Scheduling with Soft Constraints

Jason Cong[†*] cong@cs.ucla.edu
Bin Liu[†‡] bliu@cs.ucla.edu
Zhiru Zhang[‡] zhiruz@autoesl.com

[†]Computer Science Department, University of California, Los Angeles
[‡]AutoESL Design Technologies, Inc.

ABSTRACT

In a behavioral synthesis system, a typical approach used to guide the scheduler is to impose hard constraints on the relative timing between operations considering performance, area, power, etc., so that the resulting RTL design is favorable in these aspects. The mechanism is often flawed in practice because many such constraints are actually *soft constraints* which are not necessary, and the constraint system may become inconsistent when many hard constraints are added for different purposes. This paper describes a scheduler that distinguishes soft constraints from hard constraints when exploring the design space. We propose a special class of soft constraints called *integer-difference soft constraints*, which lead to a totally unimodular constraint matrix in an integer linear programming formulation. By exploiting the total unimodularity, the problem can be solved optimally and efficiently using a linear programming relaxation without expensive branch and bound procedures. We also show how the proposed method can be used to support a variety of design considerations. As an example application, we apply the method to the problem of low-power synthesis with operation gating. In a set of experiments on real-world designs, our method achieves an average of 33.9% reduction in total power; it outperforms a previous method by 17.1% on average and gives close-to-optimal solutions on several designs.

Categories and Subject Descriptors

B.7.2 [**Integrated Circuits**]: Design Aides

General Terms

Algorithms, Design

Keywords

Behavioral synthesis, low power, soft constraint, scheduling

1. INTRODUCTION

Due to increasing design complexity and time-to-market pressure, interest in behavioral synthesis has recently been revived. Scheduling, i.e., the process of transforming a behavioral description into a cycle-accurate RTL model, is recognized as a key step in behavioral synthesis [9, 20]. The design space of scheduling is enormous, and scheduling results have a significant impact on almost all the important design metrics, including performance, power, area, reliability, etc. Not surprisingly, a large number of interesting algorithms have been proposed in the literature to solve various scheduling problems in behavioral synthesis. These algorithms typically focus on intelligent strategies to automate the optimization of one or more objective functions under a set of design constraints.

One important aspect in scheduling is the way that the scheduler is guided towards a good solution. In a straightforward approach, the optimization engine iteratively evaluates a number of (partial) solutions using various estimators of QoR (quality of results, including frequency, throughput, area, power, etc.), and hopefully finds a satisfactory solution. This approach often makes it very tricky to design a good searching strategy when multiple aspects of the design are considered simultaneously. Enumerative and randomized searching methods can often achieve good QoR on small designs, but they are often too expensive computationally for moderately large designs. In practice, local heuristic methods like list scheduling [17, 18], force-directed scheduling [24] and their variants are often used. These heuristics require a translation from complex models to priorities (in list scheduling) or forces (in force-directed scheduling). The translation is most likely performed in an ad hoc manner, because (1) multiple design requirements need to be considered simultaneously, (2) practical models of QoR are often too complex to be expressed using simple priorities or forces. As a remedy, some systems add constraints about the relative timing between operations to guide the scheduler. For example, the chaining of operations can be determined before scheduling [19], and artificial dependency constraints between operations can be added during scheduling [3, 8, 22, 30]. These constraints are usually introduced when they are considered favorable in a certain aspect of QoR (like reducing area or saving power), and they can effectively prune inferior solutions when used intelligently. This mechanism seems natural, but it often leads to the following problems.

1. Design space is not well characterized. While these constraints eliminate inferior solutions from the search space, they can also forbid the exploration of some feasible and possibly good solutions due to low accuracy of estimation models at such a high level. For example, if the estimated propagation delay of a combinational path consisting of two functional units is 10.5 ns during scheduling, while the required cycle time is 10 ns, a simple method would forbid the two operations to execute in one clock cycle. However, it is probable that a solution with a slight nominal

[*]Dr. Cong also serves as the Chief Technology Advisor for AutoESL Desgin Technologies, Inc.

ICCAD'09 November 2–5, 2009, San Jose, California, USA.

timing violation can still meet the frequency requirement after various timing optimization procedures in later design stages, like logic refactoring, retiming, threshold voltage assignment, gate/wire sizing, placement/routing, etc. In this case, artificial constraints eliminate the possibility of improving other aspects of the design with some reasonable estimated violations.

2. Inconsistency can occur in the constraint system. When a lot of constraints are added for different purposes, they will probably contradict with each other. For example, scheduling two operations in a certain order will lead to lower power dissipation, but the order may conflict with other requirements like performance in less obvious ways (will be discussed in Section 3). When inconsistency happens, many scheduling algorithms will either fail or discard some constraints completely.

Both of the above deficiencies come from the rigid way in which constraints are handled in the scheduler. So, we take a different approach. We notice that many of these constraints are actually preferences (also referred to as *soft constraints*) rather than essential constraints (referred to as *hard constraints*). Unlike hard constraints, soft constraints are supposed to be followed when possible but not necessarily. Thus, soft constraints will neither limit the design space nor cause inconsistencies even if conflicting soft constraints are present.

The conceptual structure of a scheduler with support for soft constraints is illustrated in Figure 1. The scheduler accepts a functional specification and a number of QoR targets along with characterizations of the target platform. A number of hard/soft constraint generators produce hard/soft constraints based on the specifications. Hard constraint generators usually impose essential requirements of the design, like dependency between operations or performance requirement. A typical soft constraint generator considers only one aspect of the design, and formulates some favorable situations as soft constraints. The cost of violating a soft constraint can also be specified. Note that since conflicting soft constraints are allowed, a soft constraint generator does not need to consider tradeoffs with other design requirements or global feasibility of the constraint system. This makes it very easy to design a soft constraint generator and allows more accurate estimation models to be used. It is also possible to expose the interface of a soft constraint generator to the designer, so that the designer can gain detailed control over part of the design by manually adding soft constraints when necessary. The optimization engine then accepts all hard/soft constraints as well as the optimization goal, and makes intelligent decisions to get the results. The process can possibly be iterative by allowing adjustment on soft constraints after an initial result is available.

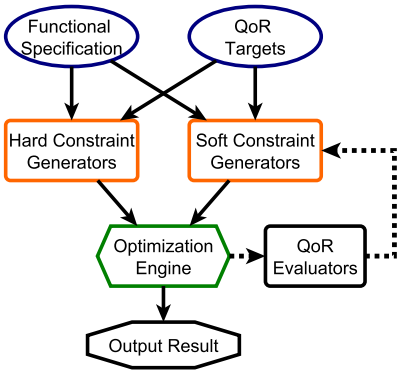

Figure 1: Structure of a scheduler with soft constraints.

Although the concept of soft constraints makes it easier to express various design intentions and to guide the scheduler, it brings major challenges for the implementation of the optimization engine. Classical algorithms for scheduling in the behavioral synthesis domain, like list scheduling [17, 18] and force-directed scheduling [24], are not designed to work with soft constraints. Some scheduling algorithms, like the iterative modulo scheduling for software pipelining [26], adopt iterative search strategies and allow earlier decisions to be adjusted later, so that they are able to work on highly constrained problems. However, we are not aware of any algorithm in this category that explicitly supports soft constraints. The soft scheduling algorithm [30], despite the name, actually adds hard constraints iteratively. Similar approaches include [3, 22]. Exact methods for scheduling with soft constraints have been developed in the artificial intelligence community, using branch and bound [23], integer-linear programming [29] or constraint satisfaction [28]. This class of methods works very well for planning and temporal reasoning problems with a small number of tasks but very complex hard/soft constraints; however, exponential time complexity prohibits the application of these methods to problems of a practical size in behavioral synthesis.

In this paper we investigate the use of soft constraints for a low-power scheduling problem and propose an efficient scheduling algorithm based on a mathematical programming formulation to support soft constraints in scheduling. The contribution of this work is twofold.

- We propose a method to perform scheduling with support for soft constraints. We find a class of soft constraints that can be efficiently handled using mathematical programming. Our method enables a methodological change on how the direction of optimization can be specified and how design space can be explored. To the best of knowledge, this is the first systematic way to formulate and support scheduling with soft constraints in the behavioral synthesis domain.
- We discuss possible applications of soft constraints in scheduling. In particular, we apply them to a low-power scheduling problem. Experimental results obtained using industrial gate-level power estimators after RTL synthesis show a significant power saving compared to a previous heuristic, and the results are close-to-optimal on all the cases where an exact formulation is affordable.

2. PRELIMINARIES

In a typical behavioral synthesis system, a compiler front-end optimizes behavioral descriptions in high-level languages like C and generates a *control/data flow graph* (CDFG). A CDFG is a graph $G = (V, E)$, where each node $v \in V$ represents an operation and each directed edge $e \in E$ represents a data flow or a control flow. For each operation v, an integer-valued *scheduling variable* s_v is introduced to represent the time slot in which operation v is performed. A *finite state machine with datapath* (FSMD) model [9] can be constructed once the scheduling variable for every operation is decided [8]. The task of scheduling is thus to decide s_v for every operation v.

Using the scheduling variables described above, the *system of difference constraints* (SDC) formulation was proposed to solve scheduling problems [8]. The method uses a special form of constraints in a linear program, so that it can avoid expensive branch and bound procedures in traditional ILP formulations for scheduling [10, 15] while still optimizing globally. SDC is flexible enough to model a wide variety of constraints and objectives in behavioral synthesis, and is extended in [16] to solve a time budgeting problem.

A special class of constraints, called *integer-difference constraints*, is used to model various design constraints in SDC.

DEFINITION 1 (INTEGER-DIFFERENCE CONSTRAINT). *An integer-difference constraint is a constraint of the form $s_u - s_v \leq d$, where d is a constant integer.*

Using a system of integer-difference constraints, [8] is able to model dependency constraints and timing constraints precisely, and model resource constraints heuristically. The advantage of integer-difference constraints is that they can be solved very efficiently using linear programming.

In this work we will show that the "soft version" of an integer-difference constraint, referred to as an *integer-difference soft constraint*, can also be handled efficiently in the optimization engine.

DEFINITION 2 (INTEGER-DIFFERENCE SOFT CONSTRAINT). *An integer-difference soft constraint is a soft constraint in the form of $s_u - s_v \leq d$, where d is a constant integer.*

Details on how to implement an optimization engine supporting integer-difference soft constraint are discussed in Section 4.

3. AN EXAMPLE APPLICATION OF SOFT CONSTRAINTS

Before describing our method for solving a scheduling problem with soft constraints, let us first consider a low-power scheduling problem where soft constraints can be applied naturally.

A scheduling strategy that enables power management during scheduling was proposed in [22]. The strategy, referred to as *operation gating* in this paper, exploits *observability don't-cares* at operation level to avoid unnecessary computations and save power. The basic idea is to schedule the operation that computes the condition (referred to as *condition operation* in the following, typically including comparison (CMP) and Boolean AND/OR/NOT operations) for a multiplexer early. This ensures that values not selected by the multiplexer can be identified at run time and power management techniques (clock gating, input isolation, power gating, etc.) can be applied to corresponding components to reduce energy consumption.

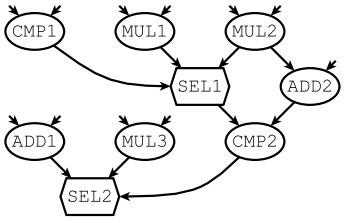

Figure 2: An example CDFG with opportunities for operation gating.

Consider the example CDFG in Figure 2, where a SEL operation selects one of its two data inputs as the result based on a Boolean condition input (it corresponds to a 2-input multiplexer in hardware). When CMP2 is evaluated as true for some input data, MUL3 will be *unobservable*, i.e., the value of MUL3 does not have any impact on the output of the module, and so the execution of MUL3 is unnecessary. Similarly, it is possible to avoid ADD1 when CMP2 is evaluated as false, or to avoid MUL1 when CMP1 is evaluated as false. In order to exploit such opportunities for power reduction, the condition operation must be scheduled earlier than the operation that can potentially be avoided — otherwise the operation cannot be avoided because it might be useful depending on the result of the condition operation. Figure 3 gives two possible schedules with the same latency: in the first schedule (shown on the left), we can possibly avoid ADD1 or MUL3 but not MUL1; in the second schedule, MUL1 can be avoided but ADD1 and MUL3 cannot. In this case, if CMP1 is evaluated as true most of the time, the first schedule will be a better choice than the second one.

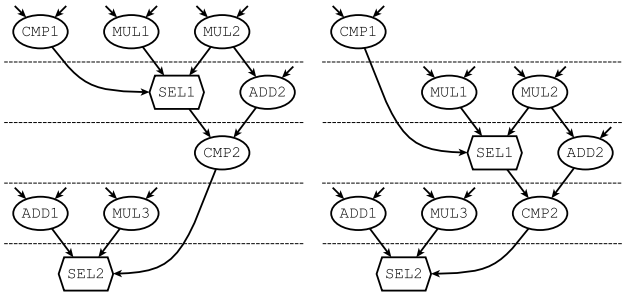

Figure 3: Two possible schedules for the CDFG in Figure 2.

It might be tempting to schedule every operation after its observability condition is completely resolved, so that the operation can be avoided when it is unobservable. However, this is not always possible, especially when a tight performance requirement is present. For example, it is not possible to avoid both MUL1 and MUL3 in Figure 2 when the total latency is limited to five clock cycles, assuming each operation takes one cycle. Thus, it is important to optimize the schedule to maximize the efficacy of operation gating. Operating gating efficacy is defined as the weighted number of operation executions that are avoided due to operation gating, with the weight being an estimated power of the corresponding operation. The problem of optimizing operation gating efficacy can be described as follows.

Given: (1) a CDFG $G = (V, E)$; (2) a set of hard constraints which include longest-path latency constraint, cycle time constraint, and possibly others; (3) profiling information; (4) a weight for each operation reflecting the power dissipation for performing the operation.

Goal: schedule every operation so that the operation gating efficacy is maximized and all hard constraints are satisfied.

To our knowledge, the first published work on operation gating for power reduction is [22], where multiplexers in the circuit (or SEL operations in the CDFG) are visited one by one. Virtual dependency edges are added from the condition operation to operations in the transitive fanin of the multiplexer, if latency constraint is not violated. Authors of [22] noticed that the results of their method depended on the order in which multiplexers were visited, and they used reverse topological order in their implementation. A more sophisticated heuristic method to decide the order is proposed in [3], using a priority function considering power saving and slack overhead. However, decisions on operation gating are still made one by one in a greedy manner, which probably leads to suboptimal results.

Using soft constraints, it is very easy to express the intention of operation gating. When it is preferred that a condition operation c is scheduled before another operation v so that v can be avoided when c takes a certain value, an integer-difference soft constraint can be added as

$$s_c - s_v \leq -b - d_c + 1, \tag{1}$$

where d_c is the number of clock cycles operation c spans, and b is

an integer constant depending on the power management technique and the target platform. A typical value of b is 1 if clock gating or operand isolation is used. Then the problem of power optimization using operation gating can be described in a mathematical-programming form as follows.

$$\begin{array}{lll} \min & \sum_k c_k s_k & \\ \text{s.t.} & s_{u_i} - s_{v_i} \le p_i, \quad i = 1,\ldots,m & \text{(hard constraints)} \\ & s_{c_j} - s_{v_j} \le q_j, \quad j = 1,\ldots,n & \text{(soft constraints)} \end{array} \tag{2}$$

Here p_i, q_j and c_k are constants. The linear objective function and all integer-difference hard constraints including dependency, cycle time, latency, resource, etc. are all formulated using techniques in the SDC formulation [8]. An integer-difference soft constraint is added between every operation pair (c_j, v_j), where c_j is a condition operation and v_j is an operation that can potentially be avoided when c_j takes a certain value. In the next section, we describe a method that considers all hard constraints and soft constraints together and optimizes globally.

4. HANDLING SOFT CONSTRAINTS

In this section we describe our method for handling soft constraints using linear programming. Our method extends the approach in SDC by supporting integer-difference soft constraints, while still guaranteeing efficient optimization.

4.1 A Penalty Method for Soft Constraints

To ease discussion, we can write the formulation in Equation 2 in vector and matrix form as

$$\begin{array}{lll} \min & \boldsymbol{c^T s} & \\ \text{s.t.} & \boldsymbol{Gs \le p} & \text{(hard constraints)} \\ & \boldsymbol{Hs \le q} & \text{(soft constraints).} \end{array} \tag{3}$$

Although a soft constraint does not need to be satisfied, there is usually a cost when it is violated. Let $\boldsymbol{H_j}$ be the jth row of $\boldsymbol{H}$. For each soft constraint $\boldsymbol{H_j s} \le q_j$, we introduce a violation variable v_j to denote the amount of violation. Then the soft constraint is transformed to two traditional constraints as

$$\begin{array}{rcl} \boldsymbol{H_j s} - v_j & \le & q_j, \\ -v_j & \le & 0. \end{array} \tag{4}$$

A penalty term depending on the amount of violation, $\phi_j(v_j)$, is added in the objective function to denote the cost for violating the jth soft constraint. Then the problem is formulated as a traditional form of mathematical programming as follows.

$$\begin{array}{lrcl} \min & \boldsymbol{c^T s} + \sum_{j=1}^{n} \phi_j(v_j) & & \\ \text{s.t.} & \boldsymbol{Gs} & \le & \boldsymbol{p} \\ & \boldsymbol{Hs} - \boldsymbol{v} & \le & \boldsymbol{q} \\ & -\boldsymbol{v} & \le & \boldsymbol{0}. \end{array} \tag{5}$$

The constraints can also be written in matrix form as

$$\begin{bmatrix} \boldsymbol{G} & \boldsymbol{O} \\ \boldsymbol{H} & -\boldsymbol{I} \\ \boldsymbol{O} & -\boldsymbol{I} \end{bmatrix} \begin{bmatrix} \boldsymbol{x} \\ \boldsymbol{v} \end{bmatrix} \le \begin{bmatrix} \boldsymbol{p} \\ \boldsymbol{q} \\ \boldsymbol{0} \end{bmatrix}. \tag{6}$$

4.2 Total Unimodularity and Implications

In the SDC formulation, total unimodularity is exploited to avoid branch and bound procedures while still guaranteeing integral solutions.

DEFINITION 3 (TOTAL UNIMODULARITY). *A matrix $\boldsymbol{A}$ is totally unimodular if every square submatrix of $\boldsymbol{A}$ has a determinant either 0, 1 or -1.*

Total unimodularity plays an important role in combinatorial optimization, as shown in Lemma 1.

LEMMA 1 (HOFFMAN AND KRUSKAL, [14]). *If $\boldsymbol{A}$ is totally unimodular and $\boldsymbol{b}$ is a vector of integers, every extreme point of polyhedron $\{\boldsymbol{x} : \boldsymbol{Ax} \le \boldsymbol{b}\}$ is integral.*

Lemma 1 implies that an integer linear programming problem

$$\begin{array}{ll} \min & \boldsymbol{c^T x} \\ \text{s.t.} & \boldsymbol{Ax} \le \boldsymbol{b} \\ & \boldsymbol{x} \in \mathbb{Z}^n \end{array}$$

can be solved by solving a linear-programming relaxation

$$\begin{array}{ll} \min & \boldsymbol{c^T x} \\ \text{s.t.} & \boldsymbol{Ax} \le \boldsymbol{b} \end{array}$$

if $\boldsymbol{A}$ is totally unimodular and $\boldsymbol{b}$ is integral, because the solution of the linear program can always be found at an extreme point of the polyhedron $\{\boldsymbol{x} : \boldsymbol{Ax} \le \boldsymbol{b}\}$.

LEMMA 2 (CONG AND ZHANG, [8]). *With only integer-difference constraints, the constraint matrix of the scheduling problem is totally unimodular.*

By exploiting the total unimodularity, the SDC formulation of scheduling can be efficiently solved using linear programming. Integral solutions are guaranteed without the expensive branch-and-bound procedure in typical integer-linear programming solvers.

LEMMA 3 (RAGHAVACHARI, [25]). *If a $m \times n$ matrix $\boldsymbol{A} \in \{-1,0,1\}^{m \times n}$ has a row (or a column) with at most one nonzero element, $\boldsymbol{A}$ is totally unimodular if and only if the resulting matrix, after removing the row (or column) from $\boldsymbol{A}$, is totally unimodular.*

THEOREM 1. *With only integer-difference constraints and integer-difference soft constraints, the constraint matrix in the optimization program in Eqn. 5 is totally unimodular.*

PROOF. Using Lemma 3, we can first remove the last n rows of the matrix in Eqn. 6, and then remove the last n columns of the resulting matrix, without changing total unimodularity.

$$\begin{bmatrix} \boldsymbol{G} & \boldsymbol{O} \\ \boldsymbol{H} & -\boldsymbol{I} \\ \boldsymbol{O} & -\boldsymbol{I} \end{bmatrix} \Rightarrow \begin{bmatrix} \boldsymbol{G} & \boldsymbol{O} \\ \boldsymbol{H} & -\boldsymbol{I} \end{bmatrix} \Rightarrow \begin{bmatrix} \boldsymbol{G} \\ \boldsymbol{H} \end{bmatrix}.$$

Then we only need to show that $\begin{bmatrix} \boldsymbol{G} \\ \boldsymbol{H} \end{bmatrix}$ is totally unimodular. Note that each row of the matrix represents the difference of two variables, so the matrix is in the same form as the constraint matrix in the SDC formulation, which is thus totally unimodular according to Lemma 2. □

4.3 Form of the Penalty Term

The penalty term for violation in the objective function, $\phi_j(v_j)$, is supposed to model the cost of violating the jth soft constraint. Typically, ϕ_j is 0 when there is no violation, and it is a non-decreasing function for positive v_j. That is,

$$\phi_j(v_j) = 0, \text{ when } v_j = 0, \tag{7}$$

$$\phi_j(v_j) \ge \phi_j(v'_j) \ge 0, \text{ when } v_j \ge v'_j \ge 0. \tag{8}$$

Note that it is impossible to have a negative v_j because the constraints in our formulation forbid it (Eqn. 4).

If the cost is linear to the amount of violation, i.e.,

$$\phi_j(v_j) = \alpha_j v_j, \tag{9}$$

where α_j is a nonnegative constant, the problem in Eqn. 5 is a linear program with a totally unimodular constraint matrix and integral right-hand side. According to Lemma 1, we can get optimal integral solution efficiently.

In fact, our formulation allows $\phi_j(v_j)$ to be a nonlinear convex function of v_j without introducing significant complexity based on the following result.

LEMMA 4 (HOCHBAUM AND SHANTHIKUMAR, [13]). *For the integer convex separable optimization problem*

$$\begin{array}{ll} \min & \sum_{i=1}^{n} f_i(x) \\ s.t. & \boldsymbol{A}\boldsymbol{x} \leq \boldsymbol{b} \\ & \boldsymbol{x} \in \mathbb{Z}^n, \end{array} \tag{10}$$

where f_i is convex, $\boldsymbol{A}$ is totally unimodular and $\boldsymbol{b}$ is integral, the optimal solution can be found in polynomial time.

It is shown in [21] that the problem in Eqn. 10 can be translated into a linear program by piece-wise linearizing f_i at integral points and removing the integral constraints. The resulting linear program is guaranteed to have an integral solution. Therefore, we have the following theorem.

THEOREM 2. *When only integer-difference constraints and integer-difference soft constraints are used, and every ϕ_j is convex, the problem can be solved optimally with an integral solution in polynomial time.*

Theorem 2 allows a rich category of penalty functions to be used in our formulation, including linear, quadratic, exponential and log barrier functions, as illustrated in Figure 4. The selection of the penalty form and its parameters can be performed on each individual soft constraint based on characteristics of the constraint. For some cases, the simple linear model will work well; for some others, the cost increases exponentially with the amount of violation. Sometimes the log-barrier form is also useful to keep the amount of violation within a certain range.

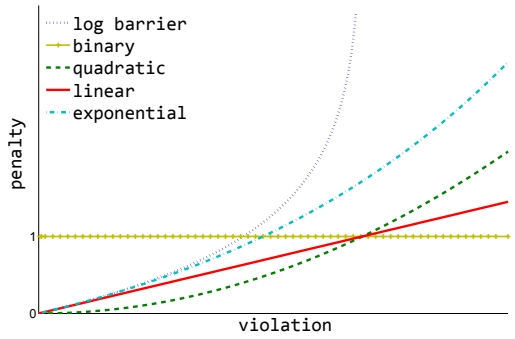

Figure 4: Some typical penalty functions. An offset is added to the exponential penalty to satisfy Eqn. 7.

Notably, although the cost of violating a soft constraint is often convex with respect to the amount of violation, it is not always the case. For example, the cost for violating some soft constraints can be a constant for any amount of non-zero violation (denoted as the binary penalty function). Let $b(x)$ be the binary penalty function with unit coefficient, defined as

$$b(x) = \begin{cases} 1 & \text{if } x > 0, \\ 0 & \text{if } x \leq 0. \end{cases} \tag{11}$$

Unfortunately, introducing such a penalty function directly to the formulation in Eqn. 5 will make the problem NP-hard.[1] So we take a different approach. We first check every soft constraint with binary penalty and eliminate the soft constraint if it obviously conflict with hard constraints. For the remaining soft constraints, we iteratively solve a sequence of subproblems with slightly different convex penalty functions to gradually approximate $b(x)$. This is motivated by the Lagrangian method [12], as well as the method for linear wire length optimization using a quadratic model in GordianL [27]. At iteration k, we use a linear function $b^{(k)}(x)$ to approximate $b(x)$ near $x^{(k-1)}$ (the value of x in the solution of the previous iteration).

$$b^{(k)}(x) = \frac{x}{\max(1, x^{(k-1)})} \tag{12}$$

The idea is that when the process converges after $k-1$ iterations, we will have $x^{(k)} = x^{(k-1)}$ and $x^{(k)}$ is a nonnegative integer; thus $b^{(k)}(x^{(k)}) = b(x^{(k)})$. For an intermediate iteration, when the violation is larger than one, the coefficient in the penalty term is scaled down in the next iteration, effectively reducing the effort to satisfy the corresponding soft constraint. In the extreme case where the violation is very large, the optimization engine will tend to ignore the soft constraint. Note that this technique cannot guarantee the optimality of the solution, but it turns out a good heuristic and the iterative process usually converges after several (less than four) iterations in our experiments.

4.4 Overall Flow

Since our optimization engine is highly efficient by taking advantage of the special constraint structure, it is feasible to run the optimization engine repeatedly in an iterative manner: after each iteration, we can adjust the soft constraints and solve again for better QoR. There are many ways soft constraints can be adjusted: a soft constraint can be added or deleted, and the penalty function of a soft constraint can be changed (e.g., the technique discussed in Subsection 4.3 that deals with binary cost). The detailed strategy for adjusting should be designed considering characteristics of the soft constraint.

When various soft constraints designed for different aspects of QoR are present, it is important that the penalty terms be carefully designed to allow a reasonable tradeoff. Orthogonal to the selection of the form of the penalty function (among which are linear, quadratic, exponential), the coefficient (weight) in the penalty term can always be adjusted in the objective function. Note that the cost of violating a soft constraint can usually be interpreted as overhead in certain QoR metrics like performance/area/power. Thus, adjusting the coefficients for different types of soft constraints enables different tradeoffs. For example, if tradeoffs between power and area are desirable, we may want to select the weights so that the penalty of a 10% power increase is comparable with that of a 10% area increase. To do this for a soft constraint considering area, we can normalize the area penalty for unit violation (δ_A) to the estimated total area (A), and multiply it by a manually selected constant α. Then the penalty term is

$$\phi(v) = \alpha \frac{\delta_A}{A} p(v),$$

where $p(v)$ is a convex function as discussed in Subsection 4.3. The same can be done for a soft constraint for power consideration. In practice, a robust approach is to run the optimization multiple times with different combinations of weights to get a sequence of Pareto

[1] A reduction from MAX-2-SAT is straightforward, omitted here due to page limitation.

optimal solutions — and then the designer can select a subset of solutions to go through later design processes.

5. SOFT CONSTRAINT GENERATOR FOR OPERATION GATING

The motivation and formulation for operation gating using soft constraints have been discussed in Section 3. In this section, we describe details of the soft constraint generator for operation gating. In particular, we describe the method used to find candidates for operation gating, and the penalty term associated with the soft constraint.

5.1 Identification of Gating Candidates

To describe our method of identifying candidate operations for gating, we adopt the concept of *fanout-free subgraph* (*FFS*) and *maximum fanout-free subgraph* (*MFFS*) defined in [7].

DEFINITION 4 (FFS & MFFS). *In a direct acyclic graph $G=(V,E)$ with the set of sink nodes denoted as $T \subseteq V$, for a set of nodes $R \subseteq V$,*

$$FFS_R = \{u \in V \mid \textit{for any path } p \textit{ from } u \textit{ to a sink node } w, \textit{ } p \textit{ passes through } v \textit{ in } R\},$$
$$MFFS_R = \{u \in V \mid u \textit{ is in some } FFS_R\}.$$

$MFFS_R$ can be computed in $O(|V|+|E|)$ [7].

Informally, the maximum fanout-free subgraph of a data-flow graph induced by a subset of operations R is the set of operations that influences the output only by influencing values in R. Let $R_{\mathsf{c}=\text{true}}$ ($R_{\mathsf{c}=\text{false}}$) be a set of unobservable operations when a condition operation c is evaluated as true (false). Then all other operations in $MFFS_{R_{\mathsf{c}=\text{true}}}$ are also unobservable when c is evaluated as true. Thus, we only need to find a number of "seed" operations that can be avoided.

Without loss of generality, consider the case when a condition operation c is evaluated as true. A straightforward situation where an operation u can be avoided is when it is only used as the false-input by a SEL operation with c as the condition input. Actually, if u is only used as the false-input by multiple such SEL operations, it can still be avoided. For example, MUL1 in Figure 5 is unobservable when CMP1 is evaluated as true. However, if the operation is also used by other operations, it cannot be avoided; for example, ADD1 in Figure 5 can still be necessary even if CMP1 is evaluated as false, because it is used by MUL2.

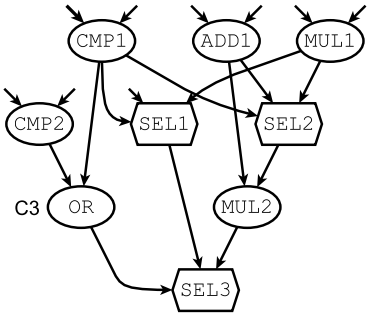

Figure 5: Another example CDFG with correlations between condition operations.

In addition to the analysis based purely on topology, we also exploit the propagation of logic values to uncover more opportunities for operation gating. Consider the case when CMP1 is evaluated as true in Figure 5; the naive method will give $R_{\mathsf{CMP1}=\text{true}} = \{\mathsf{MUL1}\}$. If we analyze the propagation of logic values, we find that C3 = CMP1 OR CMP2 will be true no matter what value value CMP2 takes — thus $\mathsf{CMP2} \in R_{\mathsf{CMP1}=\text{true}}$. Since C3 is true, we can further find that MUL2 is unobservable and thus $\mathsf{MUL2} \in R_{\mathsf{CMP1}=\text{true}}$. Then we can compute $R_{\mathsf{CMP1}=\text{true}} = MFFS_{\{\mathsf{MUL1},\mathsf{MUL2},\mathsf{CMP2}\}}$ and get $\{\mathsf{ADD1},\mathsf{MUL1},\mathsf{MUL2},\mathsf{CMP2},\mathsf{SEL2}\}$ as the set of operations that can be avoided when CMP1 is evaluated as true.

Compared to the method used in [3, 22], our approach considers more than one multiplexer simultaneously, and is able to uncover more opportunities for operation gating with analysis of Boolean AND/OR/NOT operations. For a formal description of the algorithm to compute R for every Boolean value, interested readers may refer to [6].

5.2 Soft Constraint Generation

We use the binary function described in Subsection 4.3 to penalize the violation of a soft constraint for operation gating between condition operation c and another operation v. The coefficient in the penalty function is the estimate of potential energy saving if the soft constraint is satisfied based on profiling information.

Such an estimate, however, can be nontrivial, because the execution of operation v may also depend on other condition operations. For example, in Figure 5, we have $\mathsf{MUL2} \in R_{\mathsf{CMP1}=\text{true}}$ and $\mathsf{MUL2} \in R_{\mathsf{CMP2}=\text{true}}$ (i.e., MUL2 is unobservable when either CMP1 or CMP2 is evaluated as true). Then, even if the soft constraint that CMP2 is scheduled earlier than MUL2 is not satisfied, MUL2 may still be avoided if it is scheduled after CMP1. Thus, an exact evaluation of the energy saving for such a soft constraint will require a full profiling of all possible value combinations in a group of condition operations. However, such a process can be prohibitively expensive. In practical compilers and synthesis systems, profiling is usually done on each individual value, instead of on combinations of values [11]. So, we assume that each Boolean value by a CMP operation is independent. Without loss of generality, suppose $\{\mathsf{c1},\mathsf{c2},\ldots,\mathsf{cn}\}$ is the set of independent conditions (with p_i being the probability that ci is true), and v is unobservable if any condition in the set is true. When we estimate the cost of violating a soft constraint between cj and v, we assume that other constraints are satisfied (if a soft constraint can never be met given all the hard constraints, or is not met in previous design iterations, it is excluded), then the probability of executing v when the soft constraint is satisfied is $\prod_{i=1}^{n}(1-p_i)$, and the probability becomes $\frac{\prod_{i=1}^{n}(1-p_i)}{1-p_j}$ when the soft constraint is violated. Then we get the cost of violating the soft constraint considering the increase in the execution probability of v as

$$cost^{\mathsf{v}}_{\mathsf{cj}} = \frac{p_j}{1-p_j}\left(\prod_{i=1}^{n}(1-p_i)\right)E_{\mathsf{v}}, \tag{13}$$

where E_{v} is the energy consumed for performing operation v. This $cost^{\mathsf{v}}_{\mathsf{cj}}$ is helpful to decide the weight of the binary penalty function.

6. EXPERIMENTAL RESULTS

6.1 Experiment Setup

Techniques proposed in this paper have been implemented in the scheduler of AutoPilot, a commercial behavioral synthesis tool from AutoESL Design Technologies, Inc. [1]. The tool accepts C/C++/SystemC as the input language and generates RTL specifications in VHDL or Verilog. Here we evaluate the effectiveness of our approach on the problem of latency-constrained operation gating for power reduction as described in Section 3.

Our scheduler introduces soft constraints, formulates the problem using techniques described in Section 4, and uses the iterative technique to approximate the binary penalty functions. We make comparisons to three other approaches: (1) a baseline scheduler using the SDC formulation without operation gating; (2) the iterative algorithm described in [3] for operation gating; (3) an integer-linear programming (ILP) formulation to handle binary penalty exactly for optimal operation gating. We will not compare our approach with the original work on operation gating in [22] as [3] is algorithmically similar to [22] with improved strategy. All these approaches are implemented in C++, and the programs run on a workstation with four 2.4GHz 64-bit CPU and 8G primary memory.

The ILP formulation (for the purpose of optimality study) is briefly described as follows. In addition to all variables and constraints in Eqn. 5, a variable w_i is introduced for each violation variable v_i in a binary penalty function. We add constraints

$$v_i - N \times w_i \leq 0, \tag{14}$$

$$w_i \in \{0,1\}, \tag{15}$$

where N is a large constant number so that the constraint in Eqn. 14 can always be satisfied when $w_i = 1$. Then we replace $b(v_i)$ with w_i in the objective function, and explicitly enforce the constraint that every variable is an integer. It is easy to verify that in the solution of the ILP formulation, we have

$$w_i = \begin{cases} 1 & \text{when } v_i > 0, \\ 0 & \text{otherwise,} \end{cases}$$

which means $w_i = b(v_i)$.

After scheduling, a binding algorithm described in [4] is performed. The RTL code generated by the behavioral synthesis tool is fed to the Magma Talus RTL-to-GDSII toolset. Gate-level simulation under typical input vectors is performed using the Aldec Riviera simulator to obtain power dissipation. All designs are implemented using a TSMC 90nm standard cell library. In this experiment the actual operation gating is carried out by the clock gating on the output registers of the gated operations. Further power savings can potentially be achieved if we apply additional low-power techniques (e.g., feeding sleep vectors for leakage reduction).

Several designs in arithmetic and multimedia applications are used in our experiments. Characteristics of these designs are given in Table 1.

Table 1: Benchmark Characteristics.

Name	#node	Description
addr	88	address space translation unit
BoxMuller	333	Gaussian noise generator
dfmul	351	floating-point multiplier
MotionComp	1306	motion compensation (MPEG4 decoder)
MotionEst	621	motion estimation (MPEG4 encoder)

6.2 Results and Analysis

Results of the four approaches are reported in Table 2. Here, area and power after gate-level implementation are reported for each approach. Since the Magma Talus synthesis tool meets the clock cycle time constraint for all cases, we do not report the frequency for each individual approach. We also normalize the power values to those generated by the approach with soft constraints. For some larger designs, the exact ILP formulation (solved by Clp [2], a state-of-the-art open-source ILP solver) fails to find a solution within 7200 seconds. All the three other methods finish within 60 seconds for all cases.

From the results, it is clear that operation gating is a useful technique to create opportunities for power management at the RT level without significant overhead in area. Compared to the SDC scheduling algorithm without considering operation gating, all of the three other methods that optimize for operation gating improve the power dissipation: on average, the method in [3] reduces power by 20.1%, the exact method given by ILP reduces power by 34.6%, and our proposed method by 33.9%. The reduction tends to be particularly significant when the design has a complex control structure, like addr. When large memory blocks are present and the access pattern is fixed (for example, the total size of RAM in MotionEst is over 10Mb and the memory traffic is fixed; the situation is similar for MotionComp), operation gating tends to be less effective, because the memory power is roughly a constant. While power consumed in memory blocks can be a very important part of total power, it is usually not controlled by the operation scheduler when memory operations are unavoidable and the access pattern is fixed. Possible techniques that help to reduce memory power include behavioral transformations (loop transformation to enhance memory locality, to leverage burst-mode memory access, etc.), memory architecture selection, memory partitioning [5], etc, but those are beyond the scope of this study. For a fair comparison, we include the memory power for every design in Table 2.

Compared to [3], the proposed approach further reduces total power dissipation by an average of 17.1%. As discussed above, the reduction can be more significant if memory power is excluded. This saving is because we are able to consider all opportunities for operation gating simultaneously, and optimize globally in our approach. The approximation of binary penalty function turns out to work very well — the results generated using our approach are very close to those by the exact formulation, and the observed optimality gap in term of power is about 1%. At the same time, our method is much more scalable than the exact formulation.

7. CONCLUSION

In this work we introduce the concept of soft constraints to scheduling in behavioral synthesis and show its application in power optimization. Following the idea of [8], we find a class of soft constraints — integer-difference soft constraints, which enable optimal solutions within polynomial time. Potentially conflicting soft constraints are allowed, so that various optimization strategies can be specified easily, either by automatic soft constraint generators or possibly by the designer. The optimization engine is then able to make tradeoffs with a global view of the design objective and hard constraints, as well as all soft constraints together. Such a scheduler not only provides a more user-friendly interface to the designer, but also allows a methodological change in how the scheduler can be guided and how the design space can be explored. More soft constraint generators, such as ones for chaining and slack distribution, can be developed to make our scheduler more versatile and powerful. We believe that automatic soft constraint generators based on machine learning and statistical methods are possible, and we leave them for future work.

8. ACKNOWLEDGMENTS

This work is partially supported by the Semiconductor Research Corporation under Contract 2009-TJ-1879. The authors are grateful to Ms. Janice Martin-Wheeler for assistance in editing.

9. REFERENCES

[1] http://www.autoesl.com.

Table 2: Experimental Results.

design	cycle	SDC [8]			iterative [3]			ILP			soft constraints		
		area	power	N_p	area	power	N_p	area	power	N_p	area	power	N_p
addr	2.0	5628	1.16	2.32	5771	0.71	1.42	5775	0.49	0.98	5775	0.50	1.00
BoxMuller	3.5	72127	2.83	1.40	72258	2.20	1.09	72810	2.02	1.00	72810	2.02	1.00
dfmul	3.5	143546	7.15	1.39	150335	6.14	1.19	–	–	–	146938	5.16	1.00
MotionComp	3.0	57850	5.03	1.38	57983	4.01	1.10	–	–	–	58657	3.64	1.00
MotionEst	3.0	66341	9.36	1.27	66137	9.25	1.26	–	–	–	66158	7.36	1.00
geomean				1.51			1.21			0.99			1.00

Cycle is in ns, area is in μm^2, and power is in mW. The N_p column is the normalized power.

[2] `http://projects.coin-or.org/Clp`.

[3] C. Chen and M. Sarrafzadeh. Power-manageable scheduling technique for control dominated high-level synthesis. In *Proc. Design, Automation & Test in Europe*, pages 1016–1020, 2002.

[4] J. Cong, Y. Fan, and W. Jiang. Platform-based resource binding using a distributed register-file microarchitecture. In *Proc. Int. Conf. Computer Aided Design*, pages 709–715, 2006.

[5] J. Cong, W. Jiang, B. Liu, and Y. Zou. Automatic memory partitioning and scheduling for throughput and power optimization. In *Proc. Int. Conf. Computer Aided Design*, 2009.

[6] J. Cong, B. Liu, and Z. Zhang. Behavior-level observability don't-cares and application to low-power behavioral synthesis. In *Proc. Int. Symp. Low Power Electronics and Design*, pages 139–144, 2009.

[7] J. Cong and S. Xu. Technology mapping for fpgas with embedded memory blocks. In *Proc. Int. Symp. FPGA*, pages 179–188, 1998.

[8] J. Cong and Z. Zhang. An efficient and versatile scheduling algorithm based on sdc formulation. In *Proc. Design Automation Conf.*, pages 433–438, 2006.

[9] D. D. Gajski, N. D. Dutt, A. C.-H. Wu, and S. Y.-L. Lin. *High-level synthesis: introduction to chip and system design*. Kluwer Academic Publishers, 1992.

[10] C. H. Gebotys and M. Elmasry. Global optimization approach for architectural synthesis. *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, 12(9):1266–1278, Sept. 1993.

[11] R. Gupta, E. Mehofer, and Y. Zhang. Profile guided compiler optimization. In Y. N. Srikant and P. Shankar, editors, *The Compiler Design Handbook: Optimization and Machine Code Generation*, pages 143–174. CRC Press, 2003.

[12] M. R. Hestenes. Multiplier and gradient methods. *J. Optimization Theory and Applications*, 4:303–320, 1969.

[13] D. S. Hochbaum and J. G. Shanthikumar. Convex separable optimization is not much harder than linear optimization. *J. ACM*, 37(4):843–862, 1990.

[14] A. J. Hoffman and J. B. Kruskal. Integral boundary points of convex polyhedra. In H. W. Kuhn and A. W. Tucker, editors, *Linear Inequalities and Related Systems*, pages 22–46. Princeton University Press, 1956.

[15] C.-T. Hwang, J.-H. Lee, and Y.-C. Hsu. A formal approach to the scheduling problem in high level synthesis. *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, 10(4):464–475, Apr. 1991.

[16] W. Jiang, Z. Zhang, M. Potkonjak, and J. Cong. Scheduling with integer delay budgeting for low-power optimization. In *Proc. Asia South Pacific Design Automation Conf.*, pages 22–27, Jan. 2008.

[17] A. A. Kountouris and C. Wolinski. Efficient scheduling of conditional behaviors for high-level synthesis. *ACM Trans. Des. Autom. Electron. Syst.*, 7(3):380–412, 2002.

[18] D. Landskov, S. Davidson, B. Shriver, and P. W. Mallett. Local microcode compaction techniques. *ACM Comput. Surv.*, 12(3):261–294, 1980.

[19] T. Ly, D. Knapp, R. Miller, and D. MacMillen. Scheduling using behavioral templates. In *Proc. Design Automation Conf.*, pages 101–106, 1995.

[20] G. D. Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Higher Education, 1994.

[21] A. J. Miller and L. A. Wolsey. Tight formulations for some simple mixed integer programs and convex objective integer programs. *Mathematical Programming*, 98(1-3):73–78, 2003.

[22] J. Monteiro, S. Devadas, P. Ashar, and A. Mauskar. Scheduling techniques to enable power management. In *Proc. Design Automation Conf.*, pages 349–352, 1996.

[23] M. Mouhoub and A. Sukpan. Managing temporal constraints with preferences. *Spatial Cognition & Computation*, 8(1):131–149, 2008.

[24] P. G. Paulin and J. P. Knight. Force-directed scheduling for the behavioral synthesis of asics. *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, 8(6):661–679, 1989.

[25] M. Raghavachari. A constructive method to recognize the total unimodularity of a matrix. *J. Mathematical Methods of Operations Research*, 20(1):59–61, Jan. 1976.

[26] B. R. Rau. Iterative modulo scheduling: an algorithm for software pipelining loops. In *Proc. Int. Symp. Microachitecture*, pages 63–74, 1994.

[27] G. Sigl, K. Doll, and F. M. Johannes. Analytical placement: A linear or a quadratic objective function? In *Proc. Design Automation Conf.*, pages 427–432, 1991.

[28] T. C. Son and E. Pontelli. Planning with preferences using logic programming. *Theory Pract. Log. Program.*, 6(5):559–607, 2006.

[29] M. van den Briel, S. Kambhampati, and T. Vossen. Planning with preferences and trajectory constraints through integer programming. In *Proc. ICAPS Workshop on Preferences and Soft Constraints in Planning*, pages 19–22, 2006.

[30] J. Zhu and D. D. Gajski. Soft scheduling in high level synthesis. In *Proc. Design Automation Conf.*, pages 219–224, 1999.