Incremental Component Implementation Selection: Enabling ECO in Compositional System Synthesis

Soheil Ghiasi
Department of Electrical and Computer Engineering
University of California, Davis
soheil@ece.ucdavis.edu

ABSTRACT

The component implementation selection problem (CISP) is to select the appropriate implementation for components of a design, such that the timing constraint is met and some global design objective is optimized. CISP is a generic problem that implicitly or explicitly appears in many stages of CAD flow. In this paper, we present a methodology for quick and efficient updating of CISP solutions in face of incremental engineering changes. For a commonly-used formulation, we discuss necessary and sufficient conditions for optimality of a CISP solution based on which, we develop an algorithm that maintains both validity and optimality of a solution subject to incremental changes. We implemented our approach to incrementally update the threshold voltage assignment solution for a netlist going through engineering changes. On average, our method ran over 300 times faster than the "from-scratch" solver, while delivering the same results.

1. INTRODUCTION

In a typical development scenario, a design goes through many incremental changes before its development process is finished. Incremental changes often do not demand a fresh, computationally-intensive and "from-scratch" solution to CAD problems, because, the solution determined in previous iterations can usually be quickly and efficiently updated to handle the perturbations [4, 3, 1, 2].

The problem of component implementation selection is a generic formulation that is either implicitly or explicitly solved in different stages of library-based CAD flow. Essentially, this problem tries 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 (e.g., energy dissipation) is minimized [5].

In this paper, we present an incremental method that handles engineering changes, and quickly updates the solution of implementation selection problem on a directed-acyclic graph. While guaranteeing to meet the timing constraint, our technique maintains the optimality of the solution. We present necessary and sufficient conditions for quick validation of a candidate solution, and develop guidelines to quickly and optimally handle primitive incremental operations such as insertion or deletion of a net in the design.

To validate the theoretical contributions of this paper, we applied our technique to the problem of gate-level threshold voltage (V_t) assignment for leakage optimization. Assuming that an engineering change is composed of about 50 primitive netlist-changing operations (Subsection 4.1), our incremental solution updating approach was on average 303.3 times faster than re-executing the solver to get a fresh V_t assignment, while delivering the same results.

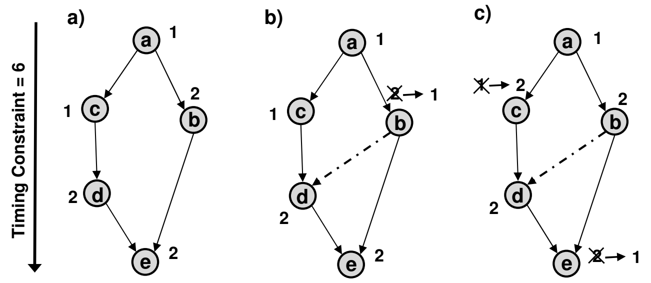

Figure 1: a) Example optimal solution. b) After inserting an arc, greedy selection correction is sub-optimal. c) Optimal solution. Note that it does not modify the two end points in this example.

2. MOTIVATING EXAMPLE

Figure 1 depicts a simple example to illustrate the idea of incremental CISP. The figure shows an example directed acyclic graph that models an application. Nodes and edges in the graph represent tasks (computations) and dependencies, respectively. All nodes are assumed to have two possible implementations. Specifically, 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 $b \to d$ is added to the DAG. Insertion of the new arc creates the path $a \to b \to d \to e$ that takes 7 unit of time and hence, 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 deliver incremental algorithms for legalizing the solution (i.e., meeting the timing) while maintaining its optimality, under incremental updates (ECO) to the design. We discuss theoretical foundations of the prob-

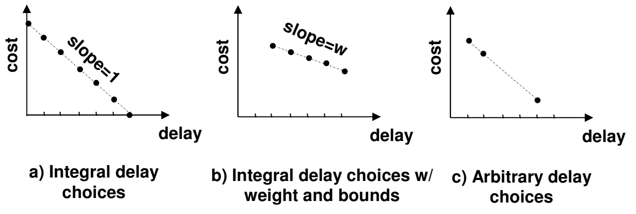

Figure 2: Three cases of discrete implementation delay choices: a) Integral delay choices. b)Integral delay choices with weights and bounds. c) Arbitrary delay choices. Cases a and b are handled efficiently.

lem, and present algorithms that can efficiently handle a variety of incremental updates. Examples updates include insertion or deletion of arcs, change in the delay of a node, and change in relative utility improvement (energy in our motivating example) with unit delay.

3. IMPLEMENTATION SELECTION PROBLEM

We focus on designs that can be modeled using directed acyclic graphs (DAG) $G = (V, E)$). Examples include gate-level netlists or high-level applications modeled as task graphs. 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 easily be modeled by inserting additional nodes on E. Node delays can be calculated using common gate-level delay models such as load-dependent model.

In library-based design methodologies, there are usually a number of pareto-optimal implementations available to realize components of a design [5]. For example, a complex gate can be implemented in several ways. Similarly from a functionality point of view, an addition operation in high-level design can be mapped to any adder module in the library. Different implementations come at different costs. In most cases, faster implementations of components incur higher cost in terms of typical design quality metrics, such as energy dissipation, area or dollar cost. For instance, a parallel multiplier takes more area and runs faster than a serial multiplier. Here, we assume that the cost of implementing a component is a linear function of its delay. Later we argue that our approach is extensible to any convex cost function of the delay.

An essential task in system synthesis is to select the proper pareto-optimal implementation for components of the design such that its timing constraint is met, and design cost metric is minimized. We refer to this problem as *implementation selection.* An important property of the problem that primarily determines its hardness, is the richness of delay choices available for implementing nodes. Ideally, we would like to be able to implement a node with any desired delay value. This is equivalent to assuming that delay choices of possible implementations of a node are *continuous.* Continuous delay choices do not exist in real life, and are not feasible from a practical point of view.

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. Note that integral delay choices implies that a unit relaxation in timing of any node would incur the same 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 shows this case. 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). In the next two sections, we focus on properties of the problem under the assumption of weighted and bounded integral delay choices.

We have developed a min-cost flow based algorithm that optimally solves the problem of implementation selection under weighted and bounded integral delay choices [8]. The details of our min-cost flow based technique are essential to understanding our incremental CISP method. Interested readers are referred to our publications on the topic [8, 7].

4. INCREMENTAL COMPONENT IMPLEMENTATION SELECTION

In order for designers to view the cost (e.g., energy dissipation) implications of engineering change orders applied to the design, component implementation selection problem has to be implicitly or explicitly solved. Applying the min-cost flow based method discussed in Section 3 requires iterative solution of min-cost flow on numerous subject graphs, which is prohibitive due to slow runtime of min-cost flow algorithms. Note that the time complexity of best min-cost flow algorithms can be as high as $O(n^3 \log(n))$ for dense design graphs [6].

Fortunately, only a small part of the design is updated during ECO. Thus, there is a good opportunity to quickly update the existing implementation selection solution (or simply solution) calculated on the original graph, without going through the lengthy process of re-solving the implementation selection problem from scratch. Our objective is to devise algorithms that given the original graph, the corresponding implementation assignment and a set of incremental changes, can efficiently update the implementation assignment while maintaining optimality.

4.1 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. The sequence translates the original design graph into the updated design graph. Our choice of primitive incremental operations include:

1. **Arc Insertion**: An edge is inserted between two existing nodes. The edge might render the problem infeasible in presence of lower bounds on node delays.

2. **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.

3. **Node Delay Increase**: Delay of a selected node is incremented. The change might render the problem infeasible due to imperative violation of timing constraint.

4. **Node Delay Decrease**: Delay of a selected node is decremented.

5. **Node Weight Change**: Weight of a selected node is increased or decreased by $w > 0$.

4.2 Problem Formulation

Recall that a component implementation selection instance, specified with a design graph G, timing constraint and possible node implementations, can be represented with the corresponding flow network H (according to the algorithm reviewed in Section 3). The initial full solution to the problem delivers min-cost flow solution of H and shortest-path distances in H^* with respect to a fixed node.

Our target incremental implementation selection problem 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 the implementation selection solution such that the timing constraint of the updated graph is met, and the optimality of the solution is maintained. The updated graph is simply constructed by sequential in-order application of primitive operations.*

5. INCREMENTALLY UPDATING A CISP SOLUTION UNDER ECO

In this section, we develop techniques to handle any single primitive operation. We present necessary and sufficient conditions to quickly evaluate validity (meeting the timing constraint) and optimality of a solution. Furthermore, the conditions provide a set of directives to incrementally update the selected implementations, if needed. Starting from an optimal solution, we prove that after handling any single primitive operation, the timing constraint is met (if possible) and solution optimality is maintained. Thus, our technique preserves the optimality of the solution under any high-level ECO, i.e., any sequence of primitive operations.

We denote the very last node of the design graph in topological ordering with z. Formally, z refers to the start node of the reverse timing constraint edge in min-cost flow network. Intuitively, the reverse timing constraint edge is added so that any violation of timing constraint leads to creation of a negative cycle in graph. Also, we denote the shortest path of z to any node i with π_i. Note that π is well-defined when there is no negative cycle in the graph. For our problem at hand, existence of a negative cycle means that the timing constraint is violated.

For edge e_{ij}, its reduced cost is defined as $c^\pi_{ij} = c_{ij} + \pi_i - \pi_j$, where c_{ij} is the cost of edge e_{ij}. Recall that cost of an edge, $c_{ij} = -d_{ij}$, where d_{ij} is the minimum implementation delay of the node corresponding to edge e_{ij} (Section 3). The vectors π and c^π represent *node potentials* and *reduced costs* in network flow terminology, respectively.

COROLLARY 1. *The summation of reduced costs over any cycle in H^* is equal to the summation of edge costs over that cycle.*

The complementary slackness conditions are necessary and sufficient conditions for optimality of a min-cost flow solution [6]:

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

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

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

where f_{ij} is the amount of flow on edge e_{ij}, and u_{ij} is its capacity (upper bound on its flow).

The basic idea of our algorithm is the following: For a min-cost flow solution to be optimal, the node potentials vector π has to satisfy the complementary slackness conditions (equations 1-3). We refer to a node potential vector that satisfies those conditions as *valid*. Hence, in our terminology, a valid node potential vector delivers optimal solution as well. Naturally, the initial full solution ("from-scratch") gives a valid node potential vector.

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 incrementally updated graph. However, if the primitive operation violates the validity conditions, the flow solution and node potentials need to be updated to comply with equations 1-3.

Our algorithm works based on this idea. 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. The *corrective measures* are based on properties of min-cost flows and their sensitivity to incremental changes [6]. Due to page limitation in the final version, we are forced to remove the details of our algorithm.

Intuitively, reduced cost of an edge is equal to relaxation in its implementation timing. For example, a reduced cost of 2 for some edge in H^* means that the corresponding node in G will be implemented with delay of $d + 2$, where d is the minimum possible delay for that node. For our problem at hand, all of the edges of graph H have infinite capacity. Thus, the case of equation 3 cannot happen. In other words, the reduced costs (c^π_{ij}) for all edges of H^* should be non-negative for node potentials to be valid.

6. EXPERIMENT RESULTS

We applied our technique to problem of gate-level threshold voltage (V_t) assignment for leakage optimization. For each gate, there are two possible implementations available in the library that correspond to fabricating that gate with either high or low threshold voltage. Implementation with low V_t, as opposed to high V_t, results in faster but leakier gates. The relation between leakage and delay is convex, and implementations fall under arbitrary delay choices category (Figure 2.c).

We implemented both from-scratch implementation selector and our incremental algorithm in SIS. After temporary relaxation of arbitrary delay choices to consecutive integers and solving using aforementioned 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.

Circuit	Cell count	Leakage (normalized)				Runtime (sec)		
		original	full	incremental	error(%)	full	incremental	speedup
C2670	505	4384	808.2	808.2	0	9.15	0.29	31.9
C1355	510	2913.4	814.7	812.8	0.23	5.97	0.26	22.9
alu4	1579	7398.2	1164.6	1164.6	0	64.23	0.48	132.9
spla	4603	24127.2	3625.7	3625.7	0	824.07	1.52	543.8
ex1010	5045	21090.7	3187.5	3187.5	0	937.59	5.03	186.3
pdc	5812	31294.4	4669.7	4669.7	0	1582.32	1.75	901.7
Average	3009				0.04			303.3

Table 1: Runtime and leakage comparison between incremental and from-scratch implementation selection.

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. Gate delay, leakage, and input capacitance data are borrowed from the study performed by Khandelwal and Srivastava [9].

We assume that alterations imposed by an average ECO can be replicated by a sequence of 50 primitive incremental operations such as, delay change, arc insertion, weight change, or arc deletion. Primitive operation types (e.g., delay change or arc insertion) and their associated parameters (e.g., how much to change the delay or location to insert the arc) are generated randomly. Primitive operations that would render the solution infeasible are not considered.

Table 1 summarizes our experimental results. For each circuit, original leakage (all gates assigned to low V_t) and optimized leakage after running both full (from-scratch) and incremental algorithms are reported in columns $3-5$ of the table. Note that one iteration of the full algorithm is being compared with 50 iterations of incremental algorithm. Column 6 ($error\%$) 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.

The last three columns report the runtime of 1)one call to full solver to handle an average ECO, 2) successive calls to our incremental handling method after each primitive operation, and 3) the speedup gained by our approach. The runtimes were recorded over 40 runs and the average numbers are reported. Our experiments show that the incremental algorithm is about 303 times faster than running the full algorithm, while delivering the same quality results. The runtime improvement is more significant for larger circuits, and hence, the gap is expected to widen for more complex benchmarks.

For practical arbitrary delay choices, our min-cost flow solution serves as a mechanism to devise a powerful heuristic, which delivers high quality solutions (not necessarily optimal). Thus, 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.

7. 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. 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 quality results.

8. REFERENCES

[1] 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.

[2] D. Brand, A. Drumm, S. Kundu, P. Narain. "Incremental Synthesis". In *International Conference on Computer-Aided Design*, pages 14–18, 1994.

[3] J. Cong, M. Sarrafzadeh. "Incremental Physical Design". In *International Symposium on Physical Design*, pages 84–92, 2000.

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

[5] P. Yang, F. Catthoor. "Pareto-optimization-based Run-time Task Scheduling for Embedded Systems". In *International Symposium on HW/SW Codesign*, pages 120–125, 2003.

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

[7] 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.

[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] 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.