DeltaSyn: An Efficient Logic Difference Optimizer for ECO Synthesis *

Smita Krishnaswamy‡, Haoxing Ren‡, Nilesh Modi+, and Ruchir Puri ‡
‡ IBM T.J. Watson Research Center, Yorktown Heights, NY
+ Dept. of ECE, University of California, Santa Barbara, CA
{skrishn, haoxing, ruchir}@us.ibm.com, nilesh@ece.ucsb.edu

Abstract

During the IC design process, functional specifications are often modified late in the design cycle, after placement and routing are completed. However, designers are left either to manually process such modifications by hand or to restart the design process from scratch—a very costly option. In order to address this issue, we present DeltaSyn, a method for generating a highly optimized logic difference between a modified high-level specification and an implemented design. DeltaSyn has the ability to locate boundaries in implemented logic within which changes can be confined. DeltaSyn demarcates the boundary in two phases. The first phase employs fast functional and structural analysis techniques to identify equivalent signals forming the input-side boundary of the changes. The second phase locates the output-side boundary of the changes through a novel dynamic algorithm that detects matching logic downstream from the changes required by the ECO. Experiments on industrial designs show that together these techniques successfully implement ECOs while preserving an average of 97% of the existing logic. Unlike previous approaches, the use of bit-parallel logic simulation and fast SAT solvers enables high performance and scalability. DeltaSyn can process and verify a typical ECO for a design of around 10K gates in about 200 seconds or less.

1 Introduction and Background

As the IC industry matures, it becomes common for existing designs to be modified incrementally. Since redesigning logic involves high costs in terms of design effort and time, previous designs must be maximally re-utilized whenever possible. Designers have noted that, in existing flows, even a small change in the specification can lead to large changes in the implementation [7]. More generally, the need for CAD methodologies to be less sequential in nature and allow for transformations that are "incremental and heterogeneous" has been recognized by leaders in industry [6].

Recent advances in incremental physical synthesis [2, 20], placement [15], routing [23], timing analysis [21] and verification [5] have made ECO tools practical. However, logic synthesis remains a bottleneck in incremental design for several reasons. First, it is difficult to process incremental changes in the design manually since logic optimizations can render intermediate signals unrecognizable. Second, the inherent randomness in optimization choices makes the design process unstable, i.e., a slight modification of the specification can lead to a different implementation. Therefore, a general logic ECO methodology that is able to quickly derive a small set of changes in logic to handle incremental updates is necessary.

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

Prior work on logic ECO tends to be impractical due, in some cases, to the use of unscalable functional techniques [11, 19] involving BDD-manipulation. In other cases, the heavy reliance on structural correspondences [3, 17] can result in large difference models that disrupt much of the existing design when such correspondences are unavailable.

In this paper, we present DeltaSyn, a method for producing a synthesized delta or *logic difference* between an RTL-level ECO specification and an implemented design. As illustrated in Figure 1, DeltaSyn combines both functional and structural techniques to minimize the logic difference. As a pre-processing step, we compile the modified specification into a preliminary technology-independent gate-level netlist with little optimization. Phase I finds structurally and functionally equivalent gates to determine the input-side boundary of the logic difference. Phase II uses a novel topologically-guided functional technique that finds matching subcircuits starting from primary outputs and progressing upstream to determine the output-side boundary of the change. DeltaSyn allows designers to avoid most design steps including much of logic synthesis, technology mapping, placement, routing, buffering, etc., on the unchanged logic.

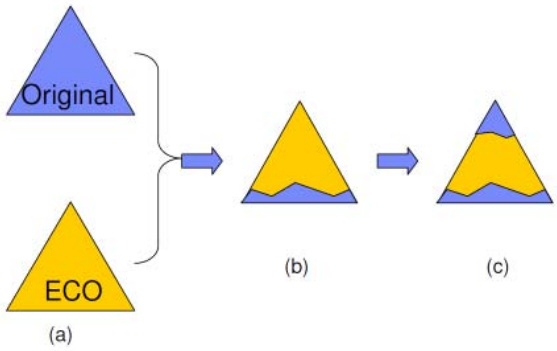


Figure 1: The main phases of DeltaSyn: (a) the original design and the modified specification are given as inputs to DeltaSyn, (b) functional and structural equivalences forming the input-side boundary of the changes are identified, (c) matching subcircuits which form the output-side boundary of the changes are located and verified.

The main contributions of this paper include:

- An efficient dual-phase flow that integrates fast functional and structural techniques to reduce the logic difference through the identification of input and output-side boundaries of the change.
- A novel dynamic algorithm that finds matching subcircuits between the modified specification and implemented design to significantly decrease the logic difference.

A key feature of our approach is that, unlike other ECO methodologies, we make no assumptions about the type or

extent of the changes in logic. The remainder of the paper is organized as follows. Section 2 describes previous work in logic ECO synthesis. Section 3 describes the overall flow of DeltaSyn. Sections 3.1 and 3.2 describe our equivalence-finding and subcircuit-matching phases of logic difference reduction. Section 4 presents empirical results and analysis. Finally, Section 5 concludes the paper.

2 Previous Work

Recently, the focus of ECO design has been on changes to routing or placement [10, 15, 2]. However, there have been several papers dealing specifically with logic ECO. Authors of [3, 17] present techniques that depend on structural correspondences. They find topologically corresponding nets in the design. Then, gates driving these nets are replaced by the correct gate type. While this type of analysis is generally fast, it can result in many changes to the design since such structural correspondences are hard to find in designs that undergo many transformations.

In contrast, the method from [11] does not analyze topology. Instead, it uses a BDD-based functional decomposition technique to identify sets of candidate signals that are able to correct the outputs of the circuit to achieve the ECO. The authors rewrite functions of each output $O(X)$ in terms of internal signals t_1, t_2 to see whether there are functions that can be inserted at t_1, t_2 to realize a new function O'. In other words, they solve the Boolean equation $O(X, t_1, t_2) = O'$ for t_1 and t_2 and check for consistency. This method does not scale well due to the memory required for a BDD-based implementation of this technique.

More recently, Ling et al. [12] present a MAX-SAT formulation similar to that of [16] for logic rectification in FPGA-based designs. Rectification refers to corrections in response to missing or wrong connections in the design. They find the maximum number of clauses that can be satisfied between a miter that compares the original implementation and the modified specification. Then, gates corresponding to unsatisfied clauses are modified to correct the logic. They report that approximately 10% of the netlist is disrupted for five or fewer errors. For more significant ECO changes, MAX-SAT can produce numerous unsatisfied clauses since it depends on the existence of functional equivalences. Further, different maximally satisfiable clause sets can produce different results. Optimal solutions can require searching through the space of MAX-SAT solutions for a particular instance, which can be computationally costly.

3 DeltaSyn

In this section, we describe the logic ECO problem and our solution techniques. First, we define terms that are used through the remainder of the paper.

Definition 1 *The* original model *is the original synthesized, placed, routed, and optimized design.*

Definition 2 *The* ECO specification *is the modified RTL-level specification, i.e. the change order.*

Definition 3 *The* difference model *is a circuit representing the changes to the original model required to implement the ECO specification. The set of gates in the difference model represent new gates added to the original model. Wires represent connections among these gates. The primary inputs and primary outputs of the difference model are annotated by connections to existing gates and pins in the original model.*

Given the original model, the ECO specification and a list of corresponding primary outputs and latches, the objective of combinational ECO synthesis is to derive a *difference model* that is minimal in the number of gates. The new specification, generally written in an RTL-level hardware description language (such as VHDL), is compiled into a technology-independent form called the *ECO Model*. This step is relatively fast because the majority of the design time is spent in physical design including physical synthesis, routing, and analysis [14] (see Figure 18).

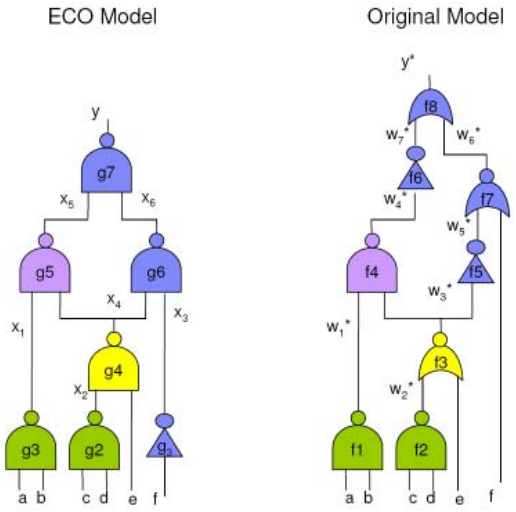


Figure 2: Sample circuits to illustrate our method.

The circuits in Figure 2 are used to broadly illustrate the two phases of our difference optimization. By inspection, it is clear that $f3$, which was modified from a *NOR* to a *NAND* gate, is the only difference between the two circuits. Although some of the local logic has undergone equivalent transformations (gates g6 and g7 have been modified through the application of DeMorgan's law), most of the circuit retains its global structure. The nearby logic in the original model being unrecognizable despite the actual change being small is typical of ECO examples.

DeltaSyn recognizes and isolates such changes as follows: Our first phase involves structural and functional equivalence checking. For the given example, the equivalences $x_1 = w_1^*, x_2 = w_2^*$ are recognized by these techniques. Our second phase is geared towards finding matching subcircuits from the primary outputs. Through a careful process of subcircuit enumeration and subcircuit-input mapping, the subcircuits consisting of $\{f5, f6, f7, f8\}$ from the original model and $\{g3, g6, g7\}$ are matched (using Boolean matching techniques) under the intermediate input mapping $\{(x_5, w_4^*), (x_4, w_3^*), (f, f)\}$. This match leads to an additional match between $g5$ and $f4$ when the matching algorithm is recursively called on signals x_5 and w_4^*. The conclusion of the two phases leaves the single-gate difference of $g4$, as desired. The remainder of this section explains the algorithms involved in these steps.

3.1 Phase 1: Equivalence-based Reduction

Phase I is illustrated in Figure 3. Starting with the given list of corresponding primary inputs and latches, DeltaSyn builds a new correspondence list L between matched signals in the original and ECO models. Matches are found both structurally and functionally. Candidates for functional equivalence are identified by comparing simulation responses and verified using SAT.

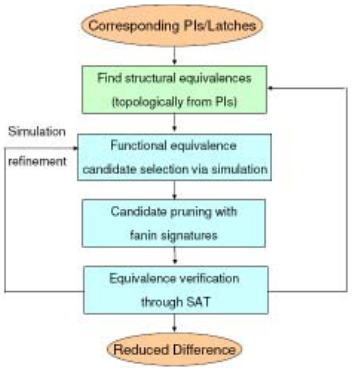


Figure 3: Logic difference reduction through equivalences.

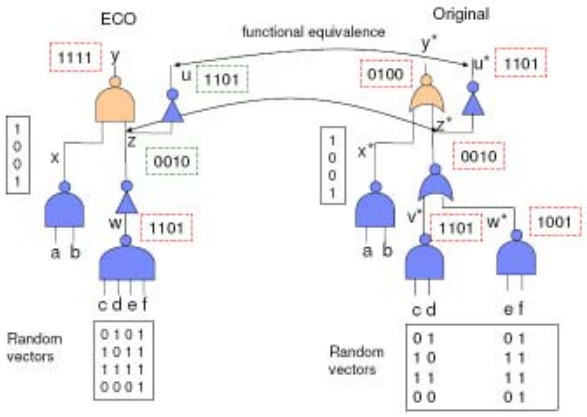


Figure 4: Identifying structural and functional equivalences.

Structural equivalences are found inductively, starting with corresponding primary inputs and latch outputs. All gates g, g' whose input signals correspond, and whose functions are identical, are added to the correspondence list. The correspondence list can keep track of all pairwise correspondences (in the case of one-to-many correspondences that can occur with redundancy removal). This process is then repeated until no further gate-outputs are found to structurally correspond with each other.

Example 1 *In Figure 4 the initial correspondence list is* $L = \{(a,a^*)(b,b^*)(c,c^*)(d,d^*)(e,e^*)(f,f^*)\}$. *Since both the inputs to the gate with output x are in L, we examine gate* x^* *in the original model. Since this gate is of the same type as x,* (x,x^*) *can be added to L.*

After the structural correspondences are exhausted, the combinational logic is simulated in order to generate candidate functional equivalences. The simulation proceeds by first assigning common random input vectors to signal-pairs in *L*. Signals with the same output response on thousands of input vectors (simulated in a bit-parallel fashion) are considered candidates for equivalence, as in [13, 9]. These candidates are further pruned by comparing a pre-computed *fanin signature* for each of these candidate signals. A fanin-signature has bit position representing each PI and latch in the design. This bit is set if the PI or latch in question is in the transitive fanin cone of the signal and unset otherwise. Fanin signatures for all internal signals can be pre-computed in one topological traversal of the circuit.

Example 2 *In Figure 4, the same set of four random vectors are assigned to corresponding input and internal signals. The output responses to each of the inputs is listed horizontally. The simulations suggest that* $(z,z^*),(u,u^*),(w,v^*)$ *are candidate-equivalences. However, the fanin list of* $v*$ *contains PIs* c^*,d^* *but the list for w contains* c,d,e,f. *Therefore, these signals are not equivalent.*

Equivalences for the remaining candidates are verified using SAT. We construct miters between candidates signals by connecting the corresponding primary inputs together and check for satisfiability. UNSAT assignments can be used to update simulation vectors.

Note that it is not necessary for all intermediate signals to be matched. For instance, if two non-identical signals are merged due to local observability don't cares (ODCs) as in [24], then downstream equivalences will be detected after the point at which the difference between the signals becomes unobservable. After functional equivalences are found, all of the gates driving the signals in *L* can be deleted from the difference model.

3.2 Phase 2: Matching-based Reduction

Phase II of DeltaSyn finds subcircuits that are functionally equivalent under some permutation of intermediate signals. Since ECOs tend to be small changes in large netlists, there are large areas of logic that are identifiably unchanged once the dependence on the changed logic is removed. In other words, once the "output-side boundary" of the change is disconnected, the remaining logic should be equivalent under an appropriate association (connection) of internal signals (as illustrated in Figure 2).

At the outset, the task of finding matching subcircuits seems to be computationally complex because it is unclear where the potentially matching subcircuits are located within the ECO and original models. Enumerating all possible subcircuits (or even a fraction of them) is a computationally intractable task with exponential complexity in the size of the circuit. Additionally, once such candidate subcircuits are located, finding an input ordering such that they functionally match is itself an *NP*-complete problem known as *p*-equivalence Boolean matching (we actually find all such input orders). While these problems are generally high in complexity, we take advantage of two context-specific properties in order to effectively locate and match subcircuits:

1. Most ECOs are small changes in logic.
2. The majority of logic optimizations performed on the original implementation involve local transformations that leave the global structure of the logic in tact.

In fact, about 90% of the optimizations that are performed in the design flow are physical synthesis optimizations such as factoring, buffering, and local timing-driven expansions [20, 18, 8, 22]. While redundancy removal can be a non-local change, equivalent signals between the two circuits (despite of redundancy removal) can be recognized by techniques in Phase I. Since we expect the change in logic to be small, regions of the circuit farther from the input-side boundaries are more likely to match. Therefore, we enumerate subcircuits starting from corresponding primary outputs in order to find upstream matches. Due to the second property, we are able to utilize local subcircuit enumeration. The subcircuits we enumerate are limited by a width of 10 or fewer inputs, thereby improving scalability. However, after each subcircuit pair is matched, the algorithm is recursively invoked on the corresponding inputs of the match.

Figure 5 illustrates the main steps of subcircuit identification and matching. Candidate subcircuits are generated by expanding two corresponding outputs along their fanin cone. For each candidate subcircuit pair, we find input symmetry classes, and one input order under which the two circuits are equivalent (if such an order exists). From this order, we are able to enumerate all input orders under which the circuits are equivalent. For each such order, the algorithm is called recursively on the corresponding inputs of the two subcircuits.

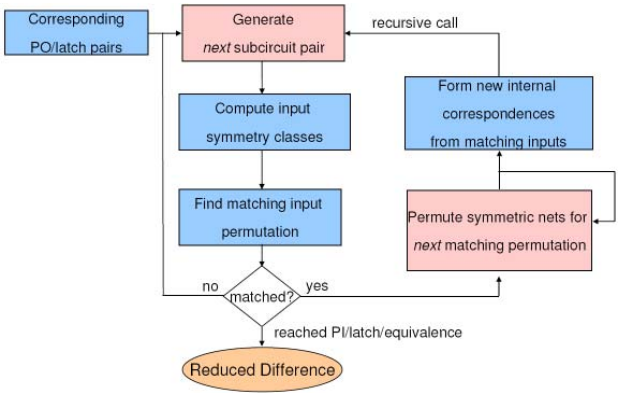


Figure 5: Difference reduction through subcircuit matching.

3.2.1 Subcircuit Enumeration

For the purposes of our matching algorithm we define a subcircuit as follows:

Definition 4 *A subcircuit C consists of the logic cone between one output O, and a set of inputs* $\{i_1, i_2, \ldots i_n\}$.

Pairs of subcircuits, one from the original model, and one from the ECO model are enumerated in tandem. Figure 6 illustrates the subcircuit enumeration algorithm. Each subcircuit in the pair starts as a single gate and expands to incorporate the drivers of its inputs. For instance, in Figure 6, the subcircuit initially contains only the gate driving primary output z and then expands in both the x and y directions. The expansion in a particular direction is stopped when the input being expanded is a) a primary input, b) a latch, c) a previously identified equivalence , d) a previously matched signal, e) the added gate increases the subcircuit width beyond the maximum allowed width, or f) the signal being expanded has other fanouts outside the subcircuit (signals with multiple fanouts can only be at the output of a subcircuit).

The pseudocode for subcircuit expansion is shown in Figure 8. The *get_next_pair* function fills in the variables C and C^*. The *orig_queue* is popped. If the *orig_queue* is empty, then all the possible subcircuits C^* in the original model have already been enumerated for a particular subcircuit C in the ECO model. In this case, a new ECO subcircuit is found by popping the *eco_queue*. If a particular pair of subcircuits has already been seen (and recorded in the *pair_history*) then the next pair is generated. If the *eco_queue* is also empty, then all possible pairs of subcircuits have already been enumerated for the pair (N, N^*) and the process terminates.

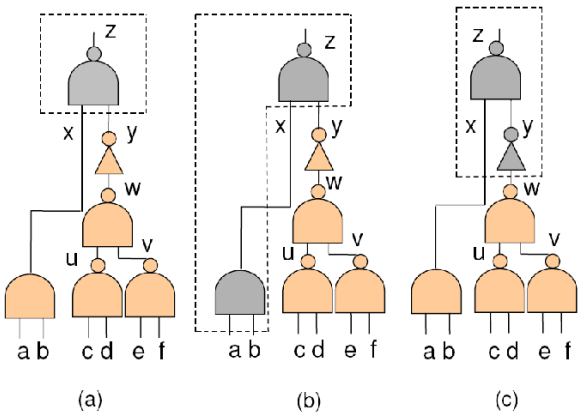


Figure 6: Candidate subcircuit enumeration.

```
CLASS subcirc_enum(A)
{
        eco_output
        orig_output
        eco_queue
        orig_queue
        pair_history

        next_subcircuit_pair(C1, C2)
        expand_subcircuit( subcircuit C, queue Q)
}
```

Figure 7: The data structure for enumerated subcircuits.

```
bool NEXT_SUBCIRCUIT_PAIR(subcircuit C_eco, subcircuit C_orig)
{
        do
                if (eco_queue.empty())
                        return FALSE
                C_eco = eco_queue.front()
                if(orig_queue.empty())
                        eco_queue.pop()
                        expand_subcircuit(C_eco, eco_queue)
                        orig_queue.clear()
                        orig_queue.push(new_subcircuit(driver(O*)))
                C_orig = orig_queue.front()
                orig_queue.pop()
                expand_subcircuit(C_orig, orig_queue)
        while(pair_history.find(C_eco,C_orig))
        pair_history.add(C_eco,C_orig)
        return TRUE
}
```

Figure 8: Subcircuit pair enumeration algorithm.

```
EXPAND_SUBCIRCUIT(subcircuit C, queue Q)
{
        for(all inputs i ∈ C)
                if(has_outside_fanouts(i))continue
                g = get_driver(i)
                if(is_PI(g)|| is_latch(g)continue
                if(is_equivalent(g)|| is_matched(g))continue
                if(num_inputs((C⋃g) > MAX))continue
                Q.push(new_subcircuit(C⋃g))
}
```

Figure 9: Algorithm for subcircuit expansion.

3.2.2 Subcircuit Matching

For two candidate subcircuits (C, C^*) realizing the Boolean functions $F(i_1, i_2, \ldots i_n)$ and $F^*(i_1*, i_2*, \ldots i_n*)$ respectively, our goal is to find *all* of the permutations of the inputs of F^* such that $F = F^*$. Note that this is not necessary for

most uses of Boolean matching (such as technology mapping). We elaborate on this process below.

Definition 5 *A* matching permutation $\rho_{(F^*,F)}$ *of a function* $F^*(i_1,i_2,\ldots,i_n)$ *with respect to a function* F*, is a permutation of its inputs such that* $F^*(\rho_{(F^*,F)}(i_1),\rho_{(F^*,F)}(i_2),\ldots\rho_{(F^*,F)}(i_n)) = F$.

Definition 6 *Two inputs* i_x *and* i_y *of a function* F *are said to be* symmetric *with respect each other if*

$$F(i_1,i_2,\ldots\mathbf{i_x},\ldots\mathbf{i_y}\ldots i_n) = F(i_1,i_2,\ldots\mathbf{i_y},\ldots\mathbf{i_x}\ldots i_n)$$

Definition 7 *Given a function* F *and a partition of its inputs into symmetry classes*

$$sym\_F = \{sym\_F[1], sym\_F[2], \ldots sym\_F[n]\},$$

a symmetric permutation τ_F *on the inputs of* F *is a composition of permutations on each symmetry class* $\tau_F = \tau_{sym\_F[1]} \circ \tau_{sym\_F[2]} \ldots \tau_{sym\_F[n]}$. *Each constituent permutation* $\tau_{sym\_F[i]}$ *leaves all variables not in* $sym\_F[i]$ *fixed.*

We now state and prove the main property that allows us to derive all matching permutations.

Proposition 1 *Given a matching permutation* $\rho_{(F^*,F)}$*, all other matching permutations* $\pi_{(F^*,F)}$ *can be derived by composing a symmetric permutation* τ *with* $\rho_{(F^*,F)}$*, i.e.,* $\pi_{(F^*,F)} = \tau \circ \rho_{(F^*,F)}$ *for some symmetric permutation* τ.

Proof 1 Assume there exists a matching permutation $\pi_{(F^*,F)}$ that cannot be derived by composing a symmetric permutation with $\rho_{(F,F^*)}$. Then, there is a permutation ϕ which permutes a set of non-symmetric variables S' such that $\phi \circ \rho_{(F,F^*)} = \pi_{(F^*,F)}$. By definition of symmetry, $F^*(\phi(\rho_{(F,F^*)}(i_1)),\phi(\rho_{(F,F^*)}(i_2)),\phi(\rho_{(F,F^*)}(i_3))\ldots) \neq F^*(\rho_{(F,F^*)}(i_1),\rho_{(F,F^*)}(i_2),\rho_{(F,F^*)}(i_3))$. However, since $F^*(\rho_{(F,F^*)}(i_1),\rho_{(F,F^*)}(i_2),\rho_{(F,F^*)}(i_3)) = F$,

$$F^*(\phi(\rho_{(F,F^*)}(i_1)),\phi(\rho_{(F,F^*)}(i_2)),\phi(\rho_{(F,F^*)}(i_3))\ldots) \neq F.$$

Therefore, $\pi_{(F^*,F)}$ cannot be a matching permutation. For the other side, suppose ϕ is any symmetric permutation of F^* then by definition of symmetry $F^*(\phi(\rho_{(F,F^*)}(i_1)),\phi(\rho_{(F,F^*)}(i_2)),\phi(\rho_{(F,F^*)}(i_3))\ldots) = F^*(\rho_{(F,F^*)}(i_1),\rho_{(F,F^*)}(i_2),\rho_{(F,F^*)}(i_3))$, and by transitivity,

$$F^*(\phi(\rho_{(F,F^*)}(i_1)),\phi(\rho_{(F,F^*)}(i_2)),\phi(\rho_{(F,F^*)}(i_3))\ldots) = F$$

Therefore, $\phi \circ \rho$ is also a matching permutation of F^* with respect to F.

Proposition 1 suggests that all matching permutations can be derived in these steps:

1. Computing the set of input symmetry classes for each Boolean function, i.e., for a function F we compute $sym\_F = \{sym\_F[1], sym\_F[2], \ldots sym\_F[n]\}$ where classes form a partition of the inputs of F and each input is contained in one of the classes of sym_F.
2. Deriving one matching permutation through the use of a Boolean matching method.
3. Permuting symmetric variables within the matching permutation derived in step 2.

```
COMPUTE_SYM_CLASSES(function F)
{
    unclassified[] = all_inputs(F)
    do
        curr = unclassified[0]
        for(i < |unclassified|)
            F' = swap(F, curr, unclassified[i])
            if(F' == F)
                new_class.add(unclassified[i])
            new_class.add(curr)
            sym_F.add(new_class)
            unclassified[] = all_inputs(F) -symmetry_classes
    while(!unclassified.empty())
    return sym_F
}
```

Figure 10: Computing symmetry classes.

To obtain the set of symmetry classes for a Boolean function F we recompute the truth-table bitset after swapping pairs of inputs. This method has complexity $O(n^2)$ for a circuit of width n, and this method is illustrated in Figure 10.

We derive a matching permutation of F^*, or determine that one does not exist through the algorithm shown in Figure 11. In the pseudocode, instead of specifying permutations $\rho_{F^*,F}$, we directly specify the ordering on the variables in F^* that is induced by ρ when F is ordered in what we call a *symmetry class order*, i.e. F with symmetric variables adjacent to each other, as shown below:

$$F(sym\_F[1][1], sym\_F[1][2], \ldots sym\_F[1][n], sym\_F[2][1], sym\_F[2][2], \ldots sym\_F[2][n]\ldots)$$

The $reorder(F, sym\_F)$ function in the pseudocode is used to recompute the functions F according to the order suggested by sym_F (and similarly with F^*). The overall function is explained below:

1. First, we check whether number of inputs in both the functions is the same.
2. Next, we check the sizes and number of symmetry classes. If the symmetry classes all have unique sizes, then the classes are considered *resolved*.
3. If the symmetry classes of F and F^* are resolved, they can be associated with each according to class size and immediately checked for equivalence.
4. If the symmetry classes do not have distinctive sizes, we use a simplified form of the method from [1], denoted by the function $matching\_cofactor\_order$ in Figure 11. Here, cofactors are computed for representative members of each unresolved symmetry class, and the minterm counts of the nth-order cofactors are used to associate the classes of F with those of F^*. This determines a permutation of the variables of F^* up to symmetry classes.

The remaining matching permutations are derived by enumerating symmetric permutations as shown in Figure 12. The $next\_permutation$ function enumerates permutations of individual symmetry classes. Then all possible combinations of symmetry class permutations are composed with each other.

The different input orders induced by matching permutations define different associations of intermediate signals between the subcircuit from the original model C^* and that of the ECO model C. Figure 13 illustrates that although two subcircuits can be equivalent under different input orders, the "correct" order leads to larger matches upstream.

```
bool COMPUTE_MATCHING_PERM_ORDER(function F, function F*)
{
        if(|inputs(F)|! = |inputs(F*)|)
                return UNMATCHED
        sym_F = compute_sym_classes(F)
        sym_F* = compute_sym_classes(F*)
        sort_by_size(sym_F)
        sort_by_size(sym_F*)
        if(|sym_F|! = |sym_F*|)
                return UNMATCHED
        for(0 <= i < |sym_F|)
                if(|sym_F[i]|! = |sym_F*[i]|)
                        return UNMATCHED
        if(resolved(sym_F*))
                reorder(F*, sym_F*)
                reorder(F, sym_F)
                if(F* == F) return MATCHED
                else return UNMATCHED
        if(matching_cofactor_order(F, sym_F, F*, sym_F*))
                return MATCHED
        else
                return UNMATCHED

}
```

Figure 11: Compute a matching permutation order.

```
NEXT_MATCHING_PERM_ORDER(sym_classes sym_F*, function F*)
{
        index = −1
        for(0 <= i < |sym_F*|)
                if(next_permutation(sym_F*[i]))
                        index = i
                        break
        if(index == −1)
                return NULL
        for(0 <= j < i)
                next_permutation(sym_F*[j])
        reorder(sym_F*, F)
}
```

Figure 12: Enumerating matching input orders.

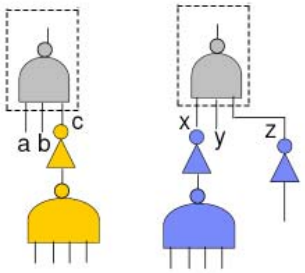


Figure 13: Although the single-gate subcircuits in the boxes have completely symmetric inputs, the input ordering (c,b,a) leads to a larger upstream match than (a,b,c).

3.2.3 Subcircuit Covering

In this section, we describe a recursive covering algorithm which derives a set of subcircuits or *cover* of maximal size.

Definition 8 *A* subcover *for two corresponding nets* (N,N^*) *is a set of connected matching subcircuit-pairs that drive N and* N^*.

Different subcircuit matches at (N,N^*) can lead to different subcovers as shown in Figure 14. Once the subcircuit D of the original and D^* are generated through subcircuit enumeration algorithm of Figure 8, the algorithm of Figure 11 finds an input ordering under which they are functionally equivalent. Figure 14a shows the initial ordering where inputs $(0,1)$ of are associated with inputs $(0,1)$ of D^*. The subcover induced by this ordering is simply $\{(D,D^*)\}$, leaving the logic difference $\{A,B,C\}$. However, an alternate input ordering—derived by swapping the two symmetric inputs of D^*—yields a larger cover.

Since $D(1,0) = D^*(0,1)$, the covering algorithm is invoked on the pairs of corresponding inputs of D and D^*. The subcircuits (B,B^*) are eventually found and matched. The inputs of B,B^* are then called for recursive cover computation. One of the inputs of B is an identified functional equivalence (from phase 1) so this branch of recursion is terminated. The recursive call on the other branch leads to the match (A,A^*) at which point this recursive branch also terminates due to the fact that all of the inputs of A are previously identified equivalences. The resultant logic difference simply consists of $\{C\}$. Note that this subcover requires a reconnection of the output of A to C which is reflected in the difference model.

Figure 15 shows the algorithm to compute the *optimal subcover*. The algorithm starts by enumerating all subcircuit pairs (see Figure 8) and matching permutations (see Figure 12) under which two subcircuits are equivalent. The function is recursively invoked at the input of each matching mapping in order to extend the cover upstream in logic. For each match C_eco, C_orig with input correspondence $\{(i_1,j_1),(i_2,j_2),(i_3,j_3)\ldots\}$ (defined by the matching permutation), the best induced subcover is computed by combining best subcovers $opt\_match(i_1,j_1), opt\_match(i_2,j_2)\ldots$ at the inputs of the match. The subcovers are corrected for any conflicting matches during the process of combining. For example, if a net in the ECO model has higher fanout than a net in the original model then different subcovers may correspond the ECO net with different original nets. When such conflicts occur, the correspondence that results in the larger subcover is retained.

```
COMPUTE_COVER(net N, net N*)
{
        subcirc_enum N_enum
        while(N_enum.next_subcircuit_pair(C,C*)){
                F* = compute_truth_table(C*)
                F = compute_truth_table(C)
                sym_F = compute_symm_classes(F)
                sym_F* = compute_symm_classes(F*)
                if(!compute_matching_perm_order(F, F*))
                        continue
                do{
                        for(0 <= i < |sym_F|)
                        for(0 <= j < |sym_F[i]|)
                                if(is_PI_latch_matched(sym_F[i][j]))
                                        continue
                                if(is_PI_latch_matched(sym_F*[i][j]))
                                        continue
                                compute_cover(sym_F[i][j], sym_F*[i][j])
                        this_match = combine_subcovers(sym_F, sym_F*)
                        if(|this_match| > |opt_match(N,N*)|)
                                opt_match(N,N*) = this_match
                }while(next_matching_perm_order(sym_F*))
        }
        mark_matched_gates(opt_match(N,N*))
}
```

Figure 15: The recursive subcircuit covering algorithm.

In this process, we search for matches starting with topologically corresponding primary outputs, and further topological correspondences emerge from the matching processes. Since topologically connected gates are likely to be

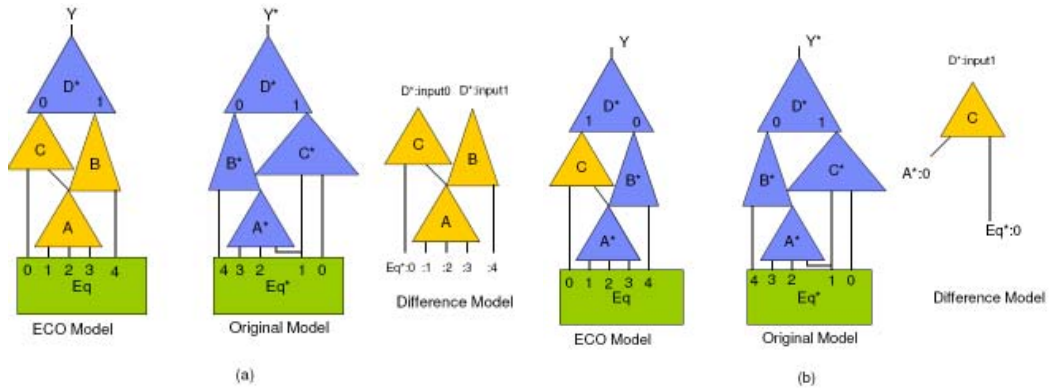


Figure 14: Snapshots of subcircuit covering: (a) Subcover induced by input orderi g $D^*(0,1)$ on original model and resulting difference (b) Subcover induced by input ordering $D^*(1,0)$, and the resulting (smaller) logic difference.

placed close to each other during physical design, many of the timing characteristics of the original implementation are preserved in reused logic. After a subcircuit cover is found, the outputs of subcircuit-pairs are added to the correspondence list L and all the covered gates in the ECO model are removed from the difference model.

4 Empirical Validation

Design	No. Gates	Runtime CPU (s)	Cone Size	Diff. Model Size	% Diff. Reduced	% Design Preserved
ibm1	3271	35.51	342	17	95.03	99.48
ibm2	2892	47.40	1590	266	83.27	90.80
ibm3	6624	192.40	98	1	98.98	99.98
ibm4	20773	20.32	774	4	99.48	99.98
ibm5	2681	10.01	1574	177	88.75	100.00
ibm6	1771	4.99	318	152	52.20	91.42
ibm7	3228	180.00	69	0	100.00	100.00
ibm8	5218	9.01	22	13	40.91	99.75
ibm9	532	38.34	77	20	74.03	96.24
ibm10	11512	0.40	1910	429	77.54	96.27
ibm11	6650	211.02	825	126	84.73	98.11
ibm12	611	0.23	47	0	100.00	100.00
ibm13	1517	6.82	21	6	71.43	99.60
Avg.					82.03	97.31

Table 1: DeltaSyn statistics on IBM benchmarks.

	Runtime (s)		% Runtime	% Slack
Design	Entire Design	Difference	Decrease	Decrease
ibm1	23040	823	96.43	27.79
ibm2	3240	1414.13	56.35	-20.83
ibm3	10800	1567	85.49	21.95
ibm4	50400	2146	95.74	9.36
ibm5	22680	1315	94.20	99.02
ibm6	2160	665	69.21	-2.97
ibm7	2160	748	65.38	69.72
Avg.			80.40	29.15

Table 2: PDSRTL [20] runtime and slack comparison between incremental design and complete redesign.

We empirically validate our algorithms on actual ECOs, i.e., modifications to the VHDL specifications, performed in IBM server designs. Our experiments are conducted on AMD Opteron 242, 1.6GHZ processors with 12GB RAM. Our code is written in C++ and compiled with g++ on a GNU linux operating system. For our experimental setup, we initially compiled the modified VHDL into a technology independent netlist with some fast pre-processing optimizations [18] that took 0.01% of the design time. The result, along with the original mapped/placed/routed design was analyzed by DeltaSyn to derive a logic difference. Results of this experiment are shown in Table 1. The logic difference is compared with the difference derived by the *cone-trace system*, which is used in industry. The cone-trace system copies the entire fanin cone of any mismatching primary output to the difference model and resynthesizes the cone completely. Table 1 shows an average improvement of 82% between the results of DeltaSyn and those of the cone trace system. The entries with difference size 0 represent changes that were achieved simply by reconnecting nets. Note that the reduction numbers only reflect the results of DeltaSyn and not pre-processing optimizations.

While the lack of standard benchmarks in this field makes it hard to directly compare to previous work, it should be noted that DeltaSyn is able to derive a small difference model for benchmarks that are significantly larger than previous work [3, 11]. DeltaSyn processes all benchmarks in 211 or fewer seconds. The more (global) structural similarity that exists between the ECO model and the original model, the faster DeltaSyn performs. For instance, ibm12 is analyzed in less than 1 second because similarities between the implemented circuit and the ECO model allow for the algorithm in Figure 15 to stop subcircuit enumeration (i.e. stop searching for potential matches) and issue recursive calls frequently. Any fast logic optimizations that bring the ECO model structurally closer to the original model can, therefore, be employed to improve results. Figure 17 shows the relative percentages of difference model size reduction achieved by our two phases. The first phase reduces the logic difference by about 50%. The second phase offers an additional 30% difference reduction—to our knowledge, this work is the first to make extensive use of Boolean matching to reduce the logic difference.

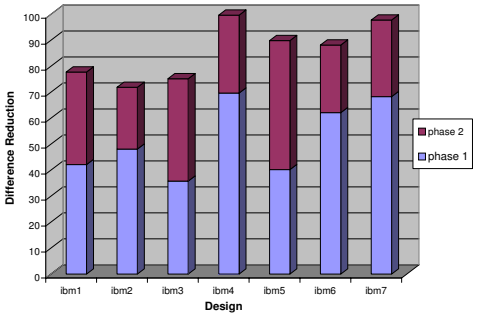
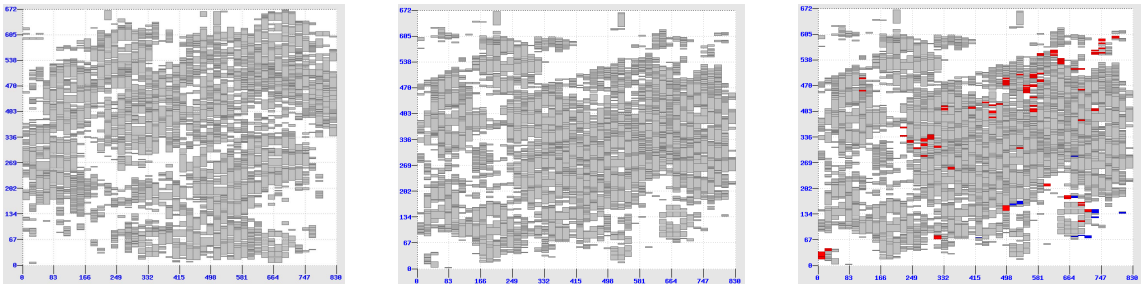


Figure 17: Difference model reduction through phases I and II of DeltaSyn.

Table 1 shows that our difference model disturbs only 3% of logic on average, which is important for preserving the design effort. Figure 18 gives a breakdown of the time



(a) (b) (c)

Figure 16: Placement illustration of (a) the ECO model placed from scratch, (b)the original model, and (c) incremental placement on the difference model stitched. Blue indicates deleted gates, red indicates newly added gates.

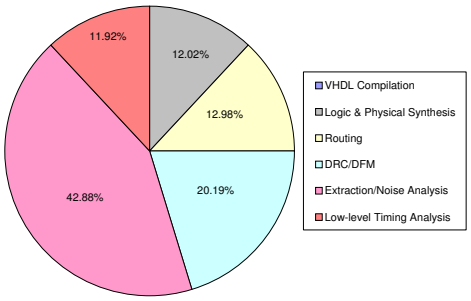


Figure 18: Percentage of time spent in various parts of the design flow [14]. The VHDL compilation step is too small to be visible.

spent in various parts of the design flow. This is derived from an average of 44 circuits that went through the complete design flow [14]. The first point to note in this figure is that the only step that DeltaSyn repeats is the VHDL compilation step which takes 0.01% of the entire design flow (not visible on the piechart). Despite some additional overhead, DeltaSyn allows designers to essentially skip the rest of the process on the unperturbed parts of the design. To demonstrate this, we have embedded DeltaSyn into the PDSRTL physical synthesis and design system [20] which performs incremental placement and optimization only on gates in the difference model (leaving all other gates and latches fixed). Table 2 indicates that the runtime decreases drastically for all available benchmarks. In addition, the total slack generally improves or stays close to the same. In the case of ibm2, the fanout of a particular gate in the logic difference increased drastically and disturbed timing. We confirmed that the electrical correction step returns the slack to its original value.

Figure 16 shows an example of incremental placement enabled by DeltaSyn. The original model and the final model (with the difference model stitched in) look very similar while the entirely re-placed ECO model appears significantly different. Preserving the placement generally has the effect of preserving wire routes and also maintaining slack. In summary, results indicate that by isolating the boundaries of logic change, DeltaSyn is able to automatically identify changes which leave a large portion of the design unperturbed through the design flow.

While the results of DeltaSyn are promising, there is still room for improvement. For instance, DeltaSyn may output a larger-than-optimal difference model when a small change occurs immediately before a large and restructured fanout branch. Since the subcircuit matching avoids crossing fanout boundaries, the matching algorithm will tend to stop at the fanout stem. Ongoing work includes restarting subcircuit matching at other "similar" points in the circuit, as identified by structural or functional hashing.

5 Conclusions

In this paper, we presented DeltaSyn, a method for identifying a minimal set of changes in logic, known as the logic difference, to implement an incremental change to the design specification. DeltaSyn out-performs previously known approaches for logic ECO synthesis in runtime and scalability. Results show that DeltaSyn reduces the logic difference by an average of 80% as compared to previous methods. Further, typical ECOs were processed by reusing an 97% of existing logic, on average. Future work involves additional methods of difference reduction and extensions to ECOs on sequential logic.

References

[1] A. Abdollahi, M. Pedram, "Symmetry Detection and Boolean Matching Utilizing a Signature-Based Canonical Form of Boolean Functions," *TCAD* vol. 27, no. 6, June, 2009.

[2] C.J. Alpert, C. Chu, P.G. Villarrubia, "The Coming of Age of Physical Synthesis," *ICCAD* 2007, pp. 246-249.

[3] D. Brand, A. Drumm, S. Kundu, P. Narain, "Incremental Synthesis," *ICCAD* 1994, pp. 14-18.

[4] J. Cong and M. Sarrafzadeh, Incremental Physical Design, *ISPD* 2000, pp. 2000.

[5] K.-H. Chang, D. A. Papa, I. L. Markov, V. Bertacco, "Invers: An Incremental Verification System with Circuit Similarity Metrics and Error Visualization," *IEEE D&T* vol. 26, no. 2, March 2009, pp. 34-43.

[6] R. Goering, Cadence CTO: CAD Foundations Must Change, EETimes, April 11, 2006, http://www.eetimes.com/showArticle.jhtml?articleID=185300099

[7] R. Goering, "Xilinx ISE Handles Incremental Changes," EETimes, January 15, 2007. //http://www.eeproductcenter.com/embedded/review/showArticle.jhtml?articleID=196900852

[8] V.N. Kravets, P. Kudva, "Implicit Enumeration of Structural Changes in Circuit Optimization," *DAC* 2004, pp. 438-441.

[9] A. Kuehlmann, V. Paruthi, F. Krohm, and M. Ganai, "Robust Boolean Reasoning for Equivalence Checking and Functional Property Verifcation," *TCAD* 2002, vol. 21, no. 12, pp. 1377-1394.

[10] Y.-L. Li, J.-Y. Li, W.-B. Chen, "An Efficient Tile-Based ECO Router Using Routing Graph Reduction and Enhanced Global Routing Flow," *TCAD*, vol. 26, no. 2, Feb. 2007, pp. 345-358.

[11] C-C. Lin, K-C. Chen, M. Marek-Sadowska, "Logic Synthesis for Engineering Change," *TCAD*, vol. 18, no. 3, March 1999, pp. 282-292.

[12] A. C. Ling, S. D. Brown, J. Zhu, S. Safarpour, "Towards Automated ECOs in FPGAs," *Proc. International Symposium on Field Programmable Gate Arrays* 2009, pp. 3-12.

[13] A. Mishchenko, S. Chatterjee, R. Jiang, R. Brayton, "FRAIGs: A Unifying Representation for Logic Synthesis and Verifcation," *ERL Technical Report, Berkeley* http://www.eecs.berkeley.edu/ alanmi/publications/.

[14] P. Osler, *Personal Communication*, Design Closure Engineer, IBM Systems & Technology Group, May 11, 2009.

[15] J.A. Roy, I.L. Markov, "ECO-System: Embracing the Change in Placement," *TCAD*, vol. 26, no. 12, Dec. 2007, pp. 2173-2185.

[16] S. Safarpour, H. Mangassarian, A. G. Veneris, M. H. Liffiton, and K. A. Sakallah, "Improved Design Debugging using Maximum Satisfiability, *FMCAD* 2007, pp. 1319.

[17] T. Shinsha, T. Kubo, Y. Sakataya, and K. Ishihara, "Incremental Logic Synthesis Through Gate Logic Structure Identification, *DAC* 1986, pp. 391397.

[18] L. Stok, D.S. Kung, D. Brand, et al., "BooleDozer: Logic Synthesis for ASICs," *IBM Journal of Research and Development*, vol. 40, no. 4, July 1996.

[19] G. Swamy, S. Rajamani, C. Lennard, R. K. Brayton, "Minimal Logic Re-synthesis for Engineering Change," *ISCAS* 1997, pp. 1596-1599.

[20] L. Trevillyan, D. Kung, R. Puri, L. N. Reddy, and M. A. Kazda, "An Integrated Environment for Technology Closure of Deep-Submicron IC Designs," *IEEE Design and Test*, vol. 21, no. 1, Jan-Feb 2004, pp. 1422.

[21] C. Visweswariah, K. Ravindran, K. Kalafa, S.G. Walker, S. Narayan, "First-Order Incremental Block-Based Statistical Timing Analysis," *DAC* 2004, pp. 331-336.

[22] J. Werber, D. Rautenback, C. Szegedy, "Timing Optimization by Restructuring Long Combinatorial Paths," *ICCAD 2007* pp. 536-543.

[23] H. Xiang, K.-Y. Chao, M.D.F Wong, "An ECO Routing Algorithm for Eliminating Coupling-Capacitance Violations," *TCAD*, vol. 25, no. 9, Sept. 2006, pp. 1754- 1762.

[24] Q. Zhu, N. Kitchen, A. Kuehlmann, A. L. Sangiovanni-Vincentelli, "SAT sweeping with Local Observability Don't-Cares," *DAC* 2006 pp. 229-234.