SDC-Based Modulo Scheduling for Pipeline Synthesis

Zhiru Zhang
School of Electrical and Computer Engineering,
Cornell University, Ithaca, NY
zhiruz@cornell.edu

Bin Liu [*]
Micron Technology, Inc., San Jose, CA
binliu@micron.com

Abstract

Modulo scheduling is a popular technique to enable pipelined execution of successive loop iterations for performance improvement. While a variety of modulo scheduling algorithms exist for software pipelining, they are not amenable to many complex design constraints and optimization goals that arise in the hardware synthesis context.

In this paper we describe a modulo scheduling framework based on the formulation of system of difference constraints (SDC). Our framework can systematically model a rich set of performance constraints that are specific to the hardware design. The scheduler also exploits the unique mathematical properties of SDC to carry out efficient global optimization and fast incremental update on the constraint system to minimize the resource usage of the synthesized pipeline. Experiments demonstrate that our proposed technique provides efficient solutions for a set of real-life applications and compares favorably against a widely used lifetime-sensitive modulo scheduling algorithm.

1. Introduction

Recent years have seen an increasingly important role of high-level synthesis in improving both design productivity and quality for nanometer-scale integrated circuits [4, 6]. High-level synthesis is particularly popular in computation-intensive applications (e.g., image and video processing, wireless communication, etc.) where loops abound in the input behavioral descriptions. In such an application context, loop pipelining is often used to meet the stringent performance requirements, since it allows multiple iterations of a loop to operate in parallel by starting an iteration before the previous iteration finishes.

Modulo scheduling [27] is a popular compilation technique to enable loop pipelining. It constructs a static schedule for a loop iteration so that the same schedule can be repeated at a constant interval without causing any resource conflicts or dependence violations. The constant interval between the start of successive iterations is typically termed the *initiation interval* (II). While a variety of modulo scheduling algorithms exist for software pipelining [18], they are not particularly amenable to many hardware-specific design constraints and optimization goals. For example, a software compiler can reasonably assume that most instructions take one or more clock cycles. In hardware synthesis, however, many operations are cheap in delay (e.g., constant shifts, logical operations, and bitwise operations) and contribute little to the cycle time. Even nontrivial arithmetic operations may chain with each other at a low operating frequency. Additional timing constraints such as latency and relative I/O constraints would further complicate the scheduling decisions. Several scheduling algorithms have been proposed to manage different types of timing constraints in high-level synthesis [5, 17, 24]. However, these algorithms primarily focus on sequential non-pipelined architectures.

To address this deficiency, we propose a modulo scheduling framework based on the formulation of system of difference constraints (SDC). Our framework can systematically model a rich set of performance constraints that are specific to the hardware design. The scheduler intelligently exploits the unique mathematical properties of SDC to carry out efficient global optimization and incremental update on the constraint system to minimize resource usage of the synthesized pipeline. More specifically, our contributions are threefold:

(1) We generalize the SDC-based scheduling formulation [5] to support loop pipelining. We show that various forms of pipeline-dependent scheduling constraints can be captured in the SDC system, enabling efficient feasibility checking and global optimization based on linear programming (LP).

(2) We formulate an LP problem over SDC to minimize the overall value lifetimes for reducing registers in the synthesized pipeline. Since the underlying constraint matrix of SDC is totally unimodular, this problem can be solved optimally with guaranteed integral solutions.

(3) We propose a novel incremental scheduling algorithm to minimize pipeline II and register pressure under resource constraints. This algorithm employs a perturbation-based priority function and incrementally updates the SDC constraint system in an efficient stepwise manner to find legal schedules for the resource-constrained operations.

The rest of this paper is structured as follows: Section 2 reviews the previous work on loop pipelining; Section 3 provides background of the SDC formulation and preliminaries for the modulo scheduling problem; Section 4 presents our SDC-based modulo scheduling algorithm; Section 5 reports experimental results, followed by conclusions in Section 6.

[*] Bin Liu was a graduate student at UCLA computer science department when conducting this research.

2. Related Work

Various forms of loop pipelining have been proposed for high-level synthesis in the past. Loop winding [11] focuses on pipelining acyclic data flow graphs without recurrences (i.e., loop feedbacks). The classic force-directed scheduling algorithm [22] can also be extended to support loop winding by folding the resource distribution graph. Sehwa [21] is capable of generating multiple pipelined implementations and exploring the design space with an exhaustive search. Rotation scheduling [2] uses a retiming formulation to iteratively move operations across iterations to create a more compact schedule for the loop body. A recent work extends list scheduling for pipelining and considers binding for more accurate timing estimation [16].

In the compiler domain, loop pipelining is known as software pipelining [18], and is extensively used to achieve a higher order of instruction-level parallelism by moving operations across iteration boundaries. Modulo scheduling [27] is arguably the most popular technique to enable software pipelining since it can achieve high-quality solutions with relatively low overhead in terms of code expansion. Since finding an optimal modulo schedule is NP-hard in general with the presence of both recurrences and resource constraints, various heuristics have been proposed and implemented in research and production compilers. Iterative modulo scheduling [26] schedules and unschedules operations with backtracking to find the minimum possible II under the given resource constraints. Slack modulo scheduling [13] uses a slack-based priority function for operation ordering and employs a bidirectional approach to choosing the time slots so that the conflicting variable lifetimes are minimized. Swing modulo scheduling [20] is able to reduce the register requirements of the resulting schedule by placing each operation close to either its predecessors or successors. A comparative study and quantitative analysis of these modulo scheduling heuristics can be found in [3].

Several recent high-level synthesis systems have adopted the modulo scheduling approach for loop pipelining. For example, PICO-NPA [28] (now part of Synopsys Synphony C compiler) employs iterative modulo scheduling for synthesizing nonprogrammable loop accelerators in hardware. An integer linear programming (ILP) formulation is proposed in [8] to solve a cost-sensitive modulo schedule problem in a loop accelerator synthesis flow derived from PICO-NPA. Another formulation based on satisfiability modulo theory (SMT) is introduced in [9] to optimize loop accelerators with limited degree of programmability. C-to-Verilog [1] performs modulo scheduling to reduce memory port usage under a fixed II constraint.

As previously mentioned, existing modulo scheduling algorithms for software pipelining lack support for many hardware-specific scheduling constraints, such as frequency constraints, latency constraints, etc. General-purpose formulations based on ILP [8] or SMT [9] can capture these constraints but not in scalable forms. Prior techniques also lack efficient global optimization to reduce the register requirements. Instead, they often resort to ad hoc solutions. In this paper we propose an efficient SDC-based modulo scheduling algorithm to address these limitations.

3. Preliminaries and Motivations

In this section we review the SDC-based scheduling formulation and provide preliminaries and motivations for our modulo scheduling problem.

3.1 SDC-Based Scheduling Formulation

SDC-based scheduling algorithm [5] uses a linear programming formulation based on system of difference constraints. Unlike previous ILP approaches which use $O(mn)$ binary variables to encode a scheduling solution with n operations and m steps [14], SDC formulation uses a continuous representation of time with only $O(n)$ variables: for each operation u, a scheduling variable s_u is introduced to represent the time step at which the operation is scheduled. By limiting each constraint to the integer-difference form, i.e.,

$$s_u - s_v \leq d_{u,v} \tag{1}$$

where $d_{u,v}$ is an integer, it is shown that a totally unimodular constraint matrix can be obtained. A totally unimodular matrix (TUM) defined as a matrix whose every square submatrix has a determinant of 0 or ± 1. A linear program with a totally unimodular constraint matrix is guaranteed to have integral solutions. Thus, an optimal integer solution can be obtained without expensive branch-and-bound procedures.

A set of commonly encountered constraints in high-level synthesis can be expressed in the form of integer-difference constraints. For example, data dependences, control dependences, relative timing in I/O protocols, clock frequencies, and latency bounds can all be expressed precisely. Some other constraints, such as resource limitation, do not directly fit into the form. In such cases, approximations can be made to generate pair-wise orderings that can then be expressed as integer-difference constraints. Other complex constraints can be handled in similar ways, using approximations or other heuristics. In Section 4.1 we shall generalize the SDC formulation to support several pipelining constraints.

3.2 Limiting Factors of Pipeline Rate

It is important to note that not every loop is fully pipelinable to achieve $II = 1$, as the pipeline rate can be limited by several factors. Recurrences arise from data feedbacks between loop iterations and often play a major role in limiting the maximum pipeline rate. More precisely, a loop contains a recurrence if an operation in the loop has a direct or indirect dependence on the same operation from a previous iteration.

The resource limitation is another key factor in determining the final II. Other timing constraints such as frequency and latency constraints may impact the II as well. Needless to say, the scheduling heuristic is crucial to handle the interplay between all these constraints to achieve the highest possible pipeline rate (i.e., lowest II).

Figure 1 uses a small example to illustrate the challenges facing a modulo scheduler when both recurrence and resource constraints are involved. The recurrence manifests it-

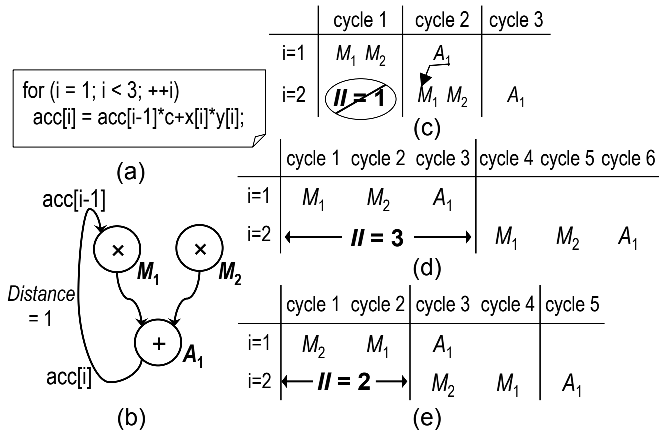

Figure 1. Scheduling with recurrence and resource constraints: (a) C code; (b) Corresponding data flow graph; (c) An illegal schedule violating the inter-iteration dependence; (d) A suboptimal schedule given a single multiplier resulting in $II = 3$; (e) An optimal schedule achieving $II = 2$.

self as a circuit in the data flow graph in Figure 1 (b). This recurrence circuit contains one multiplication M_1 and one addition A_1 which uses and updates the value of $acc[i]$ every iteration, [1] respectively. The forward edge in this circuit captures the *intra-iteration dependence* between A_1 and M_1. There is also a feedback edge between these two operations representing an *inter-iteration dependence*. This feedback edge is associated with a distance value (=1), which indicates the number of iterations separating the two operations involved in this inter-iteration dependence.

Assuming that a multiplier and an adder cannot be chained in one cycle to meet the required frequency, it is infeasible to achieve $II = 1$ without violating the inter-iteration dependence between A_1 and M_1. If we are given an additional resource constraint to only use one multiplier, the modulo scheduler has to carefully prioritize M_1 and M_2 to achieve the optimal $II = 2$ in Figure 1 (e). In fact, conventional heuristics typically place M_1 ahead of M_2 since M_1 belongs to a recurrence circuit. Without backtracking and rescheduling, such decision would lead to a suboptimal solution shown in Figure 1 (d). In Section 4 we will propose a perturbation-based priority function to mitigate this ordering problem.

3.3 Register Pressure

Reducing the register pressure is one of the main optimization objectives of a modern modulo scheduler. In a pipelining context, lifetimes of data values produced in one iteration may overlap with lifetimes of those produced in the subsequent iterations, thereby leading to additional register requirements in hardware. Lifetime-sensitive modulo scheduling techniques have been proposed and employed in production compiler toolchains [12, 13, 20]. Existing heuristics typically place operation nodes either in a top-down topological order where each node is scheduled as soon as possible (ASAP) to be near to its predecessors or in a bottom-up reverse topological order where each node is scheduled as late as possible (ALAP) to be close to its successors. However, such ad hoc ordering schemes may often result in suboptimal schedules in register requirements due to the lack of global optimization. This limitation can be especially pronounced for high-level synthesis when operation chaining is allowed and data values have non-uniform bitwidths.

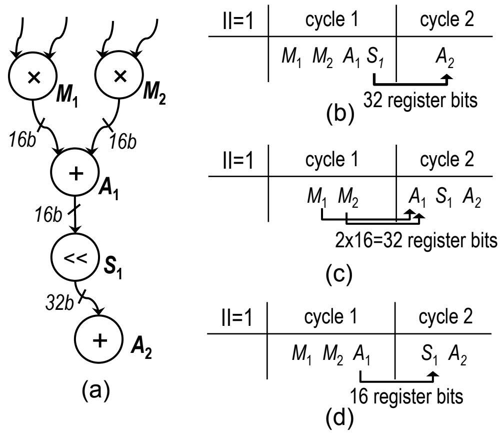

Figure 2. Scheduling strategies for register reduction: (a) Data flow graph; (b) An ASAP schedule resulting in 32 register bits; (c) An ALAP schedule resulting in 32 register bits; (d) An optimal schedule achieving 16 register bits.

Consider the data flow graph in Figure 2 (a) and three different scheduling strategies to reduce the register usage. Here we assume that a constant left shift (S_1) only incurs negligible delay and a 16-bit adder (A_1) can be combinationally chained with a 16-bit multiplier (M_1, M_2) or a 32-bit adder (A_2) without violating the frequency constraint. Figure 2 (b) shows that scheduling operations ASAP in a top-down manner would result in a 32-bit register to store the output of S_1. An ALAP schedule in Figure 2 (c) also requires 32 register bits in total since the results of both M_1 and M_2 need to be saved. The optimal schedule is shown in Figure 2 (d) where only 16 register bits are necessary with A_1 chained with its predecessor and S_1 chained with its successors, respectively.

In Section 4.2 we shall provide an LP-based formulation to minimize the value lifetimes in scheduling to address the aforementioned limitations in prior approaches.

4. SDC-Based Modulo Scheduling

In this section we propose an SDC-based modulo scheduling algorithm which can efficiently handle various scheduling constraints, minimize the initiation interval, and in the mean time, effectively reduce the register usage. The problem we seek to solve is formally stated as follows.

Given: (1) A loop L represented by a cyclic control data flow graph with intra- and inter-iteration dependences; (2) A set of scheduling constraints C which may include resource constraints, cycle time constraints, latency constraints, and relative timing constraints.

[1] We omit array load and store operations in the data flow graph for brevity.

Goal: The scheduler generates a pipelined schedule for L with minimum initiation interval II without violating any given constraints in C, and at the same time, minimizing value lifetimes to reduce register requirements.

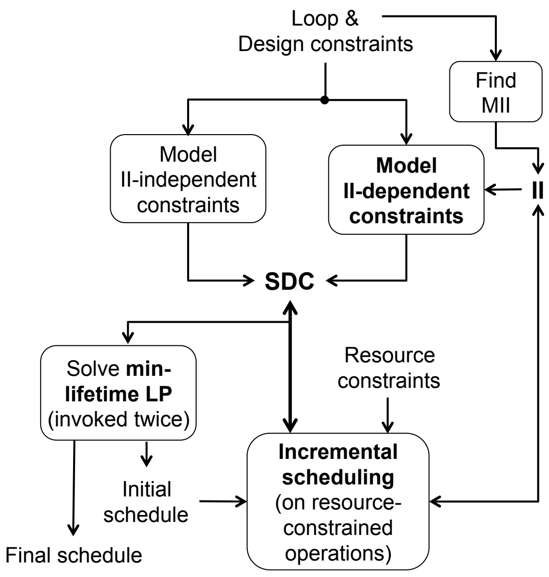
Loop & Design constraints
Find MII
Model II-independent constraints
Model II-dependent constraints
II
SDC
Solve min-lifetime LP (invoked twice)
Resource constraints
Incremental scheduling (on resource-constrained operations)
Initial schedule
Final schedule

Figure 3. SDC-based modulo scheduling framework.

Figure 3 shows the overall framework of our modulo scheduling algorithm. The SDC serves as a central model in this design flow and it consists of the scheduling constraints whose formations are specific to the target II and other constraints that are independent from the pipeline rate. We employ the techniques described in [5] to convert the II-independent scheduling constraints into the SDC model. These constraints typically include intra-iteration dependences and possibly other timing constraints that are specified over the loop body.

To obtain the initial target II, we leverage the techniques described in [26] to compute the lower bound on II, denoted as MII, based on existing recurrences in the loop ($RecMII$) and available resources ($ResMII$). Afterwards, we can convert the II-dependent constraints into SDC as well. Most of these constraints are induced by the recurrences. Section 4.1 will discuss this step in more detail.

Once the base SDC is formed, we solve a minimum-lifetime scheduling problem based on a linear programming formulation (Section 4.2). This step provides the reference schedule for every operation in the loop under the SDC constraints. We then perform an incremental scheduling (Section 4.3) which accounts for the resource constraints and intelligently exploits the unique properties of SDC to perform fast incremental update on the constraint system. In case the incremental scheduling is unable to find a legal schedule, we will increase the target II, update the II-dependent difference constraints in SDC, and reschedule loop kernel in an iterative fashion. After the process finishes, all feasible schedules that satisfy the SDC also satisfy resource constraints; then we can solve the min-lifetime linear program again to obtain the final schedule.

4.1 Modeling Pipelining Constraints

The SDC-based scheduler is able to efficiently convert a number of pipeline-independent scheduling constraints into a set of difference constraints. To account for the pipelining behavior, we generalize the methods described in [5] to model the following pipeline-specific scheduling constraints.

Inter-iteration dependence constraint Given an inter-iteration dependence between operations u and v with a distance $Dist_{u,v}$, we need to guarantee that operation u in iteration k is finished before we start the operation v in iteration $k + Dist_{u,v}$. This can be captured by the following difference constraint.

$$s_u - s_v \le II \times Dist_{u,v} - Lat_u \tag{2}$$

Here Lat_u denotes the latency of operation u, which can be either a combinational (Lat_u = 0) or multi-cycle operation.

Cycle time constraint Loop pipelining exposes possibilities of chaining operations across iterations; namely, the result produced by an operation can be directly consumed in the same clock cycle by another operation that belongs to a later iteration. If such a path exists and involves one or multiple inter-iteration dependences (i.e., back edges), we need to construct the following constraint to ensure that a register is inserted on this path to prevent the unfavorable cross-iteration chaining from causing a frequency violation.

$$\begin{aligned} &T_{cycle} < Delay(cp(u,v)) \le 2 \times T_{cycle} : \\ &s_u - s_v \le II \times Dist_{cp(u,v)} - Lat_u - 1 \\ &where\ Dist_{cp(i_1,i_k)} = \sum_{p=2}^{k} Dist_{i_{p-1},i_p} \\ &with\ u = i_1\ and\ v = i_k \end{aligned} \tag{3}$$

Here we use $cp(u,v)$ to denote a timing-critical path between two operations u and v if the total combinational delay along this path exceeds the target cycle time T_{cycle}. Equation (3) can also be expressed as $s_v + II \times Dist_{cp(u,v)} > s_u + Lat_u$, which enforces that the instance of operation v after $Dist_{cp(u,v)}$ iterations must not chain with operation u of the current iteration to avoid a potential frequency violation.

Relative timing constraints Relative timing constraints can also be specified between the operations in the consecutive iterations. These constraints are particular useful to capture the user-specified I/O protocols which may require that an input port read must happen in a fixed number of cycles after an output port write of the previous iteration.

(i) A minimum timing constraint $MinLat_{u,v}$ between u in iteration k and v in iteration $k + 1$:

$$s_u - s_v \le II - MinLat_{u,v} \tag{4}$$

(ii) A maximum timing constraint $MaxLat_{u,v}$ between u in iteration k and v in iteration $k + 1$:

$$s_v - s_u \le MaxLat_{u,v} - II \tag{5}$$

It is worth noting that we can use a maximum relative timing constraint to model a cross-iteration latency constraint over a group of operations.

4.2 Minimum-Lifetime Schedule over SDC

We provide a linear programming formulation over SDC to minimize the overall register pressure. As indicated in Figure 3, this global optimization problem is solved twice. The first invocation is to provide the ideal positions for all operations before resource constraints are resolved by incremental scheduling. The second time is to generate the final schedule.

For a value produced by operation p (the producer) and used by operations $q_1, q_2, \cdots, q_k$ (consumers of p), the lifetime of the value produced by operation p (later referred to as value p for convenience) starts at the step where p is scheduled, and ends at the step where the last consumer is scheduled. Let l_p be the lifetime of value p, we have

$$\begin{array}{l} s_{q_i} - s_p - l_p \leq d_{q_i,p}, i = 1, 2, \ldots, k. \\ where\ d_{q_i,p} = Lat_p - Dist_{p,q_i} \times II \end{array} \tag{6}$$

Equation (6) can also be expressed as $(s_{q_i} + Dist_{p,q_i} \times II) - (s_p + Lat_p) \leq l_p$, which indicates that the time span between the instance of consumer q_j after $Dist_{p,q_i}$ iterations and the producer p from the current iteration is bounded by the lifetime variable l_p.

We can then combine these lifetime-related constraints with other SDC constraints in the form of Equation (1), and optimize the weighted sum of the value lifetimes using the following linear programming formulation.

$$\begin{array}{lll} \min & \sum_i w_i l_i & \\ \text{s.t.} & s_q - s_p - l_p \leq d_{q,p} & \forall q \text{ using } p \\ & s_u - s_v \leq d_{u,v} & \text{other SDC constraints} \end{array} \tag{7}$$

Here w_i is a constant factor determined by the bitwidth of the corresponding value.

Similar to a proof described in [15] where a time budgeting problem is shown to have a totally unimodular constraint matrix, we can also leverage a theorem from [10] and prove the new constraint matrix with the additional lifetime variables remains a TUM.[2] The TUM property of the constraint matrix guarantees that we can optimally solve the above lifetime minimization problem with integral solutions to directly obtain the ideal schedule for each node.

4.3 Incremental Scheduling

In this section we propose an incremental scheduling algorithm which performs efficient update on the SDC model in a stepwise manner to avoid resource conflicts. The novelties of our approach are threefold:

(1) Unlike previous approaches, this step only attempts to fix schedules of the resource-constrained operations while preserving scheduling freedom for the rest of operations. The final schedule of these "free" operations will be determined by the global linear programming optimization (Section 4.2) to minimize the overall value lifetimes.

[2] We omit the formal proof here due to the page limit.

(2) We determine the schedule of a resource-constrained operation by incrementally adding difference constraints into SDC. Each scheduling decision is fully aware of the existing constraints and efficiently maintains the feasibility of the SDC model (Section 4.3.1).

(3) We compute a unique priority function during incremental scheduling so that the perturbation to the SDC model is minimized when a resource-constrained operation is moved from its ideal position (Section 4.3.2).

4.3.1 Constraint Graph and Incremental Update

An SDC can be conveniently represented by a *constraint graph* [23], in which every scheduling variable becomes a vertex, and every constraint $u - v \leq d$ becomes a d-weighted directed edge from u to v. An SDC is feasible if and only if its corresponding constraint graph is free of negative cycles. Hence we can apply Bellman-Ford algorithm to solve a single-source shortest-path (SSSP) problem on the constraint graph to check the feasibility for the given SDC.

From the feasible solution to the SSSP problem, we can derive the ASAP schedule for each operation by negating the resulting shortest path distance to each vertex on the constraint graph. The ALAP schedule can be obtained in a similar way examining the shortest path distances on the reversed constraint graph, i.e., a graph with the same vertex set as the original graph, but with edge directions reversed.

More importantly, for each newly added (or modified) difference constraint, we can employ an algorithm proposed in [25] to incrementally update the SDC constraint graph. The core idea of [25] is to apply Dijkstra's algorithm with the scaling technique in [7] to update the shortest path values on the subgraph consisting of the vertices impacted by the new constraint. It is able to either compute a new feasible solution or detect a negative cycle. The time complexity is $O(\Delta + \delta log \delta)$, where δ is the number of variables whose feasible values are affected by the new constraint and Δ is the number of constraints involving the affected variables.

Therefore, we can incrementally test/add new scheduling constraints on the SDC constraint graph and its reversed graph to check/maintain the feasibility of the constraint system, and at the same time, efficiently update the ASAP and ALAP schedules of each operation.

4.3.2 Stepwise Legalization

As previously mentioned, the SDC-based scheduling formulation does not attempt to directly model the resource constraints. To avoid resource violation, we start from an initial schedule and legalize the solution by incrementally adding to SDC an additional set of difference constraints on the resource-constrained operation nodes so that the infeasible solutions are gradually pruned away.

The pseudo code of our incremental scheduling algorithm is sketched in Algorithm 1. The inputs to this scheduler include the subject loop L, target II, and the base SDC formed by the II-independent as well as II-dependent scheduling constraints. At the beginning, this step invokes the LP solver to obtain an ideal schedule that minimizes the total value lifetimes. While this min-lifetime schedule may not satisfy all

Algorithm 1: $IncrementalScheduling(L,\ SDC,\ II)$

```
1  use min-lifetime schedule as the reference solution
   /* All schedules remain tentative here. */
2  $step \leftarrow 0$; /* Start from the first step. */
3  while more resource-constrained nodes to schedule do
4      foreach unscheduled resource-constrained node $n$ currently at step, in priority order do
5          for $s$ in $\{step, step-1, \ldots, ASAP(n)\}$ do
6              if scheduling $n$ at $s$ is feasible then
7                  add constraint $s_n = s$ to SDC
8                  update resource scoreboard
9                  break
10         if Backward search fails then
11             Add constraint $s_n \geq step + 1$ to SDC
12             if SDC is infeasible then
13                 report failure and exit
14             put $n$ to $step + 1$ in the reference solution
15     $step \leftarrow step + 1$;
```

resource constraints, it provides a useful initial reference for the scheduler to gradually refine to a legal solution.

More specifically, our incremental scheduling algorithm operates in a stepwise manner. In each time step, we inspect resource-constrained operations that are tentatively placed to this particular step[3] (Line 4) and compute a priority for each of these operations. We then try to commit the operation with the highest priority to the current step (Line 5–8). Here we need to update the SDC model with an equality constraint (i.e., a pair of inequality constraints) to fix the schedule of an operation (Line 7). We also need to update a scoreboard-like data structure that tracks the resource usage of II number of issue slots by marking the corresponding resource on slot $s_n \% II$ as "taken" (Line 8). In the event of a resource conflict with previously committed operation(s), we search backward towards the earliest feasible schedule (Line 5). If such backward search fails, we impose a new constraint to SDC to postpone the operation schedule to a later time step (Line 11). If this new constraint breaks the feasibility of SDC, the incremental scheduling will return failure and the overall modulo scheduling flow will increase the target II.

One of the main rationales behind our stepwise legalization scheme is to minimize the perturbations to SDC and thus preserve as much proximity as possible between the legal solution and the ideal schedule for minimal value lifetimes. To achieve this goal, we use a perturbation-based priority function to rank the operation nodes. Given a time step and a set of candidate operations, we attempt to move each operation away from the given step by tentatively adding an inequality constraint to SDC. By tracing the incremental update process on the SDC constraint graph, we can collect information on how many SDC vertices are impacted due to this change, which is essentially the δ value discussed in Section 4.3.1.

[3] More intuitively, this step is the preferred schedule in order to minimize value lifetimes.

Intuitively, this δ value provides a useful measure of the cost of moving an operation away from its preferred schedule step. If two nodes have equal perturbations, the ALAP/ASAP values are used to break ties in priority ranking. Revisiting the example in Figure 1, we can observe that moving operation M_1 away from $cycle$ 1 will directly impact the schedule of A_1, resulting in a perturbation cost of 1. In contrast, the perturbation cost for M_2 is 2 since moving it to $cycle$ 2 will first impact A_1 which in turn affects the schedule of M_1 to achieve $II = 2$. Hence M_2 would receive a higher priority and stay in $cycle$ 1 and our approach is able to find the optimal schedule shown in Figure 1 (e).

4.3.3 Complexity Analysis

Given a loop $L(V,\ E)$, the number of vertices in the SDC constraint graph (i.e., scheduling variables) n is $O(|V|)$, and the number of edges in the constraint graph (i.e, difference constraints) m is $O(|V|^2)$. Since each resource-constrained operation can be moved at most II times and each constraint update costs $O(m + nlogn)$ in worst case [25], the overall time complexity of the incremental scheduling is $O(II(m+nlogn)n)$. We note that in practice: (1) m is typically linear to $|E|$; (2) resource-constrained operations often represent a small subset of V; and (3) an incremental constraint update usually only affects a small number of variables and constraints.

5. Experimental Evaluation

In this section we report experimental results on a few loop-intensive C/C++ functions, including dataflow-intensive DSP modules which do not contain variable-bound loops and conditional blocks (Table 1), as well as others with more complex control structures extracted from an MPEG-4 decoder (Table 2). The MPEG-4 modules also feature more complex interface accesses to FIFO channels and memories, where limited FIFO/memory ports naturally impose resource constraints. These designs only contain integer operations.

We have implemented the proposed scheduling algorithm in the LLVM compiler infrastructure [19]. Our synthesis flow performs front-end parsing and standard LLVM code optimization together with customized transformation passes for hardware synthesis. Specifically, we perform scalar promotion, bitwidth optimization, and if-conversion to reduce hardware cost and increase instruction-level parallelism. Scheduling is performed after these transformations. The pipeliner implements the proposed algorithm, as well as the swing modulo scheduling [20] for comparison. For code regions that are not pipelined, the SDC-based scheduling algorithm presented in [5] is used.

The swing modulo scheduling algorithm processes operations following the topological order, interleaving top-down and bottom-up passes to schedule operations close to their processors or successors. When cycles are present in the dependence graph due to inter-iteration dependences, it starts from operations in the cycle with the least slack. Swing modulo scheduling is designed to optimize register pressure under resource constraints as well as intra- and inter-iteration

Table 1. Register usage on simple dataflow-intensive loops.

Design	II	Resource	300 MHz			200 MHz			100 MHz		
		$(\times, \pm)$	swing	sdc	ratio	swing	sdc	ratio	swing	sdc	ratio
chem	1	(160, 178)	33358	31534	0.95	24362	23498	0.96	9660	6331	0.66
	4	(40, 45)	13875	14247	1.03	11153	9518	0.85	5011	4115	0.82
	8	(20, 23)	8595	6680	0.78	4661	4533	0.97	4823	2548	0.53
dir	1	(52, 55)	5505	5345	0.97	4046	3982	0.98	1371	891	0.65
	4	(13, 14)	1938	2017	1.04	1295	1135	0.88	588	556	0.95
	8	(7, 7)	1025	958	0.93	1057	814	0.77	605	637	1.05
feig	1	(62, 457)	30530	24680	0.81	22479	15755	0.70	9005	5556	0.62
	4	(16, 115)	12698	9351	0.74	7415	6036	0.81	7792	4403	0.57
	8	(8, 58)	7577	6345	0.84	8809	6377	0.72	8041	4969	0.62
honda	1	(48, 47)	4400	4224	0.96	3328	3200	0.96	1253	540	0.43
	4	(12, 12)	2081	1852	0.89	1439	1279	0.89	894	734	0.82
	8	(6, 6)	1313	1104	0.84	1233	1071	0.87	913	687	0.75
lee	1	(19, 29)	3069	2163	0.70	2826	1826	0.65	1469	588	0.40
	4	(5, 8)	1361	803	0.59	1232	768	0.62	733	492	0.67
	8	(3, 4)	864	738	0.85	766	558	0.73	637	525	0.82
mcm	1	(26, 74)	4996	4756	0.95	3176	3037	0.96	1398	861	0.62
	4	(7, 19)	1591	1351	0.85	1351	1204	0.89	1269	580	0.46
	8	(4, 10)	1303	1047	0.80	1175	996	0.85	1127	613	0.54
pr	1	(16, 26)	1955	1827	0.93	1730	1506	0.87	491	331	0.67
	4	(4, 7)	943	590	0.63	1039	751	0.72	926	429	0.46
	8	(2, 4)	1040	720	0.69	1024	720	0.70	815	592	0.73
wang	1	(18, 26)	2235	1755	0.79	2042	1434	0.70	579	195	0.34
	4	(5, 7)	1305	615	0.47	871	695	0.80	726	324	0.45
	8	(3, 4)	792	503	0.64	437	421	0.96	549	389	0.71
geomean					0.81			0.82			0.62

Table 2. Register usage on loops with more complex control structures and interfaces.

Design	II	Resource	300 MHz			200 MHz			100 MHz		
		$(\times, \pm)$	swing	sdc	ratio	swing	sdc	ratio	swing	sdc	ratio
IDCT_Row	4	(3, 8)	707	675	0.96	601	569	0.95	386	354	0.92
IDCT_Col	8	(2, 3)	1554	1418	0.91	1076	950	0.83	861	731	0.85
MotionCompensate	4	(1, 12)	1091	878	0.80	(II=5) 681	618	0.91	(II=5) 582	516	0.89
TextuerUpdate	4	(1, 4)	1015	991	0.98	860	835	0.97	817	761	0.93
geomean					0.91			0.93			0.90

data dependence constraints; yet it lacks support for other types of constraints, like cycle time and latency constraints. Latency constraints are often necessary in our pipeline synthesis flow, because the loop exit condition needs to be computed within the first II steps of the pipeline, in order to determine whether the next iteration should start when speculative execution is not performed. Thus, we implement swing modulo scheduling on top of the SDC system: every edge in the SDC constraint graph is regarded as a data dependence. Then the latency constraint is handled in the same way as an inter-iteration data dependence, except that the constraint does not involve the lifetime of any value.

We target a generic technology library where the delay of a 32-bit integer ALU (for addition/subtraction) is about $1.8ns$ and that of a 32-bit integer multiplier is about $3.2ns$. Under the same functional unit constraints and target II, the total number of register bits is reported based on the post-binding resource estimation. Table 1 shows the results for simple loops, and Table 2 shows the results for more complex loops. For different target clock frequencies and IIs, we report the ratio of the register usage of our algorithm (sdc) compared with that of the swing modulo scheduling (swing). The results show that our SDC-based pipelining algorithm outperforms swing modulo scheduling in terms of register usage in most scenarios. The reduction in total register bits can be more than 50%, and is particularly significant in cases where high performance is required and abundant functional units are available. In many cases, the advantage is more prominent when clock frequency is lower, because there is more opportunity to optimize operation chaining for register reduction, as illustrated in Figure 2. For modules extracted from the MPEG-4 design, the advantage in register pressure is less significant. This is partly due to the streaming nature of the design, where the ordering of operations is influenced by the order of FIFO/memory accesses. Hence different algorithms tend to generate similar schedules. Nevertheless, we note that our algorithm is able to achieve a smaller II for MotionCompensate at lower frequencies.

6. Conclusions

In this paper we propose an SDC-based modulo scheduling algorithm to support loop pipelining. Our algorithm unifies the normal scheduling constraints and the pipeline-specific constraints into a single SDC. It incrementally searches for the operation schedule to achieve the best possible initiation interval and uses linear programming to minimize the register usage. The experimental results demonstrate that our technique compares favorably against swing modulo scheduling, a widely used heuristic algorithm.

Acknowledgments

This work was supported in part by a research gift from Xilinx Inc. The authors thank Professor Jason Cong at UCLA for his helpful feedback on an initial version of the pipelining algorithm.

References

[1] Y. Ben-Asher, D. Meisler, and N. Rotem. Reducing Memory Constraints in Modulo Scheduling Synthesis for FPGAs. *ACM Trans. on Reconfigurable Technology and Systems*, 3(3), 2010.

[2] L.-F. Chao, A. S. LaPaugh, and E. H.-M. Sha. Rotation Scheduling: a Loop Pipelining Algorithm. *Design Automation Conf.*, pages 566–572, June 1993.

[3] J. Codina, J. Llosa, and A. González. A Comparative Study of Modulo Scheduling Techniques. *Int'l Conf. on Supercomputing*, pages 97–106, June 2002.

[4] 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*, 30(4):473–491, Apr. 2011.

[5] J. Cong and Z. Zhang. An Efficient and Versatile Scheduling Algorithm Based on SDC Formulation. *Design Automation Conf.*, pages 433–438, July 2006.

[6] P. Coussy and A. Morawiec, editors. *High-Level Synthesis: from Algorithm to Digital Circuit*. Springer, 2008.

[7] J. Edmonds and R. M. Karp. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *Journal of the ACM*, 19(2):248–264, 1972.

[8] K. Fan, M. Kudlur, H. Park, and S. Mahlke. Cost Sensitive Modulo Scheduling in a Loop Accelerator Synthesis System. *Int'l Symp. on Microarchitecture*, pages 219–232, Nov. 2005.

[9] K. Fan, H. Park, M. Kudlur, and S. Mahlke. Modulo Scheduling for Highly Customized Datapaths to Increase Hardware Reusability. *Int'l Symp. Code Generation and Optimization*, pages 124–133, Apr. 2008.

[10] A. Ghouila-Houri. Caractérisation des Matrices Totalement Unimodulaires. *CR Acad. Sci. Paris*, 254:1192–1194, 1962.

[11] E. Girczyc. Loop Winding – a Data Flow Approach to Functional Pipelining. *Int'l Symp. Circuits and Systems*, pages 382–385, May 1987.

[12] M. Hagog and A. Zaks. Swing Modulo Scheduling for GCC. *GCC Developers' Summit*, pages 55–64, June 2004.

[13] R. A. Huff. Lifetime-Sensitive Modulo Scheduling. *ACM SIGPLAN Conf. on Programming Languages Design and Implementation*, pages 258–267, June 1993.

[14] C.-T. Hwang, T.-H. Lee, and Y.-C. Hsu. A Formal Approach to the Scheduling Problem in High Level Synthesis. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 10(4):464–475, Apr. 1991.

[15] W. Jiang, Z. Zhang, M. Potkonjak, and J. Cong. Scheduling with Integer Time Budgeting for Low-Power Optimization. *Asia and South Pacific Design Automation Conf.*, pages 22–27, Jan. 2008.

[16] A. Kondratyev, L. Lavagno, M. Meyer, and Y. Watanabe. Realistic Performance-Constrained Pipelining in High-level Synthesis. *Design, Automation & Test in Europe*, pages 1–6, Mar. 2011.

[17] D. C. Ku and G. De Micheli. Relative Scheduling Under Timing Constraints: Algorithms for High-Level Synthesis of Digital Circuits. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 11(6):696–718, June 1992.

[18] M. S. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. *ACM SIGPLAN Conf. on Programming Languages Design and Implementation*, pages 318–328, June 1988.

[19] C. Lattner and V. Adve. LLVM: a Compilation Framework for Lifelong Program Analysis & Transformation. *Int'l Symp. Code Generation and Optimization*, Mar. 2004.

[20] J. Llosa, E. Ayguadé, A. Gonzalez, M. Valero, and J. Eckhardt. Lifetime-Sensitive Modulo Scheduling in a Production Environment. *IEEE Trans. on Computers*, 50(3), Mar. 2001.

[21] N. Park and A. C. Parker. Sehwa: A Software Package for Synthesis of Pipelines from Behavioral Specifications. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 7(3):356–370, Mar. 1988.

[22] P. G. Paulin and J. P. Knight. Force-Directed Scheduling for the Behavioral Synthesis of ASICs. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 8(6):661–678, June 1989.

[23] V. Pratt. Two Easy Theories Whose Combination is Hard. Technical report, Massachusetts Institute of Technology, 1977.

[24] I. Radivojević and F. Brewer. A New Symbolic Technique for Control-Dependent Scheduling. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 15(1):45–57, Jan. 1996.

[25] G. Ramalingam, J. Song, L. Joskowicz, and R. E. Miller. Solving Systems of Difference Constraints Incrementally. *Algorithmica*, 23:261–275, 1999.

[26] B. R. Rau. Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops. *Int'l Symp. on Microarchitecture*, pages 63–74, Nov. 1994.

[27] B. R. Rau and C. D. Glaeser. Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing. *ACM SIGMICRO Newsletter*, 12(4):183–198, 1981.

[28] R. Schreiber, S. Aditya, S. Mahlke, V. Kathail, B. Rau, D. Cronquist, and M. Sivaraman. PICO-NPA: High-level synthesis of nonprogrammable hardware accelerators. *Journal of VLSI Signal Processing*, 31(2):127–142, 2002.