An Efficient and Versatile Scheduling Algorithm Based On SDC Formulation

Jason Cong and Zhiru Zhang
Computer Science Department
University of California, Los Angeles, USA
cong@cs.ucla.edu, zhiruz@cs.ucla.edu

ABSTRACT

Scheduling plays a central role in the behavioral synthesis process, which automatically compiles high-level specifications into optimized hardware implementations. However, most of the existing behavior-level scheduling heuristics either have a limited efficiency in a specific class of applications or lack general support of various design constraints.

In this paper we describe a new scheduler that converts a rich set of scheduling constraints into a system of difference constraints (SDC) and performs a variety of powerful optimizations under a unified mathematical programming framework. In particular, we show that our SDC-based scheduling algorithm can efficiently support resource constraints, frequency constraints, latency constraints, and relative timing constraints, and effectively optimize longest path latency, expected overall latency, and the slack distribution. Experiments demonstrate that our proposed technique provides efficient solutions for a broader range of applications with higher quality of results (in terms of system performance) when compared to the state-of-the-art scheduling heuristics.

Categories and Subject Descriptors: B.8.2 [**Hardware**]

General Terms: Algorithms, Design, Performance

Keywords: Scheduling, Behavioral synthesis, SDC

1. INTRODUCTION

The design complexity of integrated circuit systems in nanometer-scale technologies is outgrowing the capabilities of current RTL-based design methods. This brought about a renewed interest in behavioral synthesis, which promises to automatically transform untimed or partially timed functional specifications into cycle-accurate RTL implementations.

Scheduling, which exploits the parallelism in the behavior-level design and determines the time at which different computations and communications are performed, is commonly recognized as one of the most important problems in behavioral synthesis. However, finding an optimal schedule is intractable in general. Over the years, a large number of scheduling techniques have been proposed in the behavioral synthesis domain, making different tradeoffs between optimality and efficiency. Existing scheduling algorithms can be broadly classified into two major categories: data-flow-based (DF-based) scheduling and control-flow-based (CF-based) scheduling.

DF-based scheduling focuses on data-flow-intensive applications such as digital signal processing and image processing. Based on the optimization goal, they can be further divided into two classes: timing-constrained and resource-constrained. Force-directed scheduling [18] is a widely used constructive heuristic to solve the time-constrained scheduling problem. It minimizes the "force" on the operations to balance computations over the available time steps so that the resource usage can be reduced. For the resource-constrained scheduling problem, the most popular heuristic is list scheduling [17, 10], in which ready operations are sorted in a list according to certain priority function and are scheduled in order into the control state with available resources.

CF-based scheduling targets control-flow-intensive applications such as controllers and network protocol processors. Path-based scheduling [3] is one of the earliest approaches that explicitly deal with control-flow dominated descriptions by scheduling each individual path as fast as possible. Loop-directed scheduling [2] schedules the operations using a depth-first search (DFS). It optimizes the average-case performance and implicitly accounts for the loop repetitions during the DFS. Wavesched [13] explores and schedules the ready operations in a wave-propagation-like manner. It achieves further performance improvement by overlapping the schedules of independent loops and using loop unrolling simultaneously. More recent scheduling algorithms incorporate speculative code motions to extract the parallelisms that are not explicitly exposed in the input descriptions. In [14] the speculative execution is integrated into the Wavesched framework to minimize the expected schedule latency in number of clock cycles. SPARK [7] introduces a set of speculative code transformations into a high-level synthesis framework. A global list-scheduling-based heuristic is used to dynamically select and apply these code motions during the scheduling.

Control-flow dominated descriptions are also characterized by a large share of I/O timing constraints for adhering to the external circuits. Relative scheduling [11] is one of the earliest attempts to handle minimum/maximum timing constraints. Behavioral templates are introduced in [15] to support relative timings by locking a number of operations into certain scheduling templates. VOTAN [16] employs a retiming-based approach to reschedule the timed VHDL by behavioral code transformations but without altering the original I/O timings. I/O timing constraints are also allowed in several exact scheduling approaches, such as the ILP-based scheduling [9, 5], the symbolic scheduling [8, 21], and the constraint-programming-based scheduling [12].

Overall, however, the existing scheduling techniques either have a limited efficiency in a specific class of design applications or lack general support of various design constraints. For instance, DF-based schedulers do not handle control-flow-intensive designs well. Meanwhile, most of CF-based scheduling techniques [3, 2, 13, 8] have an exponential time complexity in the worst case and are not efficient for large designs. Moreover, many of them [3, 2, 13] do not support the relative I/O timings. These deficiencies are particularly unfavorable given the trend wherein behavioral designs are becoming much more complex as they are driven by an escalating growth in algorithm-intensive applications such as 3G wireless, satellite communications and video/image processing. The design descriptions for such applications often feature a combination of intensive computations, controls, and communications, along with various timing/area/power constraints.

In this paper we propose a new scheduling formulation to address the above challenges. Specifically, we convert a rich set of scheduling constraints into a system of difference constraints (SDC). Using this formulation, the consistency of the constraint system can be checked efficiently by solving a single-source shortest path problem. We can also express the performance objective as a linear function so that the global optimization can be performed by solving a linear programming (LP) problem. In addition, the matrix formed by the constraint equations has a special property that guarantees integral solution, which can be directly translated into a valid schedule. Under this unified mathematical framework, we can apply a variety of powerful optimizations to a broad spectrum of design applications.

The remainder of this paper is organized as follows: Section 2 gives the preliminaries and the problem formulation; Section 3 presents our SDC-based scheduling algorithm; Section 4 reports the experimental results followed by the conclusions in Section 5.

2. PROBLEM STATEMENT

This section gives the preliminaries and the problem formulation of our scheduling algorithm.

2.1 Preliminaries

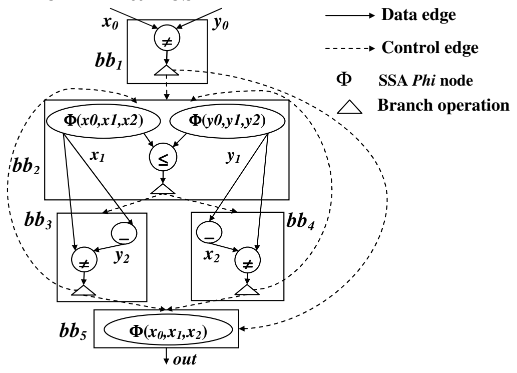

Figure 1: A CDFG example.

The scheduling algorithm is typically performed on the control data flow graph (CDFG). In this work we assume a two-level CDFG model which is defined below.

Definition 1. A **CDFG** is a directed graph $G(V_G,\ E_G)$ where $V_G = V_{bb} \cup V_{op}$ and $E_G = E_c \cup E_d$. V_{bb} is a set of basic blocks (i.e., data flow graphs (**DFGs**)). V_{op} is the entire set of operation nodes in G, and each operation node in V_{op} belongs to exactly one basic block. Data edges in E_d denote the data dependencies between operation nodes. Control edges in E_c represent the control dependencies between the basic blocks. Each control edge $e_c \in E_c$ is associated with a branching condition $bcond(e_c)$.

Figure 1 shows the CDFG for a greatest common divisor (GCD) algorithm, in which the dashed lines represent control dependencies and the solid lines represent data dependencies.

In general, the results of a scheduler can be captured by an FSM-style state transition graph (*STG*) which is described as follows:

Definition 2. An **STG** is a directed graph $G_s(s_0,\ V_s,\ E_s)$. V_s represents a set of **control states** with the initial state s_0. Each control state $s \in V_s$ contains a set of **operations** $OP(s)$ and each operation op is associated with a **guard condition** $gc(op)$ to guard its execution. E_s is the set of **transitions** between the control states and each transition tr is associated with a **transition condition** $tc(tr)$.

2.2 Problem Formulation

The scheduling problem we seek to solve in this paper is formally stated as follows:

Given: (1) A CDFG $G(V_G,\ E_G)$; (2) A set of scheduling constraints C which may include dependency constraints, resource constraints, latency constraints, cycle time constraints, and relative timing constraints.

Goal: The scheduler constructs an STG G_s so that every operation is assigned to at least one state in G_s and all constraints in C are satisfied. In the meantime, the final latency in a particular performance measure is minimized.

3. SDC-BASED SCHEDULING

In this section we present our SDC-based scheduling algorithm to solve the problem formulated in Section 2.2.

3.1 Scheduling Variables

To formally capture the schedule of an operation node in the CDFG, we introduce the concept of *scheduling variables*, which is defined as follows.

Definition 3. Given a CDFG $G(V_{bb} \cup V_{op},\ E_c \cup E_d)$, each node $v \in V_{op}$ is associated with a set of **scheduling variables** $\{sv_i(v) \mid i \in [0,\ Lv]\}$ where $Lv = Latency(v)$

- $\forall v \in V_{op},\ \forall\ i \in [0,\ Lv] :\ sv_i(v) \in \mathbb{N} \cup \{0\}$
- If $Lv \geq 1, \forall v \in V_{op},\ \forall\ i \in [1,\ Lv] :\ sv_i(v) = sv_{i-1}(v) + 1$
- Let $sv_{beg}(v) \equiv sv_0(v)$ and $sv_{end}(v) \equiv sv_{Lv}(v)$

We create one or more scheduling variables for an operation node depending on its pipeline latency.[1] The value of a scheduling variable essentially captures the relative temporal position (in terms of control state) of an operation node (or one pipeline stage of an operation node) in the final schedule. For instance, $s_{beg}(v) = K$ indicates that, in the STG, the longest simple path (without considering the loop-back transitions) from the initial state to the starting state of v has a length of K. Particularly, when the input is a pure DFG, the scheduling variables directly correspond to the absolute node schedules in terms of control steps.

3.2 Modeling Scheduling Constraints

Using the scheduling variables, we can mathematically model the scheduling constraints as a set of *difference constraints*.

Definition 4. An integer **difference constraint** is a formula in the form of $x - y \leq b$ for integer variables x and y, and a constant b.

[1]For the sake of simplicity, we only consider sequential multicycle operations in this paper. However, our algorithm can easily handle combinational multicycle operations.

In the following, we describe how different kinds of scheduling constraints can be expressed in terms of the integer difference constraints (difference constraints, for short). Specifically, three types of scheduling constraints are investigated: dependency constraints, timing constraints, and resource constraints. We model the dependency constraints and timing constraints exactly and model the resource constraints heuristically.

For the sake of convenience, we temporarily remove the loop back edges before constructing the difference constraints. Thus the remaining CDFG becomes an acyclic graph. Nevertheless, these edges will be considered when we generate the objective function and when we construct the final STG. In addition, we polarize each basic block bb by adding two artificial nodes — that is, the super-source $ssrc(bb)$ and the super-sink $ssnk(bb)$.

3.2.1 Dependency Constraints

Dependency constraints are primarily due to data dependencies and control dependencies, which are explicitly exposed by the data edges and control edges in the input CDFG.

(i) **Data dependency constraint**: Data dependencies form the intrinsic scheduling constraints that have to be satisfied to preserve the functionality of the input description. To be more concrete, if there is a data edge from node v_i to node v_j, then v_j cannot be scheduled unless v_i has completed its execution.

$$\forall e(v_i,\ v_j) \in E_d :\ sv_{end}(v_i) - sv_{beg}(v_j) \le 0 \qquad (1)$$

(ii) **Control dependency constraint**: The control dependencies are also honored in this study. Specifically, if there is a control edge from basic block bb_i to basic block bb_j, the operation nodes of bb_j are not allowed to be scheduled before those of bb_i. To formally capture this constraint, we specify the following equation on the super-sink of bb_i and the super-source of bb_j.

$$\forall e_c(bb_i,\ bb_j) \in E_c :\ sv_{end}(ssnk(bb_i)) - sv_{beg}(ssrc(bb_j)) \le 0 \quad (2)$$

3.2.2 Timing Constraints

Many timing constraints for behavioral synthesis can be efficiently transformed to the difference constraints in scheduling variables. Among the most commonly used ones are relative timing constraints, latency constraints, and cycle time constraint.

(i) **Relative timing constraints**: Two types of relative I/O timing constraints are supported in this work.

a. A minimum timing constraint l_{ij} between v_i and v_j ensures that v_j follows v_i by at least l_{ij} number of clock cycles:

$$sv_{beg}(v_i) - sv_{beg}(v_j) \le -l_{ij} \qquad (3)$$

b. A maximum timing constraint u_{ij} between v_i and v_j limits the maximum latency distance between two operations:

$$sv_{beg}(v_j) - sv_{beg}(v_i) \le u_{ij} \qquad (4)$$

Note that combining equations (3) and (4) allows us to specify the exact latency distance between two operations.

(ii) **Latency constraint**: A latency constraint is typically used to specify the maximum acceptable latency over a subgraph of CDFG that has an entry block bb_i and an exit block bb_j; for example, one single basic block, a loop body, a nested if-then-else construct, etc. Latency-constrained blocks should not contain loops.

Suppose that a latency constraint T_{lat} is specified spanning bb_i through bb_j. We then generate the following difference constraint.

$$sv_{end}(ssnk(bb_j)) - sv_{beg}(ssrc(bb_i)) \le T_{lat} \qquad (5)$$

(iii) **Cycle time constraint**: To ensure that the operating frequency of the synthesized RTL implementation meets the target, a cycle time constraint is often used to constrain the maximum combinational delay within a clock cycle.

We define a *combinational path* $cp(v_{i_1},\ v_{i_k})$ between a pair of operation nodes v_{i_1} and v_{i_k} to be a sequence of operation nodes and the data edges that connect these two nodes together, i.e., $cp(v_{i_1},\ v_{i_k}) = \{v_{i_1}, e_{<i_1,i_2>}, v_{i_2}, ..., e_{<i_{k-1},i_k>}, v_{i_k}\}$ where $\forall\ i \in [2,\ k],\ e_{<i_{k-1},\ i_k>} = e(v_{i_{k-1}},\ v_{i_k}) \in E_d$. Particularly, a *critical combinational path* $ccp(v_{i_1},\ v_{i_k})$ between nodes v_{i_1} and v_{i_k} is the combinational path with the largest delay, i.e., $D(ccp(v_{i_1},\ v_{i_k})) = max\{D(cp(v_{i_1},\ v_{i_k}))\}$ where $D(cp(v_{i_1}, v_{i_k})) = \sum_{s=1}^{k} d(v_{i_s}) + \sum_{s=2}^{k} d(e_{<i_{s-1},\ i_s>})$ with $d(v)$ denoting the computation delay and $d(e)$ denoting the communication delay. Note that $d(e(v_i, v_j))$ represents the estimated interconnect delay between the modules that implement operations v_i and v_j.

Suppose that the target cycle time is T_{clk}. For the node pair v_i and v_j with $D(ccp(v_i, v_j)) > T_{clk}$, we construct a difference constraint as follows:

$$sv_{beg}(v_i) - sv_{beg}(v_j) \le -(\lceil D(ccp(v_i, v_j))/T_{clk} \rceil - 1) \qquad (6)$$

Equation (6) states that the combinational path with total delay exceeding the target cycle time T_{clk} must be partitioned into at least $\lceil D(ccp(v_i, v_j))/T_{clk} \rceil$ number of clock cycles.

3.2.3 Resource Constraints

Since resource-constrained scheduling problem is NP-hard in general, in this work we heuristically transform the resource constraints into a set of difference constraints by introducing a set of linear orders.

We define a feasible *linear order* $V^{\pi}_{op}|_{bb_i}$ for each basic block bb_i to be one particular topological order of the underlying DFG of bb_i. Given a linear order $V^{\pi}_{op}|_{bb_i}$, we examine the resources that have limited availabilities. Suppose that the number of functional units available of type res_k is c_{res_k}. For any node pair v^{π}_i and v^{π}_j with $Res(v^{\pi}_i) = Res(v^{\pi}_j) = c_{res_k}$, if there exist $c_{res_k} - 1$ nodes of resource type res_k between v^{π}_i and v^{π}_j in $V^{\pi}_{op}|_{bb_i}$, we impose the following difference constraint on v^{π}_i and v^{π}_j.

$$sv_{beg}(v^{\pi}_i) - sv_{beg}(v^{\pi}_j) \le -Latency(v^{\pi}_i) \qquad (7)$$

Equation (7) enforces a precedence relationship between nodes v^{π}_i and v^{π}_j so that v^{π}_j has to be scheduled in a separate state after v^{π}_i.[2] By adding this type of constraint for every other c_{res_k} nodes of type res_k, we can locally obtain up to c_{res_k} precedence chains among the operation nodes of type res_k. To handle the entire CDFG, we process the basic blocks in a top-down manner using the breadth-first search (BFS) and insert the precedence edges along the execution traces across the basic block boundaries. In the end, we will form up to c_{res_k} precedence chains for any particular execution trace. Since the operations that belong to different execution traces are mutually exclusive, each control state will contain at most c_{res_k} concurrent operations of type res_k. Therefore, the resource constraints will be resolved for a general CDFG.

The linear order generation is an important step in our scheduler to achieve high-performance schedules. In the experimentation for this study, to derive the linear order for each basic block, we sort the operations in ascending order using the As-Late-As-Possible (ALAP) label as the primary key and then use As-Soon-As-Possible (ASAP) label as the tie breaker. This automatically subsumes a static list schedule [10] which has been shown to produce good quality results for DFGs. However, our scheduler does

[2]Note that Equation (7) assumes a non-pipelined resource. To enable resource pipelining, we simply change the equation to $sv_{beg}(v^{\pi}_i) - sv_{beg}(v^{\pi}_j) \le -II$, where II is the initiation interval.

not limit itself to list-scheduling-based orderings. Any algorithm that generates feasible linear orders can be applied as an orthogonal technique to complement our scheduler.

3.3 Solving System of Difference Constraints

The constraint equations described in the preceding sections form a system of difference constraints.

Definition 5. A **system of difference constraints** $SDC(X,\ C)$ consists of a set X of variables and a set C of linear inequalities of the form $x_j - x_i \leq b_k$, where $1 \leq i,\ j \leq n$ and $1 \leq k \leq m$.

This restricted form of linear constraints has a convenient graph representation, called *constraint graph*. This graph can be constructed by representing every variable as a vertex, and constraint $x - y \leq b$ as a b-weighted edge from y to x.

THEOREM 1. *An SDC is consistent (or feasible) if and only if its constraint graph has no negative cycles.*

To detect the presence of negative cycles, we can solve a single-source shortest-path problem on the constraint graph. Using the Bellman-Ford algorithm, the time complexity is $O(mn)$ where n is the number of variables and m is the number of constraints. In addition, an efficient incremental algorithm has been proposed in [19] to determine if the new system is feasible and update its solution in $O(m + nlogn)$ time when a constraint is added or modified.

Moreover, the underlying matrix of an SDC is a *totally unimodular matrix* since every nonsingular square sub-matrix has a determinant of 1 or -1. In the area of linear programming, if the constraint matrix is totally unimodular, the linear programming relaxation generates optimal integer solutions in polynomial time.

3.4 Linear Scheduling Objectives

As discussed in Section 3.3, we convert all the scheduling constraints into a totally unimodular matrix; together with a linear objective function we form a special class of LP problem which guarantees integer solutions. In this section we demonstrate that this technique is especially powerful, as it allows various optimizations by reformulating the objective function.

3.4.1 ASAP and ALAP scheduling

Given a DFG $G(V_{op},\ E_d)$, we can derive the ASAP schedule of G using the following objective function.

$$\min \textstyle\sum_{v \in V_{op}} sv_{beg}(v) \tag{8}$$

Equation (8) states that we can achieve the earliest possible schedules for all the operation nodes if the sum of their starting schedules is minimized. This statement holds because the ASAP schedule represents a unique minimizer of the objective function in an unconstrained scheduling problem.

The ALAP schedule can be generated analogously by optimizing the reverse of the objective function (8).

$$\max \textstyle\sum_{v \in V_{op}} sv_{beg}(v) \tag{9}$$

3.4.2 Optimizing Longest Path Latency

One commonly used performance metric is the *longest path latency*, which refers to the maximum number of clock cycles required to execute a simple path (i.e., a path in which no operation is repeated) from the entry to the exit of the input CDFG.

As mentioned in Section 3.1, the value of the scheduling variable of a node v directly corresponds to the length of the longest simple path from the initial state to the execution state of v in the STG. Therefore, we can minimize the longest path latency by boosting the schedule of the super-sink of the exit basic block ($exit\text{-}bb(G)$) in the input CDFG G.

$$\min\ sv_{end}(ssnk(exit\text{-}bb(G))) \tag{10}$$

3.4.3 Optimizing Expected Overall Latency

Note that the objective function in Equation (10) may lead to an inferior schedule when repetitions are considered. For the schedule in Figure 2(a), suppose that the loop iterates 100 times; we then need to execute both S_1 and S_2 for 100 times. Thus the final latency is 200 clock cycles. Figure 2(b) shows an alternative schedule whose longest path latency is suboptimal ($= 3$). However, the operations of basic blocks bb_2, bb_3, and bb_4 are scheduled into a single state. Therefore, the total latency is only 102 cycles.

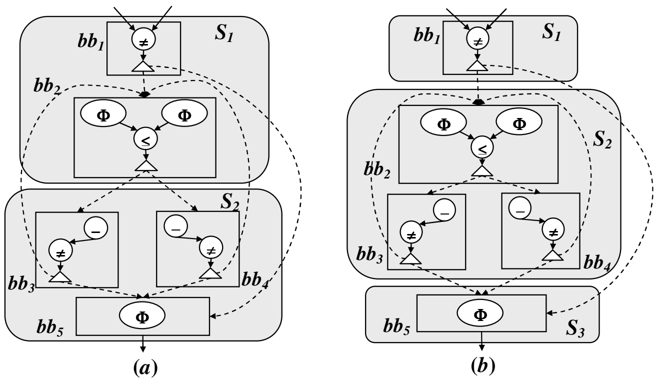

Figure 2: Two alternative schedules for the example CDFG.

The above example clearly demonstrates the need for optimizing the expected overall latency. Since the actual execution traces are typically data-dependent, it is generally difficult to make an accurate static estimation of the final latency of a CDFG. In this case, we try to approximate the expected latency by a linear function of scheduling variables based on the profiling information of the input design. For the same CDFG, suppose that we have a uniform branching probability of 0.5 and the estimated loop iteration count is 100; we can then minimize the following linear function to optimize the expected overall latency. For brevity, we use $x_0(i)$ to denote $sv_{beg}(ssrc(bb_i))$, and $x_1(i)$ to denote $sv_{end}(ssnk(bb_i))$.

$$\begin{aligned} min\ & [x_1(1) - x_0(1)] + 0.5 \times 100 \times [0.5 \times (x_1(3) - x_0(2)) \\ & + 0.5 \times (x_1(4) - x_0(2))] + [x_1(5) - x_0(5)] \end{aligned} \tag{11}$$

In equation (11), the first term and the third term capture the latency on bb_1 and bb_5, respectively; the second term captures the average latency on the loop.

To derive the appropriate objective functions for general CDFGs with complex conditional branches and nested loops, we can employ an iterative approach to traverse the loop hierarchy in a bottom-up manner. At each iteration, we first process the innermost loops and compute the corresponding linear expressions. We then collapse those loops into super-nodes and combine the linear equations on the fly.

3.4.4 Optimizing Slack Distribution

Another possible use of our flexible objective function is to optimize the slack distribution within a schedule. We define slack to be the amount of extra delay an operation (node slack) or a data transfer (edge slack) can tolerate without violating any scheduling constraints. There can be various uses for this extra amount of allowed time. For example, node slack can be exploited by intentionally slowing down the module executing this operation for area/power reduction [6]. Edge slacks are useful in accommodating the potential long interconnect delays to achieve fast timing closure [20].

To maximize the total edge slacks, we can add equation (12) into the objective, in which $indeg(v)$ and $outdeg(v)$ are the in-degree and out-degree of a node v, respectively.

$$\begin{aligned} &max \textstyle\sum_{e(v_i,\, v_j)\in E_d} [sv_{beg}(v_j) - sv_{beg}(v_i)] = \\ &max \textstyle\sum_{v_i \in V_{op}} [(outdeg(v_i) - indeg(v_i)) \times sv_{beg}(v_i)] \end{aligned} \quad (12)$$

To maximize the total node slacks, we can use the node-splitting method suggested by [6] to transform the problem into a maximum weighted edge slack problem.

3.5 Complexity and Optimality Analysis

Given a CDFG $G(V_G,\, E_G) = G(V_{bb} \cup V_{op},\, E_c \cup E_d)$, the number of scheduling variables n is $O(|V_G|)$, and the number of constraints m is $O(|V_G|^2)$. The time complexity of the difference constraint generation is $O(|V_{op}|(|V_{op}| + |E_d|))$. This is primarily due to the cycle time constraint generation process, in which we need to use BFS to compute the critical combinational path delays for up to $|V_{op}|$ times. The LP model formed by the SDC can be solved optimally in polynomial time with integer solutions. In addition, this specific LP problem is dual to the min-cost network flow problem which is solvable in $O(n^2(m + nlogn)logn)$ time.

THEOREM 2. *The SDC formulation for the objectives (8)–(12) can be solved optimally in polynomial time.*

3.6 STG Generation

Given an LP solution, each scheduling variable is assigned an integer value. We then take a three-step approach to translate these values into an actual schedule in STG representation.

Step 1. Construction of control states: First, for each basic block bb, we create a sequence of states $s_l(bb)$, $s_{l+1}(bb)$ ..., $s_u(bb)$ where $l = sv_{beg}(ssrc(bb))$ and $u = sv_{end}(ssnk(bb))$. For any forward control edge $e_c(bb_i,\, bb_j)$ with $sv_{end}(ssnk(bb_i)) = sv_{beg}(ssrc(bb_j))$, the corresponding states $s_u(bb_i)$ and $s_l(bb_j)$ are unified. Next, we deploy each operation v into one or more control states. Specifically, if v belongs to basic block bb, we assign it to states $s_b(bb)$, ..., $s_e(bb)$ where $b = sv_{beg}(v)$ and $e = sv_{end}(v)$.

Step 2. Construction of guard conditions: It is possible that certain operations are conditionally executed within their parent state. We derive the operation guard conditions in two sub-steps:

First, given a control state s with an operation set $OP(s)$, we identify the parent basic block set $PBB(s)$ of $OP(s)$, and locate the nearest common dominator dom for $PBB(s)$ in the original CDFG.

Next, for each basic block $bb \in PBB(s)$, we compute a boolean expression $ac(bb)$ which indicates the necessary condition to activate a control path from dom to bb. Such a condition expression has the general form of $\bigvee_{p \in P(dom \to bb)} \bigwedge_{e_c \in p} bcond(e_c)$, and these conditions can be derived accumulatively by a BFS from dom to the basic blocks in $PBB(s)$. Eventually for each operation op we set its guard condition $gc(op) = ac(bb)$ where bb is op's parent basic block.

Step 3. Construction of state transitions: It is straightforward to build the state transitions by examining the control edges in the CDFG. We can connect the source state to the appropriate target states under different branching conditions.

3.7 Comparison with Other Approaches

Compared to prior approaches, the SDC-based scheduling algorithm showcases the following capabilities:

1. Applicable to a broad spectrum of applications: Our scheduler can generalize the ASAP/ALAP and list scheduling techniques to schedule the data-flow-intensive portions of a behavioral design. In the meantime, global optimizations can be performed over the complex control flows with low computational complexity. Further, both untimed and partially timed descriptions are supported.

2. Amenable to a rich set of scheduling constraints: With the SDC-based formulation, our scheduler honors a variety of scheduling constraints that can be either derived from the input description (e.g., dependency constraints) and target platform (e.g., resource limits), or specified by the users (e.g., frequency constraint).

Note that similar constraint modeling techniques are used in VOTAN [16]. However, they approach the problem from a retiming perspective and only solve the rescheduling problem.

3. Capable of a variety of synthesis optimizations: Our scheduler systematically supports operation chaining when modeling the frequency constraint and allows multicycle resource pipelining when resolving the resource constraints. We can also use relative timing constraints to enable behavioral templates [15]. Moreover, incremental scheduling can be efficiently performed by making local changes to the SDC.

4. Responsive to interconnect delays: Our scheduler can directly handle the layout information and account for the interconnect delays during scheduling.

It is worth noting that the presented scheduling techniques do not perform any speculative code motions. However, we believe that our algorithm can be naturally extended to support this optimization by selectively relaxing the control dependencies.

4. EXPERIMENTAL RESULTS

We implemented the SDC-based scheduling algorithm in our xPilot behavioral synthesis system [4], which can take behavioral C as input and output RT-level VHDL along with the design constraints (e.g., multicycle path constraints) for downstream CAD tools. In this experiment we target the Altera Stratix FPGAs [1], using Quartus II v5.0 for RTL synthesis and physical design.

4.1 Results on Our Benchmarks

We have tested our SDC-based scheduler on five benchmarks with different characteristics. They are profiled in Table 1 by the operation count, basic block count, and loop count. PR and DIT are two discrete cosine transform (DCT) algorithms with pure data flows. DWT implements the discrete wavelet transform algorithm. This benchmark contains a large amount of computations and memory accesses. CACHE is a cache controller which is control intensive and has many cycle-accurate I/O operations. EDGELOOP is extracted from the deblocking filter of an H.264/AVC decoder. It features a mix of computation, controls, and memory accesses.

Table 1: Five benchmarks with different characteristics.

Benchmark	Op#	BB#	Loop#	Resources
PR	59	1	0	5+, 3*
DIT	88	1	0	5+, 3*
DWT	165	11	7	5+, 3*, 2[]
CACHE	166	34	1	5+, 2[]
EDGELOOP	703	124	5	5+, 3*, 2[]

The resource constraints specified for each benchmark are also listed in Table 1, where "+" denotes an ALU that does addition and subtraction, "$*$" is a multiplier, and "[ ]" denotes a memory read/write port.

We optimize longest path latency (10) as the primary objective and linearly combine maximum edge slack as the secondary objective (12). We set the target cycle time to be 10*ns* for *EDGELOOP* which is the the most complex design. For the rest of the designs, we use 5*ns* as the target clock period. Typical runtimes of our scheduler on these designs are within two seconds on a 2.4GHz Pentium 4 Linux PC. The synthesis results are shown in Table 2. The total number of control states generated by the SDC-based scheduling algorithm is shown in the second column of Table 2. We also list the estimated worst-case cycle counts in the third column for the scheduled designs. The final frequency (in MHz) and the resource usage (including logic element count and DSP9x9 block count) results are shown in the last three columns.

Table 2: SDC-based scheduling results on our benchmarks.

Benchmark	State#	Cycle#	Fmax	LE#	DSP#
PR	8	8	159.5	508	32
DIT	10	10	174.2	592	28
DWT	51	6926	183.6	1926	128
CACHE	47	*	161.6	371	0
EDGELOOP	107	*	100.1	7440	80
* Design contains loops with data-dependent iteration counts					

To validate the quality of results of our scheduler, we make comparisons with SPARK [7] on the same benchmark suite. SPARK is a state-of-the-art academic behavioral synthesis system which also synthesizes C descriptions into VHDLs. As mentioned in Section 1, SPARK features a list-scheduling-based heuristic with the capability of performing global speculative code motions. However, SPARK does not support relative timing constraints, and it only handles the loops with fixed constant iteration counts.

Table 3: SPARK results on our benchmarks.

Benchmark	State#	Cycle#	Fmax	LE#	DSP#
PR	8	8	134.3	462	32
DIT	10	10	130.7	724	28
DWT	53	7884	–	–	–

The synthesis results of SPARK are shown in Table 3. Under the same frequency and resource constraints, SPARK produces similar cycle count and area results on two DCT benchmarks. However, their final frequencies are inferior to ours by 25% since we can efficiently distribute the edge slacks to accommodate the interconnect delays. For DWT design, we are unable to obtain the final results on FPGA for SPARK due to its inefficient memory handling.[3] Further, SPARK fails on CACHE and EDGELOOP due to its lack of support for relative timing constraints and general loop constructs.

4.2 Results on SPARK's Benchmarks

Table 4: Four SPARK's benchmarks.

Benchmark	Op#	BB#	Loop#	Resources
ADPCM-dec	59	31	1	2+, 1*, 2<<, 2<, 2[]
ADPCM-enc	59	41	1	2+, 1*, 2<<, 2<, 2[]
GIMP-tiler	150	34	2	3+, 1*, 1/, 2<<, 2<, 2[]
MPEG-dpframe	236	37	4	4+, 1*, 2<<, 2<, 2[]

We make further comparisons with SPARK using another four designs from SPARK's benchmark suite. These designs are described in Table 4, where "<" is a comparator and "<<" is a shifter.

Since all of these four benchmarks contain memory (array) accesses, we only compare the worst-case cycle counts of the scheduled designs. The results are shown in Table 5. We set a 10ns target cycle time for both schedulers. Although currently our scheduler does not perform any speculative code motions, we can still efficiently schedule the multiple simple basic blocks (e.g., an if-then-else construct) into a single state and perform operation chaining to reduce the cycle count without violating the frequency constraint. On average, our scheduler is 16% better than SPARK in terms of the estimated worst-case total number of clock cycles.

[3]SPARK currently flattens all the arrays into primary input/output pins. This makes the synthesized RTL infeasible for implementation on the target FPGA device with limited I/O resources.

Table 5: Our scheduler vs. SPARK: Cycle count comparison.

Benchmark	SPARK		SDC		SDC / SPARK
	State#	Cycle#	State#	Cycle#	
ADPCM-dec	15	327	13	278	0.85
ADPCM-enc	16	133	13	112	0.84
GIMP-tiler	27	2234	32	1877	0.84
MPEG-dpframe	32	424	35	352	0.83

5. CONCLUSIONS AND ONGOING WORK

This paper presents an SDC-based scheduling algorithm which provides efficient solutions for a wide range of application domains and honors a rich set of design constraints. The experimentation demonstrates that the proposed technique can effectively optimize system performance. We are currently enhancing our scheduler to incorporate speculative execution and loop pipelining.

Acknowledgments

This research is partially supported by the Semiconductor Research Corporation, the Gigascale Silicon Research Center, and the National Science Foundation.

6. REFERENCES

[1] Altera Website. http://www.altera.com.

[2] S. Bhattacharya, S. Dey, and F. Brglez. Performance Analysis and Optimization of Schedules for Conditional and Loop-Intensive Specifications. In *Proc. Design Automation Conf.*, pages 491–496, June 1994.

[3] R. Camposano. Path-Based Scheduling for Synthesis. *IEEE TCAD*, 10(1):85–93, January 1991.

[4] D. Chen, J. Cong, Y. Fan, G. Han, W. Jiang, and Z. Zhang. xPilot: A Platform-Based Behavioral Synthesis System. In *Proc. SRC Techcon Conf.*, October 2005.

[5] C. H. Gebotys and M. I. Elmasry. Global Optimization Approach for Architectural Synthesis. *IEEE TCAD*, 12(9):1266–1278, September 1993.

[6] S. Ghiasi, E. Bozorgzadeh, S. Choudhury, and M. Sarrafzadeh. A Unified Theory of Timing Budget Management. In *Proc. Int. Conf. Computer Aided Design*, pages 653–659, November 2004.

[7] S. Gupta, N. Savoiu, N. Dutt, R. Gupta, and A. Nicolau. Using Global Code Motions to Improve the Quality of Results for High-Level Synthesis. *IEEE TCAD*, 23(2):302–311, February 2004.

[8] S. Haynal. *Automata-Based Symbolic Scheduling*. PhD thesis, University of California, Santa Barbara, December 2000.

[9] C.-T. Hwang, T.-H. Lee, and Y.-C. Hsu. A Formal Approach to the Scheduling Problem in High Level Synthesis. *IEEE TCAD*, 10(4):464–475, April 1991.

[10] R. Jain, A. Mujumdar, A. Sharma, and H. Wang. Empirical Evaluation of Some High-Level Synthesis Scheduling Heuristics. In *Proc. Design Automation Conf.*, pages 686–689, June 1991.

[11] D. C. Ku and G. De Micheli. Relative Scheduling Under Timing Constraints. In *Proc. Design Automation Conf.*, pages 59–64, June 1992.

[12] K. Kuchcinski. Constraints-Driven Scheduling and Resource Assignment. *ACM TODAES*, 8(3):355–383, January 2003.

[13] G. Lakshminarayana, A. Raghunathan, and N. K. Jha. Wavesched: A Novel Scheduling Technique for Control-Flow Intensive Designs. *IEEE TCAD*, 18(5):505–523, May 1999.

[14] G. Lakshminarayana, A. Raghunathan, and N. K. Jha. Incorporating Speculative Execution into Scheduling of Control-Flow-Intensive Designs. *IEEE TCAD*, 19(3):308–324, March 2000.

[15] T. Ly, D. Knapp, R. Miller, and D. MacMillen. Scheduling using Behavioral Templates. In *Proc. Design Automation Conf.*, pages 101–106, June 1995.

[16] M. Münch, N. Wehn, and M. Glesner. An Efficient ILP-Based Scheduling Algorithm for Control-Dominated VHDL Descriptions. *ACM TODAES*, 2(4):344–364, October 1997.

[17] A. Parker, J. Pizarro, and M. Mlinar. MAHA: A Program for Datapath Synthesis. In *Proc. Design Automation Conf.*, pages 461–466, June 1986.

[18] P. Paulin and J. Knight. Force-Directed Scheduling for the Behavioral Synthesis of ASICs. *IEEE TCAD*, 8(6):661–678, June 1989.

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

[20] L. Singhal and E. Bozorgzadeh. Fast Timing Closure Through Interconnect Criticality Driven Delay Relaxation. In *Proc. Int. Conf. Computer Aided Design*, pages 792–797, November 2005.

[21] A. Vijayakumar and F. Brewer. Weighted Control Scheduling. In *Proc. Int. Conf. Computer Aided Design*, pages 777–783, November 2005.