, the geomean wall-clock time for these benchmarks was reduced by 32% using backtracking and restructuring versus greedy SDC modulo scheduling. When compared to the commercial tool, our backtracking and restructuring approach improves geomean wall-clock time by 29%. This improvement is mainly due to a reduction in II caused by better scheduling of our SDC modulo scheduler when compared to greedy scheduling.

Table IV gives the area and runtime results for the four scenarios. The "ALUTs" and "Registers" columns give the number of Stratix IV combinational ALUTs and dedicated registers required. The "Tool Runtime" gives the time in seconds for each algorithm to run, this includes the entire flow from `C` to Verilog. Ratios in the table compare the geometric mean (geomean) of the column to the respective geomean in the commercial tool. Comparing our approach to the commercial tool in terms of area, the geomean combinational ALUTs increased by 9% and the geomean registers increased by 60%. The registers increased due to a lower II in pipelines generated by our approach allowing less register sharing. Comparing our approach to Zhang's in terms of area, the geomean combinational ALUTs decreased by 12% and the geomean registers decreased by 8%.

### A. *Runtime Analysis*

Now we present a runtime characterization of our new backtracking approach versus the other scheduling algorithms. First, the runtime results in Table IV show that our backtracking scheduler had 41% less geomean runtime than the commercial tool but we had a 43% increase in runtime when using restructuring. Our scheduling algorithm's runtime is influenced by the number of invocations of the linear program solver, which is proportional to the number of instructions in the loop being scheduled.

We would like to know the typical range of instructions found in a loop to be pipelined by HLS. The MediaBench II Video benchmark suite [21] is representative of modern and emerging multimedia DSP applications (MPEG-4, JPEG-2000, H.264) with applications that typically have extensive instruction level parallelism. A study of workload characteristics [21] found that the average instructions per basic block in MediaBench was 9.4 instructions, with a maximum of 61 instructions in a single basic block. We performed a characterization of the CHStone [22] benchmarks and found the median instructions per basic block was 4, while the median instructions per loop was 24. Across the CHStone suite, a single basic block contained a maximum of 378 instructions and a maximum of 805 instructions in a single loop. We assume that a loop containing many basic blocks could be merged into one hyperblock using if-conversion [23]. Therefore, we will perform runtime analysis of our algorithm for basic blocks of size up to 1000 instructions.

For this experiment, we used the *adderchain* benchmark and duplicated the body of the loop N times, where N ranged from 1 to 12, and added a final summation after the loop. Each additional duplication introduced another recurrence into the loop pipeline and increased the number of instructions by 79 instructions. Fig. 5 shows the runtime in seconds for each algorithm as the number of instructions in the loop increases. By default, we solved the SDC problem using a linear programming solver [24]. We also analysed the runtime taken when efficiently solving the SDC problem incrementally after modifying the constraints as described in [16]. The lines marked with "(incremental SDC)" show these results. However, we found the runtime difference to be minor, leading us to believe that the LP solver is quite optimized. Here we see that our backtracking algorithm's runtime increases substantially compared to the commercial tool as the number of instructions grow, but the absolute runtime is still about

TABLE IV. AREA AND RUNTIME RESULTS

| | ALUTs | | | | Registers | | | | Tool Runtime (s) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Benchmark** | **Comm** | **Zha** | **Back** | **Back+R** | **Comm** | **Zha** | **Back** | **Back+R** | **Comm** | **Zha** | **Back** | **Back+R** |
| faddtree | 1266 | 1676 | 1629 | 1638 | 2305 | 2175 | 2240 | 2374 | 16 | 34 | 9 | 59 |
| adderchain | 857 | 1190 | 1110 | 1108 | 929 | 1857 | 2178 | 2114 | 6 | 6 | 4 | 10 |
| multipliers | 77 | 122 | 124 | 108 | 68 | 173 | 193 | 110 | 0.2 | 0.4 | 0.2 | 0.2 |
| dividers | 5395 | 8488 | 5495 | 5495 | 9072 | 13771 | 9732 | 9732 | 6 | 3 | 2 | 6 |
| complex | 6551 | 4166 | 4223 | 4223 | 11571 | 14854 | 17732 | 17732 | 7 | 4 | 4 | 5 |
| **Geomean** | **1242** | **1538** | **1391** | **1354** | **1725** | **2698** | **2768** | **2488** | **4** | **4** | **2** | **5** |
| **Ratio** | **1** | **1.24** | **1.12** | **1.09** | **1** | **1.56** | **1.6** | **1.44** | **1** | **1.04** | **0.59** | **1.43** |

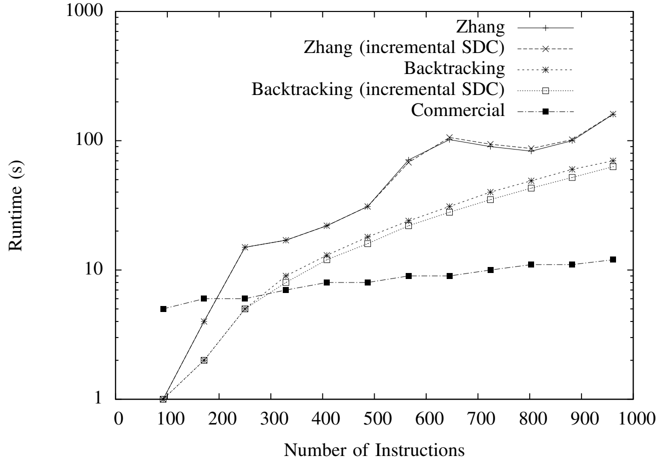



Fig. 5. Runtime Characterization For Loop Pipelining Scheduling Algorithms

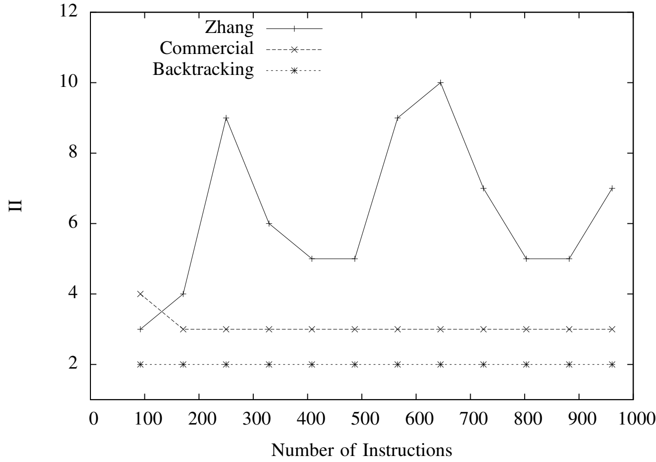



Fig. 6. Initiation Interval for Loop Pipelining Scheduled in Fig. 5

1 minute even for 1000 instructions. Although backtracking runtime compares poorly to the commercial tool, Zhang's greedy approach is actually no better, because if we fail to schedule for a given II we must iteratively increment the candidate II and attempt to reschedule again. This iterative process can be costly in terms of runtime as seen in Fig. 5. Fig. 6 provides the final pipeline II achieved in each case. We observe that the greedy algorithms, both the commercial tool and Zhang, achieve inconsistent pipeline initation intervals due to the resource constraints and cross-iteration dependencies.

## VII. CONCLUSIONS

To summarize, resource constraints and loop recurrences can have a considerable impact on greedy modulo scheduling in HLS by increasing the initiation interval of synthesized pipelines. This paper proposes a backtracking SDC-based modulo scheduling algorithm and a graph restructuring technique for expression height reduction to reduce loop recurrence lengths. Our empirical study on a set of benchmarks containing loop pipelines constrained by resources and limited by recurrences show our approach achieves a 32% improvement in geomean wall-clock versus prior work and 29% versus a commercial tool.

## REFERENCES

[1] J. Cong and Z. Zhang, "An efficient and versatile scheduling algorithm based on SDC formulation," in *ACM/IEEE DAC*, 2006, pp. 433–438.

[2] Z. Zhang and B. Liu, "SDC-based modulo scheduling for pipeline synthesis," in *ICCAD*, San Jose, CA, 2013.

[3] C.-T. Hwang, J.-H. Lee, and Y.-C. Hsu, "A formal approach to the scheduling problem in high level synthesis," *IEEE TCAD*, 1991.

[4] P. Paulin and J. Knight, "Force-directed scheduling for the behavioral synthesis of ASICs," *CAD of Integrated Circuits and Systems*, vol. 8, no. 6, pp. 661–679, 1989.

[5] D. Gajski and et al. Editors, *High-Level Synthesis - Introduction to Chip and System Design*. Kulwer Academic Publishers, 1992.

[6] C. McNairy and D. Soltis, "Itanium 2 processor microarchitecture," *Micro, IEEE*, vol. 23, no. 2, pp. 44–55, 2003.

[7] B. Ramakrishna Rau, "Iterative modulo scheduling," *Int'l Journal of Parallel Processing*, vol. 24, no. 1, Feb 1996.

[8] R. Schreiber, S. Aditya, S. Mahlke, V. Kathail, B. R. Rau, D. Cronquist, and M. Sivaraman, "PICO-NPA: High-level synthesis of nonprogrammable hardware accelerators," *Journal of VLSI signal proc. sys. for signal, image and video tech.*, vol. 31, no. 2, pp. 127–142, 2002.

[9] *C-to-Verilog*, http://www.c-to-verilog.com.

[10] 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," in *ACM/SIGDA Int'l Symp. on FPGAs*, 2011.

[11] M. Sivaraman and S. Aditya, "Cycle-time aware architecture synthesis of custom hardware accelerators," ser. CASES '02, 2002, pp. 35–42.

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

[13] A. Nicolau and R. Potasmann, "Incremental tree height reduction for high level synthesis," in *DAC*, 1991.

[14] M. S. Schlansker and V. Kathail, "Acceleration of first and higher order recurrences on processors with ILP," in *Work. on Lang. & Comp. for Par. Comp.*, 1994.

[15] Z. Iqbal, M. Potkonjak, S. Dey, and A. Parker, "Critical path minimization using retiming and algebraic speed-up," in *DAC*, 1993.

[16] G. Ramalingam, J. Song, L. Joskowicz, and R. E. Miller, "Solving systems of difference constraints incrementally," *Algorithmica*, 1999.

[17] K. A. Hawick and H. A. James, "Enumerating circuits and loops in graphs with self-arcs and multiple-arcs." in *FCS*, 2008, pp. 14–20.

[18] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *IEEE CGO*, 2004, pp. 75–86.

[19] *Stratix-IV Data Sheet*, Altera, Corp., 2010.

[20] *DE4 Dev. and Education Board*, Altera, Corp., 2010.

[21] J. E. Fritts, F. W. Steiling, and J. A. Tucek, "Mediabench II video: Expediting the next generation of video systems research," in *Electronic Imaging 2005*. Int'l Society for Optics and Photonics.

[22] Y. Hara, H. Tomiyama, S. Honda, and H. Takada, "Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis," *Journal of Information Processing*, 2009.

[23] S. A. Mahlke, D. C. Lin, and e. Chen, "Effective compiler support for predicated execution using the hyperblock," in *ACM SIGMICRO*, vol. 23, no. 1-2. IEEE Computer Society Press, 1992, pp. 45–54.

[24] "lp_solve LP solver," *http://lpsolve.sourceforge.net/5.5/*, 2014.