On Incremental Component Implementation Selection in System Synthesis

Soheil Ghiasi, *Member, IEEE*

***Abstract*—Incremental design methods can substantially improve products' time-to-market through efficient handling of engineering change orders (ECO). In this paper, we present a methodology for incrementally solving component implementation selection problem (CISP) in face of local or non-local perturbations. CISP, which refers to judicious selection of components implementation under system timing constraint, is a generic problem that implicitly or explicitly appears in many stages of CAD flow. For a commonly-used formulation of CISP, we discuss necessary and sufficient conditions for optimality of the solution. Based on the optimality conditions, we develop an algorithm that maintains both validity and optimality of a solution under incremental changes. We evaluated our approach by incrementally updating the threshold voltage assignment solution for a netlist going through engineering changes. On average, our method ran 283 times faster than the *full* solver, while delivering the same results.**

***Index Terms*—Incremental, Library Mapping, Logic Synthesis**

I. Introduction

ENGINEERING change orders (ECO) modify design specifications or constraints during development. ECOs usually impose additional design iterations, which lengthen the design process and hinder time to market. Realistically though, most ECOs are incremental in nature. Incremental changes might not require a computationally-intensive "full" solution to CAD problems, if the solution determined in previous iterations can be quickly and efficiently updated to address the incremental perturbations [1].

The problem of component implementation selection is a generic formulation for mapping design components to library modules under timing constraints, which is either implicitly or explicitly solved in different stages of library-based CAD flow. CISP aims to select the proper implementation of each component, from a number of choices available in the library, such that design timing constraint is met and some cost function, such as overall energy dissipation, is minimized.

CISP relates to the conventional slack distribution problem, which has been studied extensively in different contexts, such as wire and gate sizing [2], design timing closure [3], energy savings via voltage and frequency adjustment [4], and high-level synthesis [5]. The majority of existing results use variations of Zero Slack Algorithm (ZSA) [6] to iteratively distribute timing relaxation to non-critical components of a design. Such approaches are known to be sub-optimal [7], [8].

Incremental CISP can be utilized in two broad ways: First, perturbations can be applied to incrementally optimize some design quality metric, subject to meeting design constraints. An example would be [9], which applies perturbations post-placement to improve timing. The second group of incremental CISP methods, which are the focus of our work, try to revalidate constraints with minimal metric degradation in face of a given set of perturbation. Our technique maintains the optimality of the full solution, while guaranteeing to meet the timing constraint. We present necessary and sufficient conditions for quick validation of a candidate solution, and develop guidelines to handle primitive incremental operations such as insertion or deletion of a net in the design.

S. Ghiasi is with the Department of Electrical and Computer Engineering, University of California, Davis, CA, 95616 USA.

Manuscript received April 19, 2008; revised January 11, 2009.

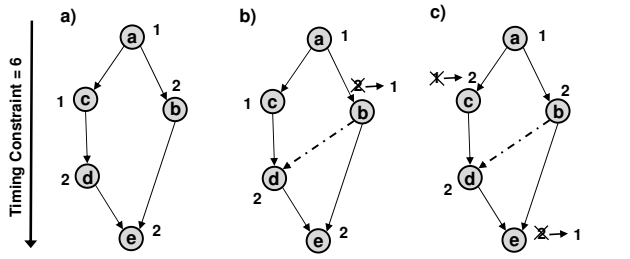

Fig. 1. a) Example optimal solution. b) After inserting an arc, greedy selection correction is sub-optimal. c) Optimal solution. Nodes b and d that are incident to the inserted arc, are not modified.

II. Motivating Example

Figure 1 depicts a simple example to illustrate the idea of incremental CISP. Nodes and edges in the graph represent components and dependencies, respectively. All nodes are assumed to have two possible implementations in the library: the first implementation has unit delay and dissipates two units of energy, while the second implementation has two units of delay and dissipates unit energy.

Figure 1.a shows the optimal selection of node implementations, such that the timing constraint of 6 time units is met, and the total energy dissipation is minimized. The optimal solution dissipates 7 units of energy. Now, let us assume that as a result of some ECO, arc $b \rightarrow d$ is added to the DAG. Insertion of the new arc creates the path $a \rightarrow b \rightarrow d \rightarrow e$, which violates the timing constraint.

A greedy approach would try to select a faster implementation for nodes incident to the arc, if possible, to meet the timing. Figure 1.b shows the resulting solution, which dissipates 8 units of energy. The optimal solution, however, dissipates 7 units of energy (Figure 1.c) by modifying the implementations of nodes c and e. Our objective is to develop an incremental algorithm to revalidate the solution (i.e., meet the timing) while maintaining its optimality, under incremental perturbations to the design.

III. Background and Preliminaries

We focus on designs that are modeled using directed acyclic graphs (DAG), such as gate-level netlists. We assume that a given design is represented as DAG $G = (V, E)$, where V is a set of vertices and E is a set of directed edges. Vertices, also referred to as nodes, model the components (e.g. gates) in the design. We use the terms vertex, node and component interchangeably. We use the notations I and O to refer to primary inputs and outputs, respectively.

Each vertex $v \in V$ is associated with a delay $d(v)$ which represents the time it takes for a signal to pass through v. Edge delays are assumed to be zero, nevertheless interconnect delay can be modeled by inserting "interconnect delay" nodes on E. Node delays can be calculated using common gate-level delay models such as load-dependent model. We assume that the signal arrives at the primary inputs at time zero. The latest time of signals to arrive at the output of any vertex $v \in V - I$ is given recursively by $a(v) = max_{u \in FI(v)} d(v) + a(u)$, where $a(v)$ is the arrival time of the signal at node v, and $FI(v)$ is the set of immediate fan-ins of v. The design is required to meet a given timing constraint T, i.e., $a(v) \leq T$ has to hold for all primary outputs.

A path $u \rightarrow v$ is a sequence of connected directed edges that form a trail along which, the signal from u can reach v. The delay of a path is defined as the sum total of the delay of the nodes on that path. Path delay is an abstract notion and does not necessarily correlate with signal traveling delay in digital circuitry. The definition, however, will assist us in our forthcoming discussion. To simplify the timing analysis in our discussion, we insert a special node, called the super primary output SO (the super primary input SI), in the graph to form a graph with a single output (input) node. SO (SI) is an abstract node with zero delay. All primary outputs are connected to SO, and it has no other fanins. Similarly, SI is connected to all primary inputs and it has no other fanouts. The delay of all $SI \rightarrow SO$ paths must be smaller than or equal to the specified critical path constraint for the design to meet its timing constraint. If the network already has one primary output (input), it is treated as the SO (SI).

A. Component Implementation Selection Problem

There are usually a number of implementations available in the library to implement components of a design. Different implementations come at different costs. Faster implementations of components incur higher "cost" in terms of typical design quality metrics, such as energy dissipation, area or dollar cost. For example, a complex gate can be implemented in several ways by trading off energy for delay. In our discussion, we use the terms cost of a component to refer to such quality metrics without loss of generality. For simplicity, we *temporarily* assume that the cost of implementing any component is a linear function of its delay. In Section VI, we extend our discussion to address a larger class of cost functions.

An essential task in design flow is to select the proper implementation for components of the design such that its timing constraint is met, and overall design cost is minimized. The overall design cost is sum total of the cost of selected component implementations. We refer to this problem as *component implementation selection problem (CISP)*, which is known to be NP-Hard in the general case [10].

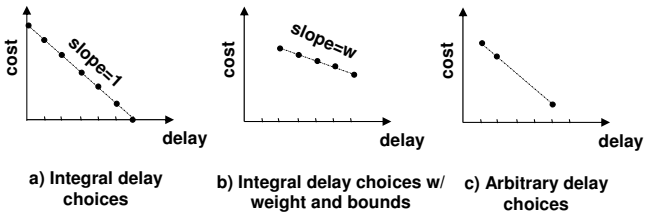

Fig. 2. Three cases of discrete implementation delay choices: a) Integral delay choices. b)Integral delay choices with weights and bounds. c) Arbitrary delay choices. Cases a and b are handled optimally.

B. Continuous, Integral and Arbitrary Delay Choices

An important property of the problem that fundamentally impacts its hardness, is the richness of delay choices available for implementing nodes. Ideally, one would like to be able to implement a node with any desired delay value. This is equivalent to assuming that possible implementation choices of a node offer *continuous* delay values, which is not feasible from a practical viewpoint. In reality, implementation choices of a node exhibit discrete delay values. We use the term *integral* delay choices to denote the case, where implementations with *consecutive* integer delays are available to realize a component. Figure 2.a illustrates this case. Black dots on the cost-delay plane denote possible implementations for a component. Note that integral delay choices implies that a unit relaxation in timing of a node would incur unit reduction in its cost.

A practical extension is to consider a specific cost-delay relation for each specific node type (weight), and consider bounds on minimum and maximum delay of possible implementations. We refer to this case as *weighted* and *bounded* integral delay choices (Figure 2.b). Finally, we use the term *arbitrary* delay choices to refer to implementations whose delays are not consecutive integers, and cannot be transformed into consecutive integers with scaling (Figure 2.c).

Throughout of this paper we assume that delay choices are integers. Note that proper scaling and rounding, according to the desired accuracy, can be used to represent delay numbers of implementation choices as integers (not necessarily consecutive). We temporarily focus on properties of the problem under the assumption of weighted and bounded integral delay choices, and later discuss extensions to handle real life component libraries.

C. CISP under Weighted and Bounded Integral Delays

We have previously studied CISP under weighted and bounded integral delay choices. We showed that a CISP instance can be transformed to a min-cost flow instance, and solved optimally using existing min-cost flow techniques. Specifically, we showed that when CISP is cast as an integer linear programming (ILP) problem, its dual problem formulates a classic min-cost flow problem [8], [11].

Figure 3.a illustrates the idea using the graph G of our working example introduced in Section II. Figure 3.b shows graph G', which is constructed by splitting nodes in G into two

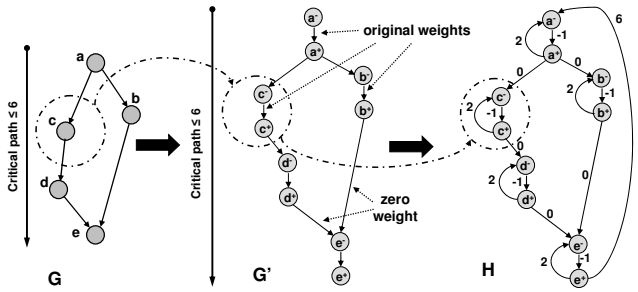

Fig. 3. a)Example implementation selection problem of Section II (G). b)Transformation to delay budgeting on edges of the graph G' c) Corresponding min-cost flow instance (H).

nodes with $-$ and $+$ signs. The edge connecting the split nodes in G' is called a split edge. For an arbitrary node v in G, we use the terms *corresponding nodes in* G' and *corresponding edge in* G' to refer to both nodes v^- and v^+, and the edge $v^- \rightarrow v^+$ in G', respectively. Non-split edges in G' have zero weight, i.e., their delay relaxation would not contribute to the overall cost function. The original CISP problem is transformed to selecting implementations for split edges of G'.

Graph H, shown in Figure 3.c, illustrates the corresponding dual min-cost flow instance. H has the same set of nodes as G'. It contains all edges of G', plus reverse edges added to split-edges and a reverse edge from the super primary output to the super primary input. Split and reverse edges refer to $v^- \rightarrow v^+$ and $v^+ \rightarrow v^-$ edges, respectively. Parameters of the min-cost flow in Graph H are determined according to Table I. Nodes with $+$ superscript in H supply flow, i.e. they are flow sources, and nodes with $-$ superscript demand flow, i.e. they are flow sinks. In our working example, all of flow supply nodes inject one unit of flow into H, and all of flow demand nodes sink one unit of flow from H. Edges of H are annotated with their cost for unit flow in the figure.

Table I summarizes the relationship between primal instance and the parameters of the dual min-cost flow instance. To the best of our knowledge, however, there is no intuitive relationship between a unit of flow in the dual problem and the original CISP problem. Interested readers are encouraged to refer to our previous publications and linear programming literature for more details [8], [11]–[14].

D. Minimum Cost Flow: Properties and Relation with CISP

We use the term *flow solution* to refer to any feasible (meeting flow conservation at all nodes) assignment of flow values to edges of H. We use the equivalent terms *the optimal or the min-cost flow solution* to refer to the flow solution that incurs the minimum cost. We briefly review the relation between the dual min-cost flow instance and the primal CISP instance, and relevant properties of min-cost flow. Readers are referred to [14] for details.

Any flow solution transforms the flow graph H into a residual graph H^*. In the residual graph, residual reverse edges (or simply residual edges) are added for any forward edge, with respect to flow direction. The cost of residual edges in H^* is complement of the cost of their corresponding forward edges. Residual edges have finite capacity equal to the amount of flow on the corresponding forward edge, which model flow cancellation on the forward edge by pushing flow along a residual edge [14].

primal problem (CISP)	dual problem (min-cost flow)
1-min. delay of node v	-(cost of split arc $v^- \rightarrow v^+$)
2-max. delay of node v	cost of reverse split arc $v^+ \rightarrow v^-$
3-timing constraint	cost of reverse arc $SO^+ \rightarrow SI^-$
4-weight (slope of cost-delay line) of node v	flow supply (demand) at node v^+ (v^-)
5-optimal delay of node i	difference in potentials (r or π) of v^- and v^+
6-optimal delay relaxation of node i	reduced cost of edge $v^- \rightarrow v^+$ in the min-cost residual graph
7-infeasible problem instance	infinite-capacity negative cost cycle

TABLE I
THE RELATIONSHIP BETWEEN PRIMAL AND DUAL PROBLEM.

Figure 4.a shows the optimal solution to the min-cost flow instance depicted in Figure 3.c in which nodes are annotated with their supply or demand, and edges are annotated with their flow in the min-cost solution. Edges that are not annotated have zero flow in the solution. Figure 4.b shows the corresponding residual graph H^* in which residual edges (dashed) exist for edges with non-zero flow. Edges of the residual graph (H^*) are annotated with their cost. The cost of residual edges is complement of their corresponding forward edges. For example, the cost of the residual edge $b^- \rightarrow b^+$ (not to be confused with split-edge $b^- \rightarrow b^+$) is -2.

The min-cost flow problem closely relates to the shortest path problem in graphs. To leverage the relation, the cost of edges in flow network should be interpreted as length, also known as distance, of edges in shortest path formulation. The length or distance of a path is defined as sum of length variables for edges on the path. The cost or length interpretation of an edge (path) annotation should be clear from the context. We denote the shortest directed path from SO^+ to any node i in the residual graph with π_i, where the subscript i can refer to any node (with superscript $+$ or $-$). π is well-defined in the residual graph of the optimal flow solution since it does not contain any negative cycle. Figure 4.c shows the residual graph of the optimal flow solution, H^*, in which, nodes are annotated with their shortest directed path from node SO^+.

The reduced cost of edge e_{ij}, an arbitrary edge in H^*, is defined as $c^\pi_{ij} = c_{ij} + \pi_i - \pi_j$, where c_{ij} is the cost of edge e_{ij}. Thus, the summation of edge reduced costs over any cycle in H^* is equal to the summation of edge costs over that cycle. The vectors π and c^π represent *node potentials* and *reduced costs* in network flow terminology, respectively. The complementary slackness conditions for all non-residual edges of the graph, outlined below, are *necessary and sufficient conditions* for optimality of a flow solution [14]:

$$c^\pi_{ij} > 0 \Rightarrow f_{ij} = 0 \quad (1)$$

$$u_{ij} > f_{ij} > 0 \Rightarrow c^\pi_{ij} = 0 \quad (2)$$

$$c^\pi_{ij} < 0 \Rightarrow f_{ij} = ucap_{ij} \quad (3)$$

where f_{ij} is flow on edge e_{ij}, and $ucap_{ij}$ is the flow capacity of the edge. Intuitively, reduced edge costs demonstrate suitability of edges for accepting flow. If the reduced cost of an edge is zero, then no statement can be made about its optimal flow value. In case of CISP, edges of H have infinite flow capacity, and hence, the condition $f_{ij} = ucap_{ij}$ cannot hold

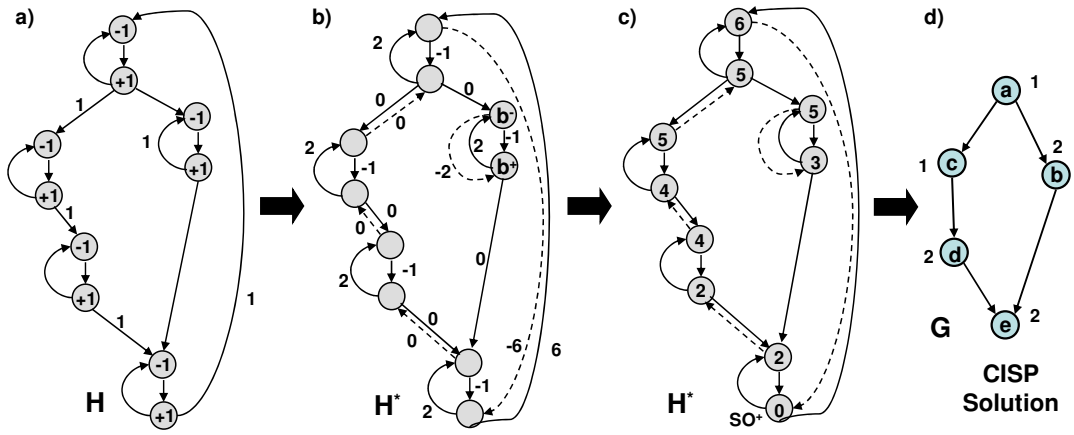

Fig. 4. a)Flow supply/demand (node annotations) and min-cost flow (edge annotations) solution. b)Residual graph (H^*) with edge cost annotations. c)Shortest path distance labels are assigned to nodes. d)Corresponding CISP solution.

for any edge. Therefore, edges of H cannot have a negative reduced cost in the residual graph of the optimal flow solution.

Reduced cost of an edge in the optimal flow solution translates to relaxation in its implementation delay (Figure 4.d) [8], [11]–[14]. For example, a reduced cost of 2 for a split-edge in H^* means that the corresponding node in G will be implemented with 2 units of delay relaxation. Equivalently, the difference between π_i and π_j in the residual graph of the optimal flow, gives the delay of the edge in G' (node in G) that corresponds to e_{ij} (Table I). The optimal CISP solution might not be unique, and a given min-cost flow solution translates to only one optimal CISP solution.

Finding the shortest SI^- to SO^+ path requires visiting the edges of the graph, excluding reverse split edges, in topological order.

We frequently refer to detection of negative cycles in graphs and determining shortest paths from a reference node. In H, any non-trivial negative cycle has to travel over the reverse timing constraint edge and hence, can be detected in $O(n)$, where n is the number of nodes. Note that in H, the number of edges grow linearly with nodes, because in practice components drive a limited number of fanouts. In the residual graph H^*, residual edges can create negative cycles that might not include the reverse timing constraint edge. Therefore, one would have to use a shortest-path algorithm that can handle negative edge lengths, such as FIFO-implementation of the label-correcting algorithm [14], which runs in $O(n^2)$ for practical designs.

IV. Incremental CISP

In order for designers to view the cost implications of ECOs applied to the design, a new CISP instance has to be solved for every ECO. Applying the full method requires iterative solution of min-cost flow on numerous subject graphs, which is prohibitive due to slow runtime of min-cost flow algorithms. Since only a small part of the design is typically updated during ECO, there is an opportunity to quickly update the existing CISP solution calculated on the original design graph, without going through the lengthy process of re-solving CISP from scratch. Note that having the original problem graph implies that the dual min-cost graph, containing characterizations of components library is available.

A. Primitive Incremental Operations

To formalize our notion of incremental modifications, we present a list of primitive graph manipulation operations. The primitive operations are selected such that they can be applied in sequence to transform one subject graph to another. The basic idea is to break down a designer's high-level ECO into a sequence of primitive incremental operations. Application of the sequence on the original design graph transforms it into the updated design. Our primitive incremental operations are:

1) **Node Delay Increment**: Delay of a node is incremented. The change might render the problem infeasible due to imperative violation of timing constraint.
2) **Node Delay Decrement**: Delay of a node is decremented. As a byproduct, node delay increment and decrement operations can also be used to incrementally change design timing constraint.
3) **Arc Insertion**: An edge is inserted between two existing nodes. The edge might render the problem infeasible if excessively long critical paths are created.
4) **Arc Deletion**: An existing edge is deleted from design graph existing nodes. It is assumed that the graph remains connected after arc deletion, since the notion of timing constraint has no practical significance in disconnected graphs.
5) **Node Weight Change**: Weight of a selected node is increased or decreased by $w > 0$. Weight change in conjunction with delay adjustment can be used to change a node in the design.

Primitive incremental operations that modify node information, i.e. node delay and weight manipulation, are local in nature. Edge manipulations, on the other hand, can potentially have global impacts on the design. An ECO is modeled as a sequence of unrelated primitive operations. Therefore, an ECO is not restricted to be local in scope although, locality is common in practical engineering change orders.

From a functional equivalence viewpoint, an ECO is equivalent to a collection, rather than a sequence, of primitive operations. That is, all of the necessary primitive operations have to be applied to make the updated design equivalent to the updated design under ECO. Sequential one-by-one application of primitive operations will create a sequence of intermediate

dummy designs all of which, except the last in the sequence, are not functionally equivalent to the updated design under ECO. Since our goal is to incrementally re-implement the design after every operation, all intermediate designs must be feasible to finally arrive at an equivalent design.

Our conjecture is that if the updated design under ECO is feasible, there exists a sequence of primitive operations that gives a sequence of feasible intermediate dummy designs. The intuitive guideline for construction of such a sequence would be to take *relaxing* operations first, and apply *restricting* operations last in the sequence. For example, imagine that a logic resynthesis ECO replaces a critical gate in the critical path with a slower gate that is not on the critical path. If the operation of incrementing the node's delay is applied first, the intermediate design would become infeasible. However, if necessary arcs are deleted, then new arcs are added, and finally the node's delay is increased, intermediate designs would be feasible. We leave "construction of operations' sequence for a given ECO" to the future work; and proceed with the assumption that if the updated design under ECO is feasible, then all intermediate designs for the given sequence of primitive operations are feasible.

B. Problem Statement

Recall that a CISP instance can be represented with the corresponding flow network H. The initial full solution to the problem delivers min-cost flow solution of H and shortest-path distances from SO^+ in H^*. Our target incremental CISP can be formally stated as follows:

Given the original problem instance G, its initial full solution and a sequence of primitive incremental operations, the objective is to update CISP solution in face of incremental operations such that the timing constraint of the perturbed design is met, and the updated solution remains optimal for the updated design. The perturbed design is constructed by sequential application of primitive operations.

V. Incremental Solution Update under ECO

A. Basic Idea

Complementary slackness conditions (equations 1-3) provide the necessary and sufficient optimality conditions for a given flow solution. Thus, for a flow solution to be optimal, its node potentials vector π has to satisfy the complementary slackness conditions. We refer to a node potential vector that satisfies those conditions, and hence refers to an optimal flow solution, as *valid*. The initial full solution ("from-scratch") gives a valid node potential vector, without any negative cost cycle in its residual graph.

The basic idea of our technique is to utilize complementary slackness conditions to 1)check optimality of the solution after application of each incremental primitive operation, and 2)incrementally update a subset of node potentials to revalidate the conditions in case they are violated after the operation. That is, if a primitive incremental operation applied to the subject graph does not violate the validity of the existing node potentials (i.e., complies with equations 1-3), the existing solution remains optimal for the perturbed graph. However,

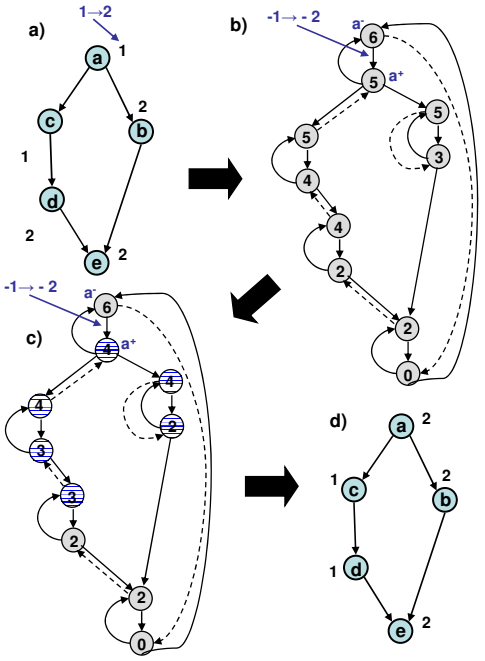

Fig. 5. a) Delay of node a is incremented in the original solution. b) corresponding change in H^* c) subgraph reachable from a^+ using edges with zero reduced cost, and their updated potentials d) updated solution

if the primitive operation violates the validity conditions, the flow solution and node potentials need to be updated to comply with equations 1-3, and to generate a new optimal solution.

After application of a primitive operation, our algorithm checks the validity of affected edges, and if needed, takes corrective measures to re-validate the node potentials. After application of the entire sequence of primitive operations and final revalidation of node potentials, the difference between the potential of the corresponding nodes in H^*, gives the delay of selected implementation for a node in G (Subsection III-D).

In CISP settings, all of the edges of graph H have infinite capacity. Thus, the case of equation 3 cannot happen in an optimal solution, and only the first two optimality conditions (equation 1 and equation 2) are applicable. In other words, in the residual graph of any optimal solution, no edge can have a negative reduced costs.

B. Delay Increment

Incrementing the minimum possible delay of node v in the design, decrements the cost of the corresponding split-edge $e_{v^-v^+}$ in H^*, and its reduced cost $c^{\pi}_{v^-v^+}$. The split-edge has infinite flow capacity and thus, if the operation creates a negative cost cycle in the min-cost graph H, then the problem would be infeasible, i.e., timing constraint will be violated even if all nodes are implemented with their fastest delay choices. Detection of such negative cycles can be done in $O(n_p)$, where n_p is the maximum number of nodes on a $SI^- \rightarrow SO^+$ path, since one would only have to track the impact of the delay change on the shortest $SI^- \rightarrow SO^+$ path.

Assuming problem feasibility, If $c^{\pi}_{v^-v^+} > 0$ before decrementing cost of $e_{v^-v^+}$, this incremental operation keeps the reduced cost of the edge non-negative. This does not violate the validity of node potentials on $e_{v^-v^+}$ and no action is

required. In the design space, it implies that enough slack is already assigned to the node to tolerate unit delay increment. However, if $c^{\pi}_{v^- v^+} = 0$ before the operation (i.e., node has no slack), post-operation node potentials would change the updated reduced cost of the edge $e_{v^- v^+}$ to -1, which makes the existing solution invalid. We try to revalidate the solution by decrementing (incrementing) node potentials for a selected group of nodes including $v^+(v^-)$.

Let us only consider edges of the graph H^* that have zero reduced cost before the operation. If there is a path from v^+ to v^- that only goes through edges with zero reduced cost, the cycle formed by this path and *updated* $e_{v^- v^+}$ has total negative cost. Therefore, we can improve the flow solution by pushing flow along this cycle. Assuming problem feasibility, such a v^+ to v^- path has to go through some residual edges whose capacity is finite.

Pushing flow along this cycle ultimately fills at least one of the residual edges to capacity, and cancels the cycle. Continuing the cycle cancellation process, ultimately, a *cut* in the subgraph including only zero reduced cost edges will be created. Then, the solution can be revalidated by decrementing node potentials, π, for all nodes in the partition containing v^+. This correction makes $c^{\pi}_{v^- v^+} = 0$. Furthermore, it gives a valid node potential because any residual edge with negative reduced length is filled to its capacity (Equation 3), i.e., the corresponding forward edge has zero flow and positive reduced length (Equation 1).

In the worst case, there are $O(E)$ edges with zero reduced cost, which translates to $O(n)$ for practical design problems. Detection of a negative cycle requires $O(n^2)$, and updating the potentials in one partition would run in $O(n)$. Making the practical assumption that the number of required cycle cancellation is bounded by a constant, the entire process would have $O(n^2)$ time complexity. In practice, however, the edges with zero reduced cost are considerably smaller in number than n, and are sparsely distributed in H^*. Therefore, the process would have a much lower complexity in the average case.

Figure 5.a illustrates our working example in which, the delay of node a is incremented to 2. Figure 5.b illustrates the corresponding graph H^*. Nodes are annotated with their potentials, and nodes a^- and a^+ are incident to the edge whose reduced cost is decremented. Nodes in H^* that are reachable from a^+ using only edges with zero reduced cost are filled with dotted pattern in Figure 5.c, in which nodes are annotated with their updated potentials. Finally, Figure 5.d illustrates the updated solution to the given CISP instance.

C. Delay Decrement

Decrementing the delay of a node in G by unit, increments the cost of corresponding edge $e_{v^- v^+}$ in H^*. Since the initial solution is valid, $c^{\pi}_{v^- v^+} \geq 0$ before the operation. The primitive operation increments the reduced cost of the edge. If $c^{\pi}_{v^- v^+} > 0$, or if $c^{\pi}_{v^- v^+} = 0$ and $f_{v^- v^+} = 0$, before the operation, the updated node potentials are valid. From a design point of view, this is equivalent to decreasing the delay of a node that can favorably utilize slack: additional slack created by decrementing node delay can be left at that node.

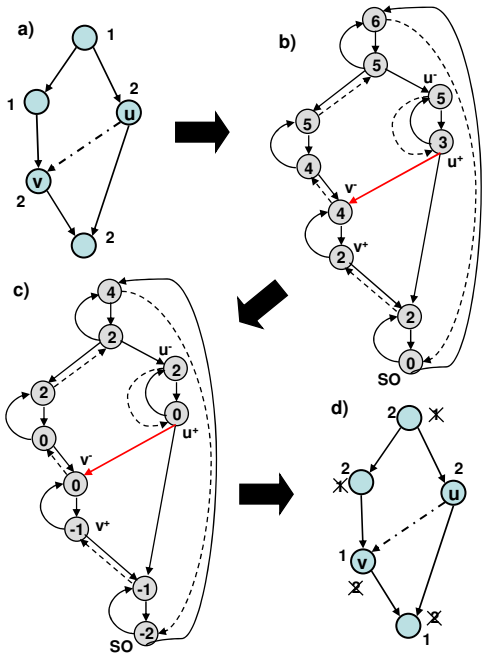

Fig. 6. a)Original CISP solution with inserted edge b)Corresponding residual graph (H^*) with node potential annotations c)Potentials updated with shortest path from v^- d)Updated CISP solution

However, if $c^{\pi}_{v^- v^+} = 0$ and $f_{v^- v^+} \neq 0$ before the primitive operation, Equation 1 will be violated after applying the primitive operation. In the design domain, this means that the slack introduced at the node should be distributed to other nodes to increase its utilization, and preserve optimality of the solution. To revalidate the solution, $f_{v^- v^+}$ units of flow have to be rerouted from v^- to v^+ along other edges to cancel the flow on $e_{v^- v^+}$. If flow rerouting only uses edges with zero reduced cost, total cost of flow would not be affected.

Similar to the delay increment case, we try to push flow along zero cost paths from v^- to v^+ in the residual graph. If this is possible, the updated node potentials will be valid without any change to the overall flow cost. Otherwise, pushing flow will eliminate some residual zero cost edge, and will create a cut in the subgraph created by zero edges. Nodes that are at distance zero from v^- (i.e., accessible via zero reduced cost edges), including v^-, form a partition, whose potential should be incremented. Similar arguments as delay increment case apply toward correctness and complexity.

D. Arc Insertion

The primitive operation of inserting an arc between u and v in G, corresponds to adding an arc with zero cost from u^+ to v^- in H^*. If this arc creates either a cycle or a path longer than the timing constraint in G, there will be an infinite-capacity negative cost cycle in H^*, and the problem will be infeasible. Existence of a cycle or a long-enough path in G can be detected in $O(n)$ by topological traversal of the design graph (Subsection III-D).

Otherwise, if $\pi_{u^+} \geq \pi_{v^-}$, the new arc has non-negative reduced cost and the initial solution remains valid. In the design domain, this case implies that the new edge does not create any new critical path and hence, existing implementation selection solution is still valid and optimal. The

challenging case occurs if no infinite-capacity negative cost cycle is created and $\pi_{u^+} < \pi_{v^-}$. This would violate the last optimality condition (equation 3) because edges of H, including the newly inserted edge, have infinite flow capacity.

In this case, we have to assign a new set of potential variables to nodes to revalidate optimality conditions. Thus, one needs to first eliminate finite-capacity negative cycles that might be created by insertion of the new edge in H^*. If shortest path from v^- to u^+ is negative, it points to a cost-reducing cycle from v^- to u^+ and back to v^- using the newly inserted edge. Assuming that the cycle has finite capacity, pushing flow along it would improve the min-cost flow solution. Detection of such a cycle would require a single-source (from v^-) label-correcting shortest path algorithm that has the complexity of $O(n^2)$ in the worst case (Subsection III-D).

Note that the capacity of such paths cannot be infinite (otherwise the timing constraint is always violated and problem is infeasible), so they can be eliminated by pushing large-enough flow. After eliminating each negative cost cycle of finite capacity, the shortest path distance might have to be updated because some residual edges might be eliminated after pushing flow. Injection of flow along finite-capacity negative cycles has to continue until the shortest path from v^- to u^+ is non-negative. In the worst case, each cycle elimination corrects one unit of flow, and the process has to be repeated W times, where $W = \sum w_i$ is the total flow supply of the graph. This worst case scenario, however, is very unlikely to occur and on average, cycle elimination converges very quickly.

After eliminating all negative v^- to u^+ paths, we set the potential of an arbitrary node i, π_i, equal to the shortest path distance from v^- to i. The following lemma proves that the new node potentials give a valid solution that complies with equations 1-3 and hence, is optimal:

Lemma 1: Shortest path distances from a reference node in the residual graph H^* of a min-cost flow, give a valid node potential vector.

Proof: Let $dist_i$ denote the shortest path of node i from a reference node in the graph. For any arbitrary edge e_{ij} in H^*, the shortest path property implies that $dist_j \leq dist_i + c_{ij}$. Assignment of distance labels as node potentials means that $\pi_j \leq \pi_i + c_{ij}$ or equivalently $c_{ij} + \pi_i - \pi_j = c^{\pi}_{ij} \geq 0$. In addition, if there is non-zero flow going over edge e_{ij} in H^*, its reverse edge e_{ji} exists in the graph with complement cost. Therefore, $c^{\pi}_{ij} = -c^{\pi}_{ji} \geq 0$ and $c^{\pi}_{ji} \geq 0$, which implies that $c^{\pi}_{ij} = 0$. Therefore, the resulting node potential vector complies with conditions 1-3 and is optimal. ■

Figure 6 illustrates the procedure using the working example we introduced in Section II. In this example, an edge is inserted between nodes u and v in graph G. Note that the inserted edge violates the timing constraint on the current implementation selection solution. Figure 6.b shows the corresponding residual graph H^* with the inserted edge in which, nodes are annotated with their potentials before edge insertion operation. In Figure 6.c shortest path distances from v^- are used to update node potentials. Since the shortest distance from v^- to u^+ is zero, there is no need to send flow from v^- to u^+. If this would not be the case, we had to send flow and update shortest paths until the potential of node u^+ becomes non-negative. Finally, Figure 6.d shows the updated solution of the CISP instance.

E. Arc Deletion

Assume that we are to delete arc e_{ij} in the design graph G, which corresponds to the edge $e_{i^+j^-}$ in H^*. We assume deleting this arc from H^* maintains its connectivity, because 1)disconnected graphs are of limited interest in practice, and 2)connected components of a disconnected graph are subject to the same analysis for connected graphs. If there is no flow on $e_{i^+j^-}$ in the optimal min-cost flow solution, deleting $e_{i^+j^-}$ does not violate any flow conservation constraint in H^*. Hence, the existing solution is still valid, and does not need to be updated. From a design perspective, this case corresponds to deleting a net that is not critical after slack assignment.

However if $f_{i^+j^-} \neq 0$, edge $e_{i^+j^-}$ would correspond to a critical net in the design domain whose deletion, might create additional slack for distribution. In this case, deletion of $e_{i^+j^-}$ will violate flow conservation constraints at nodes i^+ and j^- in H^*. Before deleting the edge, hence, one has to eliminate the flow on edge $e_{i^+j^-}$ by rerouting its flow along other edges of H^*, while incurring minimum cost. To accomplish this, $f_{i^+j^-}$ units of flow has to be pushed along the shortest path from node i^+ to node j^- excluding $e_{i^+j^-}$. If the capacity of the shortest path is less than $f_{i^+j^-}$, successive shortest paths should be found to collectively push $f_{i^+j^-}$ units of flow. After rerouting $f_{i^+j^-}$ units of flow, edge $e_{i^+j^-}$ can be safely removed from the graph without violating flow conservation constraints.

From a complexity viewpoint, handling arc deletion is subject to the same analysis as arc insertion. That is, determining shortest paths from i^+ to j^- would run in $O(n^2)$ for practical design graphs, and the process will have to be repeated $f_{i^+j^-}$ times, in the worst case. In the average case, however, rerouting flow converges very quickly without going through very many iterations.

The process of rerouting $f_{i^+j^-}$ units of flow from i^+ to j^- is likely to change the residual graph. Therefore, node potentials might have to be updated to comply with the optimality conditions (equations 1-3). Lemma 1 proves that distance from a reference node makes an optimal node potential vector. Thus, we update potential of each node to the shortest path from i^+ to the node, simply because such shortest path vector is already calculated to push flow from i^+ to j^-, and it would save on computation time.

An example is depicted in Figure 7.a in which, the edge from node i to j is going to be deleted from G. Figure 7.b shows the change in the corresponding residual graph H^*, where a unit of flow already travels along the edge from i^+ to j^-. Nodes are annotated with their potential values that was valid before deleting the edge. To reroute the unit of flow, shortest path from node i^+ to j^- is calculated, and a unit flow is pushed along the path. Figure 7.c shows the case in which, nodes are annotated with their shortest distance from i^+. Note that all primary outputs created after edge deletion are connected to a new super output node (SO).

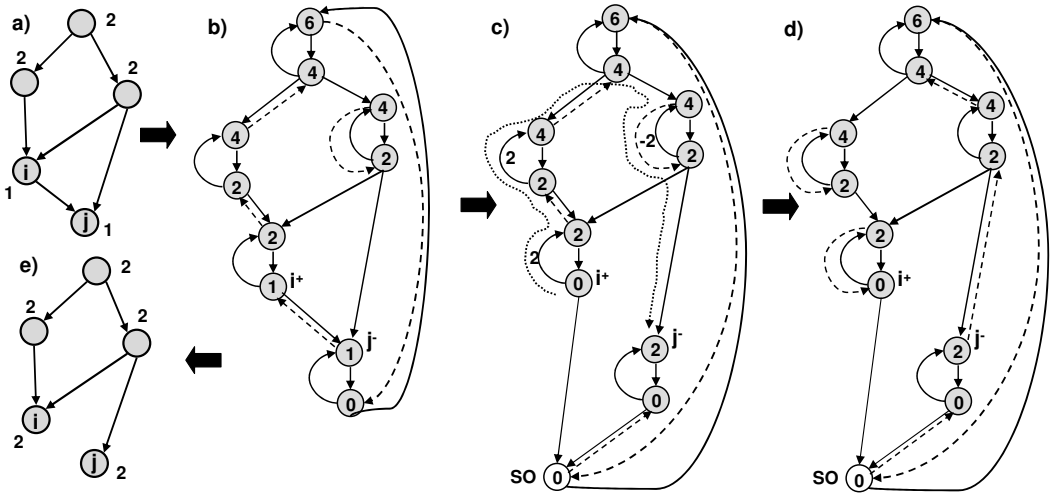

Fig. 7. a) Original CISP solution with deleted edge b) corresponding residual graph (H^*) with node potential annotations. One unit of flow travels along edge i^+j^- c) shortest path from i^+ is used to reroute the flow from i^+ to j^- after edge deletion. Nodes are annotated with shortest distance from i^+ d) updated node potentials in the updated residual graph after pushing flow e) Updated CISP solution

The numbers on edges along the path show the edge costs (for unit flow) that are assigned according to the transformation discussed in Subsection III-A. The edges that do not have any number annotation along the path have zero cost in flow formulation. Figure 7.d shows the residual graph after rerouting the unit flow in which, nodes are annotated with their shortest distance from i^+ in the updated graph. Distances happen to be the same in this example, however, that is not necessarily the case. Finally, 7.e shows the updates CISP solution after deleting the edge.

F. Weight Change

Node weights represent relative component cost reduction with unit increase in delay. From a design perspective, changes to node weights relate to updating cost-delay characterization of components in the library. Weight of node i in G, determines the supply and demand of nodes i^+ and i^- in H. Subsequently, increasing weight of node i by w in G, creates additional w units of flow supply at i^+, and additional w units of flow demand at i^- in H^*. Incremental handling of this primitive move should send w units of flow along the shortest path from i^+ to i^- in H^* to update the min-cost flow solution. If the capacity of the shortest path is less than w, successive shortest paths need to be found to collectively handle w units of flow. A path might have finite capacity only if it contains a residual edge. All non-residual edges in the graph have infinite capacity.

Similarly, decreasing the weight of node decreases the supply and demand by w. Since the initial solution is optimal, we need to send w units of flow from i^- to i^+ to cancel excessive flow supply and demand. This is achieved by augmenting w units of flow along the shortest path (or a collection of successive shortest paths). After updating the flow solution, node potentials will be updated by reseting them to their new shortest path to/from a fixed node (Lemma 1). The complexity analysis arguments of edge insertion/deletion apply here as well. That is, in the worst case the process would require w repetitions of $O(n^2)$ label-correcting shortest path algorithm. Similar to previous cases, however, in the average expected case flow pushing process does not require w steps, and converges much faster.

Our working example of Subsection II is depicted in Figure 8 in which, all nodes are initially assumed to have equal unit weight. Nodes are assumed to have two possible implementations: unit delay that incurs two units of cost, and 2 unit of delay that incurs unit cost. Thus, the optimal solution of Figure 8.a has total cost of 7. Let us assume that the incremental operation changes the weight of node c from 1 to 2, which is equivalent to assigning cost of 3 for its implementation with unit delay, and cost of 1 for its implementation with two units of delay. In residual graph H^* in Figure 8.b, the increase in weight increments the flow supply at c^+ and flow demand at c^-. Therefore, the existing solution no longer meets the flow conservation constraint and becomes invalid.

We revalidate the solution by pushing the excess flow (unit in this case) from c^+ to c^- over the shortest path. That is, shortest distance labels in H^* are updated to denote the distance from node c^+ (Figure 8.c). Subsequently, a unit flow is pushed along the shortest path, which is shown as a dashed line in Figure 8.c. The distance labels from c^+ are updated, which readily give the updated node potentials and CISP solution (Figure 8.d). Note that the new solution still total cost of 7, while the original solution after weight change had total cost of 8.

VI. Practical Extensions and Limitations

We assumed that component implementations have consecutive integer delays, and their cost-delay relation is linear (Figure 2.b). We proceed to discuss extensions to this basic model to address practical library constraints. We also discuss practical limitations of our technique.

A. Arbitrary Delay Choices

It has been shown that implementation selection for nodes of a DAG under arbitrary delay choices in NP-Hard [15]. Not only our mathematical analysis is useful from the viewpoint

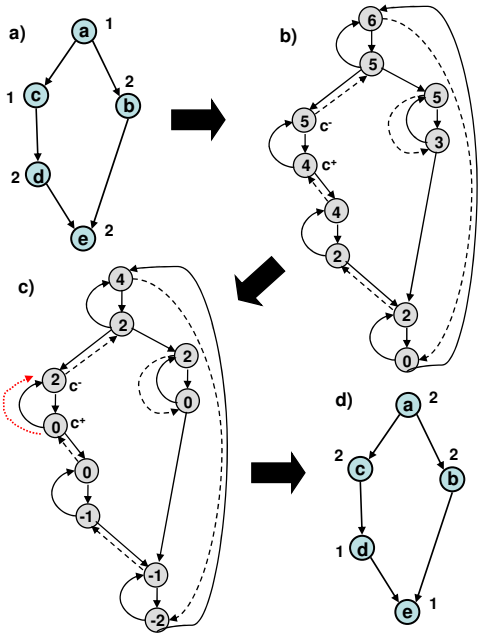

Fig. 8. a)The example of Section II, in which weight of node c is increas to 2. The solution would have total cost of 8 after the incremental operatic b)The corresponding residual graph with node potential annotations. c)A u of flow is pushed from c^+ to c^- along the shortest path and node potentia are updated. d)Updated CISP solution with optimal total cost of 7.

of *optimality gap* measurement, it also can help to develc quality heuristics under arbitrary delay choices (Figure 2.c)

We tackle CISP under arbitrary delay choices by temporari relaxing delay choices to weighted and bounded integral del choices. The temporary relaxation will be corrected by a fin legalization step. We first assume that all consecutive integral delay choices are available to implement the design (case of Figure 2.b). After solving the relaxed problem, a heuristic legalization is carried out to replace non-existent imaginary implementations with physical implementations in the library. An example legalization heuristic is delay round down, which conservatively replaces an imaginary implementation in the solution, with the closest (in terms of delay) slower physical implementation. Empirical results show that the procedure yields near-optimal results, because in practice, a relatively small number of components end up being mapped to imaginary implementations [16].

B. Convex Cost Functions

Many cost functions, such as active and leakage energy dissipation, do not have linear relationship with component delay. Our approach can be extended to handle convex cost functions with no quality loss, in case integral delay choices are available. If consecutive integral delay choices are not available, the heuristics explained in Subsection VI-A can be utilized to handle the situation.

In presence of arbitrary (or integral) delay choices, the convex cost-delay relationship can be viewed as a collection of line segments (Figure 9). Note that only the end point of line segments are available implementations, due to intrinsic discreteness of the component library. If the original cost function is convex, the slopes of line segments, which are negative, do not decrease with growth of delay.

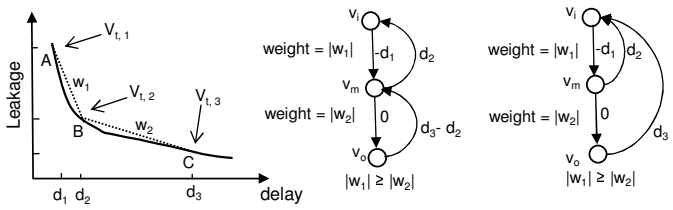

Fig. 9. Extension of basic model to handle convex cost functions and arbitrarily discrete delay choices.

Figure 9 shows the situation for an example transformation. In this example, three different implementations, A, B and C, are available for a node. The cost of each implementation is a convex function of its delay. The graphs on the right illustrate the structure corresponding to a component, when design graph G is transformed to min-cost flow graph H. They replace the split-edge transformation in case of previously-discussed linear cost functions. Intuitively, the transformation models each line segment with a node (with appropriate delay lower and upper bounds). Convexity of cost function ensures that delay relaxation is first allocated to the node corresponding to the first line segment, and so forth. Interested readers are referred to [16] for more detailed discussion on the transformation.

C. Complications and Limitations

In our target design model, nodes incur delay but edges have zero delay. One could insert dummy delay nodes on the edges, with one or multiple implementation choices, to model interconnect delay in gate-level netlists. To model hyper-edges, which are the proper abstraction for modeling gate interconnects, one would have to decompose the hyper-edge into *shared* and *dedicated* edges. For example a hyper-edge connecting a gate to its two fanouts, can be modeled with a shared edge to a dummy node, which is connected to the two fanouts using dedicated edges.

This abstraction, however, has its own complications and inaccuracies. For example, the delay of the shared and dedicated segments would be decoupled. Additionally, one would have to know physical placement of gates to accurately decompose the hyper-edge into shared and dedicated segments with specified delay numbers. Since our framework is designed for handling ECO before placement, interconnect delays would have to be estimated at high-level, either without having placement information or within an iterative design flow.

VII. Empirical Validation

We applied our technique to problem of gate-level threshold voltage (V_t) assignment for leakage optimization. Two implementations are available for gates that correspond to fabricating them with either high or low threshold voltage. Low V_t implementation results in a faster but leakier gate compared to high V_t. The relation between leakage and delay is convex, for which we utilize our practical extensions.

We used Goblin [17], an open source C++ library for graph optimization and network programming, to implement

the aforementioned graph data structures, min-cost flow based full CISP solver, and our incremental algorithm. Goblin offers a variety of graph algorithms, including label correcting shortest-path and cycle-canceling min-cost flow algorithms that were used in our study. We integrated Goblin library routines with the SIS logic optimization framework.

After temporary relaxation of arbitrary delay choices to consecutive integers and solving using our CISP technique, we round down gate delays to arrive at the fastest gate implementation that exists in the library. Gates in gen2.lib library are characterized for input capacitance, delay and leakage under 0.4 and 0.5 volts threshold voltage. The values are normalized with respect to an inverter in the library. Selected circuits from MCNC benchmark suite are mapped to gen2.lib library using SIS "map" command. We developed our own timing analyzer and leakage estimator, which look up library characterization for gate leakage, input capacitance and intrinsic delay parameters. For timing analysis, load dependent delay model is used in which, the delay of a gate is estimated as its intrinsic delay plus its load dependency factor times load capacitance: $delay_g = delay_{intrinsic} + C_{load} * slope_g$

Initially, all gates in the circuit are mapped to low V_t of 0.4 volt to capture initial leakage and timing constraint. Subsequently, the library was expanded to contain two implementation choices, corresponding to high or low V_t, for each gate. Timing relaxation was not allowed, i.e., the timing constraint of each circuit is the same as its critical path when all gates are mapped to low V_t. After assigning low and high threshold voltages to gates, another timing analysis is performed to assure the validity of results. In all cases, our algorithm met the required timing constraint. Gate delay, leakage, and input capacitance data are borrowed from the study performed by Khandelwal and Srivastava [18].

A. Average ECO Workload

The amount of computation workload that is required to handle an ECO depends on the amount of perturbation introduced in the netlist by that ECO. Consequently, the amount of computation for handling incremental updates need to be matched to expected perturbations [19]. In the extreme case, if the perturbed netlist is completely different from the original design, invoking full algorithm might be a better choice than running the incremental engine.

We define the notion of "average" ECO perturbation in order to evaluate our algorithm effectiveness. We assume that alterations imposed by an average ECO can be replicated by a sequence of 50 primitive incremental operations. It is practically hard to characterize the amount of perturbation that an average ECO inflicts on a design. We chose this number because we observed that 50 incremental operations (e.g., delay change, arc insertion, weight change, or arc deletion) can model substantial design modification.

Figure 10 illustrates the idea using a simple example. Let us assume that the left netlist undergoes an ECO that yields the right netlist. That is, gate a is replaced with gate a', characterization of the minimum delay possible for gate b is changed in the library, and finally, edge e is replaced with edge f. A sequence of five incremental operations models the example transformation. Specifically, a *weight-change* followed by *delay-change* operation replaces gate a with a'. Another *delay-change* operation updates the minimum delay for gate b. An *arc-removal* followed by *arc-insertion* replaces edge e with f. Our assumption is that the operations are applied in a feasible sequence, i.e., all intermediate designs created after application of primitive operations are feasible, if and only if the design under ECO is feasible.

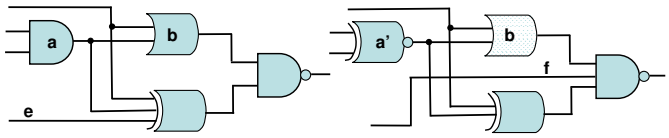

Fig. 10. Example of transforming one netlist to another.

We run the full algorithm to measure its runtime on the original netlist, and to generate an initial optimal solution. Then, 25 primitive operation types (e.g., delay change or arc insertion) and their associated parameters (e.g., location to insert the arc) are generated randomly. Reverse operations are then applied in reverse order to cancel the impact of initial 25 operations, and to arrive at the initial netlist with known solution. For example, if operation 20 happens to be *weight-change* for a specific randomly chosen node, operation 31 will be *weight-change* for the same node with negative sign. The rationale is that if the algorithm yields the same flow value after many runs, then we can empirically validate that solution optimality is preserved throughout the process. Also, the runtime of the full CISP solver for the initial and final designs are the same. Note that incremental operations are handled right away ignoring that each operation is going to be canceled by another operation in the future. Operations that would render the solution infeasible are not considered.

B. Experimental Results

Table II summarizes our experimental results. For each circuit, original leakage (all gates assigned to low V_t) and optimized leakage after running both full and incremental algorithms are reported in columns $3-5$. Column 6 ($improv\%$) shows the leakage improvement of the full algorithm compared to original leaky circuit. Column 7 ($error\%$) compares the leakage of the circuits optimized using full algorithm with circuits after undergoing incremental changes. The difference is zero in all cases except for `C1355`, where we have a negligible 0.23% improvement in leakage. Experimental results validate the effectiveness of our incremental algorithm in maintaining the quality of full algorithm.

The last three columns report the runtime of 1)one call to full solver to handle an average ECO, 2) fifty successive calls to our incremental handling method after each primitive operation, and 3) the speedup gained by our approach. Recall that an average ECO is assumed to be equivalent to 50 incremental operations. The runtimes were recorded over 40 runs and the average numbers are reported. Our experiments show that the incremental algorithm is about 283 times faster than running the full algorithm, while delivering the same quality results. The runtime improvement is more significant

Circuit	Cell count	Leakage (normalized)					Runtime (sec)		
		original	full	incr.	improv (%)	error (%)	full	incr.	speedup
too_large	456	2413.5	408.5	408.5	83.1	0	4.61	0.11	40.7
C2670	505	4384	808.2	808.2	81.6	0	9.15	0.29	31.9
C1355	510	2913.4	814.7	812.8	72.0	0.23	5.97	0.26	22.9
pair	948	8502.2	1411.6	1411.6	83.4	0	28.35	0.24	119.7
alu4	1579	7398.2	1164.6	1164.6	84.3	0	64.23	0.48	132.9
seq	2016	10558.7	1587.8	1587.8	85.0	0	139	0.43	320.6
apex2	2120	10980.8	1658.2	1658.2	84.9	0	162.35	0.51	316.6
des	2718	18737.2	2803.9	2803.9	85.0	0	284.99	0.57	502.6
spla	4603	24127.2	3625.7	3625.7	85.0	0	824.07	1.52	543.8
ex1010	5045	21090.7	3187.5	3187.5	84.9	0	937.59	5.03	186.3
pdc	5812	31294.4	4669.7	4669.7	85.1	0	1582.32	1.75	901.7
Average	2392				83.1	0.02			283.6

TABLE II
RUNTIME AND LEAKAGE COMPARISON BETWEEN FULL AND INCREMENTAL IMPLEMENTATION SELECTION.

for larger circuits, and hence, the gap is expected to widen for more complex benchmarks.

Mathematical properties of our method prove that our incremental technique maintains the optimality of the min-cost flow solution in subject graph. Under consecutive integral delay choices assumption, the min-cost flow directly maps to a component implementation selection and hence, the optimality of the solution in hardware domain is guaranteed. In case of practical arbitrary delay choices, min-cost flow solution delivers high quality, but not necessarily optimal, solutions.

As a result, it is theoretically possible that two different solutions have the same flow cost in subject graph, but incur different leakage when mapped to hardware domain. This occurs for circuit `C1355`, which is a highly interconnected circuit with many critical paths. In this case, successive perturbations allowed minor improvement of the leakage results, although both full and incremental algorithms arrive at solutions with the same amount of flow.

VIII. CONCLUSIONS

We presented an effective methodology for incrementally updating an implementation selection solution for a netlist that goes through engineering changes. Utilizing mathematical properties of min-cost flows, our technique guarantees to meet the timing constraint, and to maintain the optimality of the solution. We presented extensions to our technique that enable its application to practical engineering problems. Experiments with gate-level threshold voltage assignment show more than 2 orders of magnitude runtime improvement compared to re-executing an optimal from-scratch solver for each engineering change, while delivering the same results.

REFERENCES

[1] O. Coudert, J. Cong, S. Malik, M. Sarrafzadeh. "Incremental CAD". In *International Conference on Computer-Aided Design*, pages 236–243, 2000.

[2] Shiyan Hu, Mahesh Ketkar, and Jiang Hu. Gate sizing for cell library-based designs. In *Design Automation Conference*, pages 847–852, 2007.

[3] L. Singhal, E. Bozorgzadeh. "Fast Timing Closure through Interconnect Criticality Driven Delay Relaxation". In *International Conference on Computer-Aided Design*, pages 792–797, 2005.

[4] A. Srivastava. "Simultaneous Vt selection and assignment for leakage optimization". In *International Symposium on Low Power Electronics and Design*, pages 146–151, 2003.

[5] Chun-Gi Lyuh and Taewhan Kim. High-level synthesis for low power based on network flow method. *IEEE Transactions on Very Large Scale Integration Systems*, 11(3):364–375, 2003.

[6] R. Nair, C. Berman, P. Hauge and E. Yoffa. "Generation of Performance Constraints for Layout". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8:860–874, August 1989.

[7] E. Bozorgzadeh, S. Ghiasi, A. Takahashi and M. Sarrafzadeh. "Optimal Integer Delay Budgeting on Directed Acyclic Graphs". In *Design Automation Conference*, June 2003.

[8] S. Ghiasi, E. Bozorgzadeh, S. Choudhury, M. Sarrafzadeh. "A Unified Theory of Timing Budget Management". In *IEEE/ACM International Conference on Computer-Aided Design*, pages 653–659, 2004.

[9] Huan Ren and Shantanu Dutt. Algorithms for simultaneous consideration of multiple physical synthesis transforms for timing closure. In *International Conference on Computer-Aided Design*, pages 93–100, 2008.

[10] W.N. Li, A. Lim, P. Agrawal, S. Sahni. "On the Circuit Implementation Problem". In *Proceedings of the 29th ACM/IEEE Design Automation Conference*, pages 478–483, 1992.

[11] S. Ghiasi, E. Bozorgzadeh, P-K. Huang, R. Jafari, M. Sarrafzadeh. "A Unified Theory of Timing Budget Management". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(11):2364–2375, November 2006.

[12] G.B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1963.

[13] L.A. Wolsey. *Integer Programming*. Wiley-Interscience Publisher, John Willey & Sons Inc., 1998.

[14] R. Ahuja, T. Magnanti, J. Orlin. *"Network Flows: Theory, Algorithms, and Applications"*. Prentice Hall, 1993.

[15] S. Ghiasi, H.J. Moon, A. Nahapetian, M. Sarrafzadeh. "Collaborative and Reconfigurable Object Tracking". *Kluwer Journal of Supercomputing*, 30(3):213–238, December 2004.

[16] Soheil Ghiasi. An effective combinatorial algorithm for gate-level threshold voltage assignment. *Journal of Low Power Electronics*, 2(3):365–377, 2006.

[17] Goblin: A graph object library for network programming problems. available online at http://www.math.uni-augsburg.de/fremuth/goblin.html.

[18] V. Khandelwal, A. Davoodi, A. Srivastava. "Simultaneous Vt Selection and Assignment for Leakage Optimization". *IEEE Transactions on Very Large Scale Integration Systems*, 13(6):762– 765, 2005.

[19] A.B. Kahng, S. Mantik. "On Mismatches Between Incremental Optimizers and Instance Perturbations in Physical Design Tools". In *International Conference on Computer-Aided Design*, pages 17–21, 2000.

Soheil Ghiasi Soheil Ghiasi received his B.S. from Sharif University of Technology, Iran in 1998, and his M.S. and Ph.D. in Computer Science from University of California, Los Angeles in 2002 and 2004, respectively. He received the Harry M. Showman Prize for excellence in research communication from UCLA college of engineering in 2004. Currently, he is an assistant professor of electrical and computer engineering at the University of California, Davis. His interests include different aspects of embedded systems including system-level design automation.