Chapter 12
Logic Difference Optimization for Incremental Synthesis

Smita Krishnaswamy, Haoxing Ren, Nilesh Modi, and Ruchir Puri

Abstract During the IC design process, functional specifications are often modified late in the design cycle, often 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 tool and methodology for generating a highly optimized logic difference between a modified high-level specification and an implemented design. DeltaSyn has the ability to locate similar logic in the original design which can be reused to realize the modified specification through several analysis techniques that are applied in sequence. The first phase employs fast functional and structural analysis techniques to identify equivalent signals between the original and the modified circuits. The second phase uses a novel topologically-guided dynamic matching algorithm to locate reusable portions of logic close to the primary outputs. The third phase utilizes functional hashing to locate similar chunks of logic throughout the remainder of the circuit. Experiments on industrial designs show that, together, these techniques successfully implement incremental changes while preserving an average of 97% of the pre-existing logic. Unlike previous approaches, bit-parallel simulation and dynamic programming enable fast performance and scalability. A typical design of around 10K gates is processed and verified in about 200 s or less.

12.1 Introduction and Background

As the IC industry matures, it becomes common for existing designs to be modified incrementally. Since redesigning logic involves high expenditure of design effort and time, previous designs must be maximally re-utilized whenever possible.

S. Krishnaswamy (✉)
IBM TJ Watson Research Center, Yorktown Heights, NY
e-mail: skrishn@us.ibm.com

This work is based on an earlier work: DeltaSyn: an efficient logic difference optimizer for ECO synthesis, in Proceedings of the 2009 international Conference on Computer-Aided Design, ISBN:978-1-60558-800-1 (2009) © ACM, 2009. DOI= http://doi.acm.org/10.1145/1687399.1687546

S.P. Khatri, K. Gulati (eds.), *Advanced Techniques in Logic Synthesis, Optimizations and Applications*, DOI 10.1007/978-1-4419-7518-8_12,

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, 22], placement [17], routing [25], timing analysis [22], and verification [4] have made incremental 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 incremental synthesis methodology that is able to quickly derive a small set of changes in logic to handle incremental updates is necessary.

Prior work on incremental synthesis tends to focus on small *engineering change orders* (ECOs). Such methods primarily fall into two categories. The first category consists of purely functional techniques which attempt to isolate point changes and perform in-place rectification. Such techniques can be unscalable [12, 21] due to the use of complex BDD manipulation and laborious analysis. Further, they may simply fail to identify multiple point changes and changes that cannot be easily isolated as originating at specific points in a circuit due to logic restructuring. The second category of methods is heavily reliant on structural correspondences [3, 19] and can result in large difference models that disrupt much of the existing design when such correspondences are unavailable.

In this chapter, we present DeltaSyn, a method to produce a synthesized delta or the *logic difference* between an RTL-level *modified specification* and an original implemented design. As illustrated in Fig. 12.1, DeltaSyn combines both functional

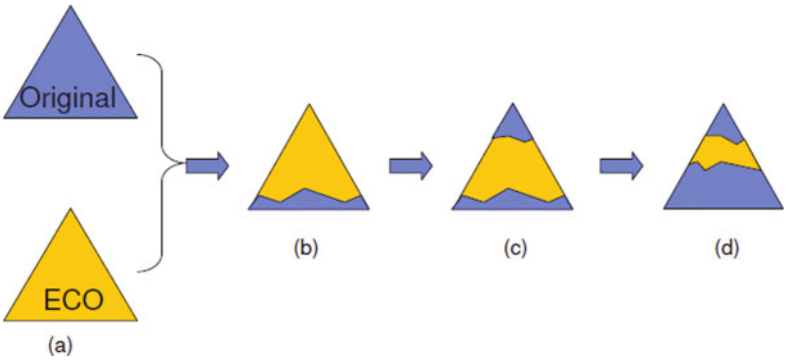


Fig. 12.1 The main phases of DeltaSyn: (**a**) the original design and the modified specification are given as inputs to DeltaSyn, (**b**) functional and structural equivalences forming the input-side boundary of the changes are identified, (**c**) matching subcircuits which form the output-side boundary of the changes are located and verified, (**d**) further reductions are identified through functional hashing

and structural analysis 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. Phase III finds further logic for reuse through a novel functional hashing technique. DeltaSyn allows designers to avoid most design steps including much of logic synthesis, technology mapping, placement, routing, buffering, and other back-end optimizations on the unchanged logic.

The main features of our method include:

- An efficient multi-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 functional hashing technique to enable wider use of matching.

A key advantage of our approach is that, unlike traditional ECO methodologies, we make no assumptions about the type or extent of the changes in logic. The remainder of the chapter is organized as follows. Section 12.2 describes previous work in incremental synthesis. Section 12.3 describes the overall flow of DeltaSyn. Sections 12.3.1 and 12.3.2 describe our equivalence-finding and subcircuit-matching phases of logic difference reduction, while Section 12.3.3 presents the functional-hashing phase of difference reduction. Section 12.4 presents empirical results and analysis. Section 12.5 concludes the chapter.

12.2 Previous Work

Recently, the focus of incremental design has been on changes to routing or placement [2, 11, 18]. However, there have been several papers dealing specifically with logic ECO. Authors of [3, 19] 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 lead to many changes to the design since such structural correspondences are hard to find in designs that undergo many transformations.

In contrast, the method from [12] 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. [13] present a maximum satisfiability (MAX-SAT) formulation similar to that of [18] 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, this method does not directly show how to correct the circuit. Deriving the correction itself can be a difficult problem – one that is circumvented by our method.

12.3 DeltaSyn

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

Definition 12.1 The *original model* is the original synthesized, placed, routed, and optimized design.

Definition 12.2 The *modified specification* is the modified RTL-level specification, i.e., the change order.

Definition 12.3 The *difference model* is a circuit representing the changes to the original model required to implement the modified 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 modified specification, and a list of corresponding primary outputs and latches, the objective of incremental synthesis is to derive a *difference model* that is minimal in the number of gates. We choose the minimal number of gates as our metric because the general procedure by which incremental synthesis occurs in the industry motivates the need to preserve as many gates as possible. Typically, when late-stage changes occur, the masks are already set for most metal layers. The changes are realized by rewiring spare gates in the top metal layer. Additionally, incremental placement and routing tools can optimize wire length and other physical concerns.

The new specification, generally written in an RTL-level hardware description language (such as VHDL), is compiled into a technology-independent form called the *modified model*. This step is relatively fast because the majority of the design time is spent in physical design including physical synthesis, routing, and analysis [Osler, P. Personal Communication (2009)] (see Fig. 12.17).

The circuits in Fig. 12.2 are used to broadly illustrate the three phases of our difference optimization. By inspection, it is clear that $f3$ and $f7$ are the only

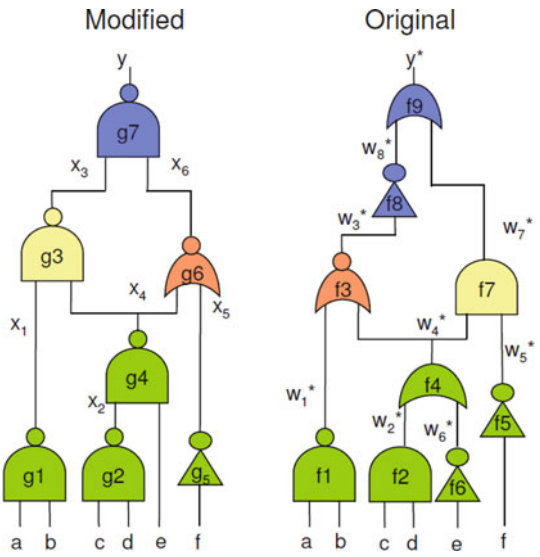


Fig. 12.2 Sample circuits to illustrate our method

differences between the two circuits. Although some of the local logic has undergone equivalent transformations (gates $g4$ 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 such examples.

DeltaSyn recognizes and isolates such changes as follows: Our first phase involves structural and functional equivalence checking. For the given example, the equivalences $x_1 \equiv w_1^*$, $x_4 \equiv w_4^*$, and $x_5 \equiv w_5^*$ are identified by these techniques. Our second phase is geared toward finding matching subcircuits from the primary outputs. Through a careful process of subcircuit enumeration and Boolean matching, the subcircuits consisting of $\{f8, f9\}$ from the original model and $\{g7\}$ are matched under the intermediate input mapping $\{(x_3, w_3^*), (x_6, w_7^*)\}$. This phase leaves $g3$ and $g6$ as the logic difference . However, in the third phase $f3$ and $g6$ are recognized as subcircuits performing the same functionality, therefore $f3$ can simply be rewired to realize $g6$. Therefore, the third phase leaves $g3$ as the optimized logic difference. The remainder of this section explains the algorithms involved in these steps.

12.3.1 Phase I: Equivalence-Based Reduction

Phase I is illustrated in Fig. 12.3. 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 modified models. Matches are found both structurally and functionally. Candidates for functional equivalence are identified by comparing simulation responses and verified using Boolean satisfiability (SAT).

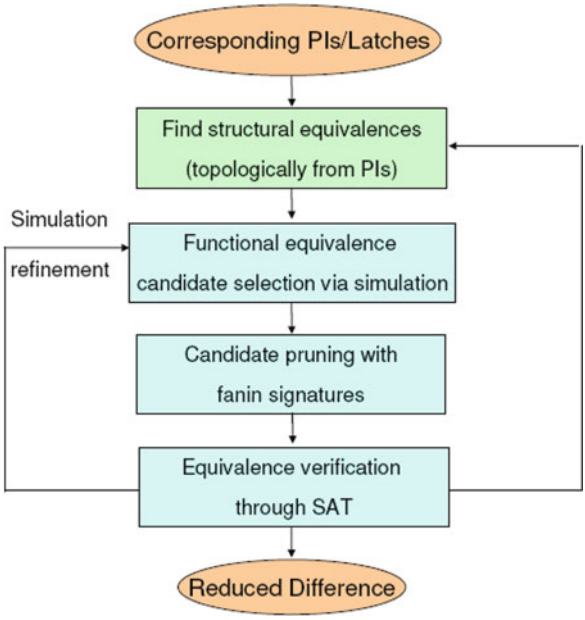


Fig. 12.3 Logic difference reduction through equivalence checking

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 12.1 In Fig. 12.4 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 [10, 15]. These candidates are further pruned by comparing a pre-computed *fanin signature* for each of these candidate signals. A fanin signature has a 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 12.2 In Figure 12.4, the same set of four random vectors are assigned to corresponding input and internal signals. The output responses to each of the inputs

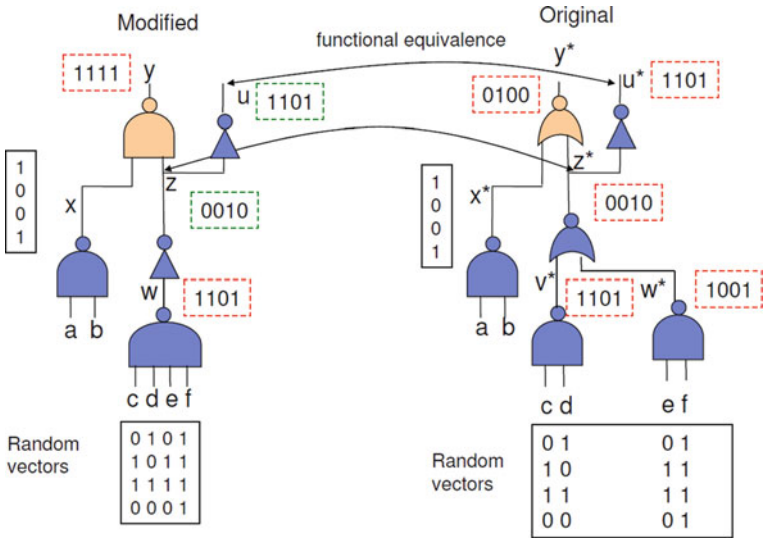


Fig. 12.4 Identifying structural and functional equivalences

are 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 candidate 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 [27], 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.

12.3.2 Phase II: Matching-Based Reduction

Phase II of DeltaSyn finds subcircuits that are functionally equivalent under some permutation of intermediate signals. Since incremental synthesis is intended to be used for 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 Fig. 12.2).

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 modified 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 *Boolean matching* . For our purposes, we actually find all such input orders instead of just one. While these problems are generally highly complex, we take advantage of two context-specific properties in order to effectively locate and match subcircuits:

1. Most of the modifications we encounter are small.
2. Many of the logic optimizations performed on the original implementation involve localized transformations that leave the global structure of the logic intact.

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 [9, 20, 22, 24]. While redundancy removal can be a non-local change, equivalent signals between the two circuits (despite 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 12.5 illustrates the main steps of subcircuit identification and matching. Candidate subcircuits are generated by expanding two corresponding outputs along their fanin cones. 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.

12.3.2.1 Subcircuit Enumeration

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

Definition 12.4 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 modified model, are enumerated in tandem. Figure 12.6 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 Fig. 12.6, the subcircuit initially

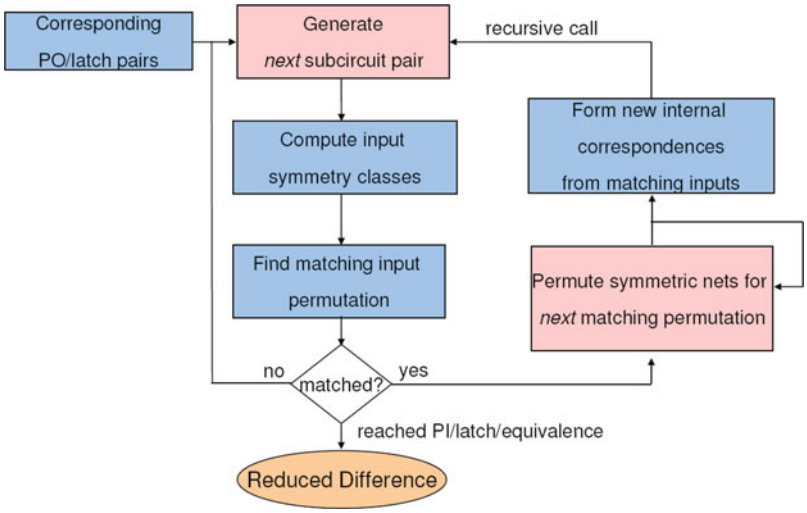


Fig. 12.5 Difference reduction through subcircuit matching

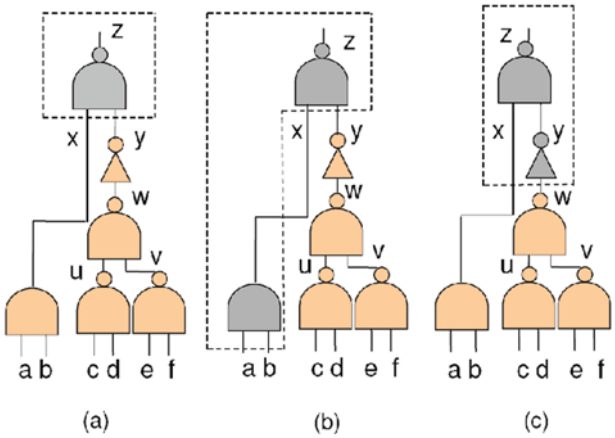


Fig. 12.6 Candidate subcircuit enumeration

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).

Pseudocode for subcircuit expansion and enumeration are shown in Fig. 12.7. A *subcirc_enum* structure (shown in Fig. 12.8) is instantiated for pairs of nets N, N^* where N is from the modified model and N^* is from the original model (starting

Fig. 12.7 Subcircuit pair enumeration algorithm

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

Fig. 12.8 The data structure for enumerated subcircuits

```
STRUCT subcirc_enum
  {
      N
      N*
      mod_queue
      orig_queue
      pair_history

      next_subcircuit_pair(C, C*)
      expand_subcircuit( subcircuit C, queue Q)
  }
```

from corresponding primary outputs). The *next_subcircuit_pair* method fills in the variables C and C^*. First, the *orig_queue* is popped. If the *orig_queue* is empty, then all the possible subcircuits in the original model have already been enumerated for a particular subcircuit C in the modified model. In this case, a new modified subcircuit is found by popping the *mod_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 *mod_queue* is also empty, then all possible pairs of subcircuits have already been enumerated for the pair of nets (N, N^*) and the process terminates.

12.3.2.2 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 12.5 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 12.6 Two inputs i_x and i_y of a function F are said to be *symmetric* with respect to 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 12.7 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 $\tau_F = \tau_{sym\_F[1]} \circ \tau_{sym\_F[2]} \circ \ldots \circ \tau_{sym\_F[n]}$. 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.

Theorem 12.1 *Given a matching permutation $\rho_{(F^*,F)}$, all other matching permutations $\pi_{(F^*,F)}$ can be derived by composing a symmetric permutation τ with $\rho_{(F^*,F)}$, that is, for some symmetric permutation τ:*

$$\pi_{(F^*,F)} = \rho_{(F^*,F)} \circ \tau$$

Proof 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 $\rho_{(F,F^*)} \circ \phi = \pi_{(F^*,F)}$. However, by definition of symmetry

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

By transitivity

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

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

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

and by definition of matching permutation:

$$\begin{aligned} &F^*(\rho_{(F,F^*)}(\phi(i_1)), \rho_{(F,F^*)}(\phi(i_2)), \rho_{(F,F^*)}(\phi(i_3))\ldots) \\ &= F^*(\rho_{(F,F^*)}(i_1), \rho_{(F,F^*)}(i_2), \phi_{(F,F^*)}(i_3)) = F \end{aligned}$$

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

Theorem 12.1 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.

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 Fig. 12.9.

We derive a matching permutation of F^* or determine that one does not exist through the algorithm shown in Fig. 12.11. In the pseudocode, instead of specifying

```
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
}
```

Fig. 12.9 Computing symmetry classes

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 Fig. 12.10. 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.

```
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

}
```

Fig. 12.10 Compute a matching permutation order

Fig. 12.11 Enumerating matching input orders

```
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)
}
```

The remaining matching permutations are derived by enumerating symmetric permutations as shown in Fig. 12.11. 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 modified model C. Figure 12.12 illustrates that although two subcircuits can be equivalent under different input orders, the "correct" order leads to larger matches upstream.

Note that the discussion in this section can be applied to finding all matching permutations under negation-permutation-negation (NPN) equivalence, by simply negating the inputs appropriately at the outset as in [1]. This involves choosing the polarity of each input variable that maximizes its cofactor minterm count and using that polarity in deriving matching permutations. In other words, for a subcircuit C realizing function F, if $|F(i_0 = 0, \ldots)| > |F(i_0 = 1, \ldots)|$ then input i_0 is used in its negated form and the remainder of the analysis follows as discussed above. In practice, this helps in increasing design reuse by ignoring intermediate negations.

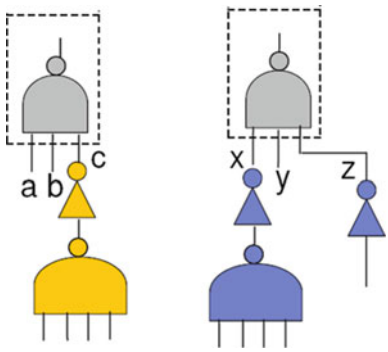


Fig. 12.12 Although the single-gate subcircuits in the boxes have completely symmetric inputs, the input ordering (c, b, a) leads to a larger upstream match than (a, b, c)

12.3.2.3 Subcircuit Covering

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

Definition 12.8 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 nets (N, N^*) can lead to different subcovers as shown in Fig. 12.13. Once the subcircuit D of the original and D^* is generated through subcircuit enumeration algorithm of Fig. 12.8, the algorithm of Fig. 12.10 finds an input ordering under which they are functionally equivalent. Figure 12.13 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.

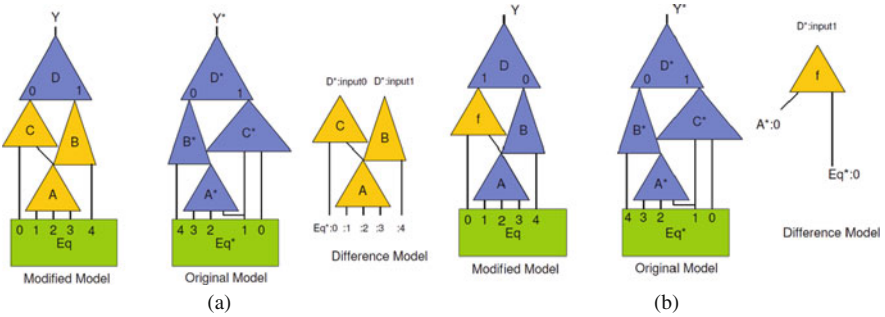


Fig. 12.13 Snapshots of subcircuit covering: (**a**) Subcover induced by input ordering $D^*(0, 1)$ on original model and resulting difference (**b**) Subcover induced by input ordering $D^*(1, 0)$, and the resulting (smaller) logic difference

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 12.14 shows the algorithm to compute the *optimal subcover* . The algorithm starts by enumerating all subcircuit pairs (see Fig. 12.6) and matching permutations (see Fig. 12.11) 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, C^* 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

```
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*))
}
```

Fig. 12.14 The recursive subcircuit covering algorithm

$opt\_match(i_1, j_1), opt\_match(i_2, j_2) \ldots$ 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 modified model has higher fanout than a net in the original model then different subcovers may correspond the modified net with different original nets. When such conflicts occur, the correspondence that results in the larger subcover is retained.

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 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 modified model are removed from the difference model.

12.3.3 Phase III: Functional Hashing-Based Reduction

In the previous section, we used a topologically-guided method to match regions of the circuit which can be reused starting from the primary outputs. However, it is possible to search for reusable logic at a finer level of granularity. Often, different logic functions have subfunctions in common. Further, certain logic optimizations,

such as rewiring, can cause global changes in connectivity while still maintaining logic similarity. For instance, if two output pins were swapped, the method of Phase-II would fail because it searches based on topological connectivity. To address this issue, we present a method that searches for similar logic throughout the circuit and not just in topologically relevant.

This method proceeds by traversing all of the nets in the unmatched portions of the original and modified designs and hashing subcircuit functions of limited size at the fanin cones of the nets. The functions within the original design that hash to the same key as the modified design are candidates for reuse. These candidates are then verified for Boolean matching using the method of Fig. 12.10 and then the matches are dynamically extended using the recursive subcircuit covering algorithm of Fig. 12.16. In other words, the hashing enables us to restart the subcircuit coveri ng algorithm from new, promising locations. Additionally, reusable modules, such as adders or priority muxes, which may be available in the logic, can be appropriated for use in realizing the changed specification.

In previous literature, functional hashing has been used in logic *rewriting* [14] to hash 4-input cuts such that cuts realizing identical functionality can be replaced by each other. However, the representative member of the corresponding NPN equivalence class is simply referenced from an exhaustive list. It is only possible to exhaustively list 4-input functions, as classified by [16] as the number of Boolean functions of five or more gets prohibitively large. Authors of [5] use another method of functional hashing, where they derive a signature for 3- and 4-input cuts. However, that method does not scale to larger circuits either.

Here, we propose an efficient key for the hash function based on easy-to-compute functional characteristics such that likely Boolean matches are placed in the same hash bucket. These functional characteristics include a subset of what is computed in order to assess a full Boolean match.

Definition 12.9 Given a Boolean function $F(i_1, i_2, \ldots, i_n)$, the *matching key* $K(F, k)$ is the $(k + 3)$-*tuple*,

$$< N, S, F_0, F_1, \ldots, F_k >$$

where

- N is the number of input symmetry classes in the function.
- S is a sequence containing the sizes of the symmetry classes in sorted order.
- F_0 is the minterm count of the function realized by the subcircuit.
- F_j for any $0 \leq j < k$ is a sequence containing the kth order positive cofactor or negative minterm counts (whichever is greater) in sorted order.

Note that if $k = n$, then $K(F, k)$ completely specifies the function K. However, in practice, one rarely needs more than $k = 2$ to sufficiently differentiate most functions. Since we do not need complete differentiation through hashing, we use $k = 1$. This observation has been corroborated by results in [1] where it is reported that $k = 2$ is enough to determine the existence of a Boolean match between two

functions in most cases. The hash values simply consist of the boundaries of the subcircuit in question.

Figure 12.15 shows the overall matching algorithm using functional hashing . After the remainder of the original circuit is functionally hashed, the modified circuit is traversed and the largest matches starting at each net are found and stored in the map $BestMatch$. At this point the largest possible matches are known, and we essentially have an optimization version of the *set cover* problem, i.e., we want a cover of the largest number of gates in a circuit. Set cover is a well-known NP-complete problem, whose best-known approximation algorithms simply pick the largest covers greedily [8]. We follow the same approach in choosing a cover. We note that this finer granularity of gate preservation will enable farther-reaching changes to be incorporated into the incremental synthesis flow especially as the synthesis of larger and larger blocks begins to be automated.

```
FIND_FUNCTIONAL_MATCHES(C_orig, C_eco){
{
        foreach(unmatched net N* ∈ C_orig){
                while(N* _enum.next_subcircuit(C*)){
                        F* = compute_truth_table(C*)
                        K(F*,k) = compute_hash_key(F*)
                        H[K(F*,k)] = C
                }
        }
        foreach(unmatched net N ∈ C_eco){
                while(N_enum.next_subcircuit(C)){
                        F = compute_truth_table(C)
                        K(F,k) = compute_hash_key(F)
                        for(0 <= i < |H[K(F,k)]|){
                                C_cov(N) = compute_cover(N,H[K(F,k)][i])
                                if(|C_cov(N)| > |BestMatch[N]|){
                                        BestMatch[N] = C_cov
                                }
                        }
                }
        }
}
```

Fig. 12.15 Functional matching algorithm

12.4 Empirical Validation

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, 1.6 GHZ processors with 12 GB RAM. Our code is written in C++ and compiled with GCC 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 [20] 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 12.1. 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 12.1 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.

Table 12.2 shows results on larger changes. These may be categorized as *incremental synthesis* benchmarks rather than traditional ECO benchmarks. On such cases, we measured the results of all three of the phases, and noted that the addition of a third phase offers an extra 8% reduction in delta size through the reuse of common subfunctions in logic. Note that the reduction numbers only reflect the results of DeltaSyn and not pre-processing optimizations.

Table 12.1 DeltaSyn statistics on IBM ECO benchmarks

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 12.2 DeltaSyn statistics on IBM incremental synthesis benchmarks. Compares two-phase difference reduction with three-phase difference reduction

Design	No. gates	Cone size	Diff. model size	New diff. model size	2-Phase runtime CPU(s)	3-Phase runtime CPU(s)	% 2-Phase diff. reduced	% 3-Phase diff. reduced
ibm14	7439	841	149	34	82.61	242.87	82.28	95.95
ibm15	4848	1679	447	169	24.77	29.27	73.38	89.93
ibm16	12681	4439	1310	584	179.13	474.86	70.49	86.84
ibm17	4556	510	12	9	23.93	22.58	97.65	98.23
ibm18	8711	1547	177	121	3.71	23.42	88.55	92.17
ibm19	3200	304	89	80	0.73	21.61	70.72	73.68
ibm20	5224	58	13	12	28.86	36.22	7.58	79.31
ibm21	6548	1910	429	261	190.82	266.69	77.54	86.33
ibm22	547	77	20	13	0.26	0.73	74.03	83.11
ibm23	8784	1299	249	174	13.93	85.74	80.83	86.61
Avg.							79.31	87.22

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, 12]. DeltaSyn processes all benchmarks in 211 or fewer seconds. The more (global) structural similarity that exists between the modified model and the original model, the faster DeltaSyn performs. For instance, ibm12 is analyzed in less than 1 s because similarities between the implemented circuit and the modified model allow for the algorithm in Fig. 12.14 to stop subcircuit enumeration (i.e., stop searching for potential matches) and issue recursive calls frequently. Any fast logic optimizations that bring the modified model structurally closer to the original model can, therefore, be employed to improve results. Figure 12.16 shows the relative percentages of difference model size reduction achieved by our three phases. The first phase reduces the logic difference by about 50%. The second phase offers an additional 30% difference reduction. The third phase offers an additional 8% of reduction on average.

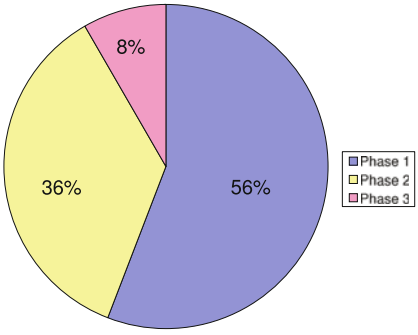


Fig. 12.16 Difference model reduction through phases I, II, III of DeltaSyn

Table 12.1 shows that our difference model disturbs only 3% of logic on average, which is important for preserving the design effort. Figure 12.17 gives a breakdown of the time spent in various parts of the design flow. This is derived from an

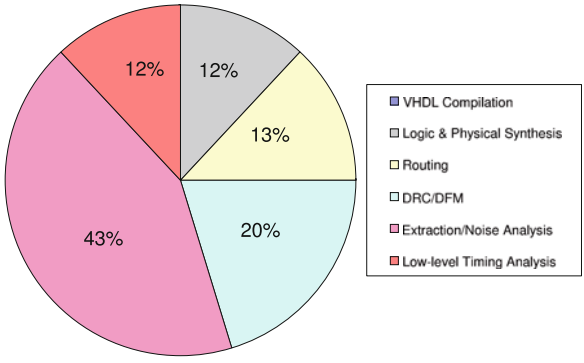


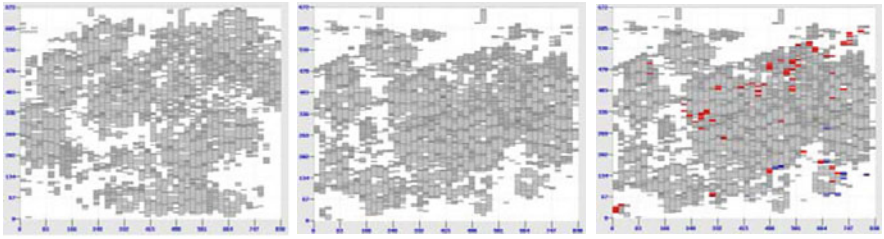
Fig. 12.17 Percentage of time spent in various parts of the design flow [Osler, P, Personal Communication (2009)]. The VHDL compilation step is too small to be visible

average of 44 circuits that were through the complete design flow [Osler, P, Personal Communication (2009)]. 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 pie chart). 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 [22] which performs incremental placement and optimization only on gates in the difference model (leaving all other gates and latches fixed). Table 12.3 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 12.18 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 replaced modified model appears significantly different. Preserving the placement generally has the effect of preserving wire routes and also maintaining slack. In summary, DeltaSyn is able to automatically identify changes which leave a large portion of the design unperturbed through the design flow.

Table 12.3 PDSRTL [22] runtime and slack comparison between incremental design and complete redesign

Runtime (s)			% Runtime	% Slack
Design	Entire Design	Difference	Decrease	Increase
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



(a) (b) (c)

Fig. 12.18 Placement illustration of (**a**) the modified 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

12.5 Chapter Summary

In this chapter, we presented DeltaSyn, a method that analyzes an original and a modified design to maximize the design reuse and design preservation. DeltaSyn uses three phases of analysis in order to find redundant and usable subcircuits in logic. These phases use a variety of techniques such as functional equivalence checking, recursive topologically guided Boolean matching, and functional hashing. Results show that DeltaSyn reduces the logic difference by an average of 88% as compared to previous methods. Further, typical specification changes were processed by reusing an 97% of existing logic, on average. Future work involves extensions to handle changes in sequential logic.

References

1. Abdollahi, A., Pedram, M.: Symmetry detection and Boolean matching utilizing a signature based canonical form of Boolean functions. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **27**(6), 1128–1137 (2009)
2. Alpert, C., Chu, C., Villarrubia, P.: The coming of age of physical synthesis. In: Proceedings of the International Conference on Computer-Aided Design, San Jose, CA, pp. 246–249. (2007)
3. Brand, D., Drumm, A., Kundu, S., Narain, P.: Incremental synthesis. In: Proceedings of the International Conference on Computer-Aided Design, San Jose, CA, pp. 14–18. (1994)
4. Chang, K.H., Papa, D.A., Markov, I.L., Bertacco, V.: Invers: An incremental verification system with circuit similarity metrics and error visualization. IEEE Design and Test Magazine **26**(2), 34–43 (2009)
5. Ganai, M., Kuehlmann, A.: On-the-fly compression of logical circuits. In: Proceedings of the International Workshop on Logic Synthesis, Dana Point, CA, (2000)
6. Goering, R.: CAD foundations must change. EETimes (2006)
7. Goering, R.: Xilinx ISE handles incremental changes. EETimes (2007)
8. Kleinberg, J., Tardos, E.: Algorithm Design. Addison Wesley (2005)
9. Kravets, V., Kudva, P.: Implicit enumeration of structural changes in circuit optimization. In: Proceedings of the Design Automation Conference, San Diego, CA, pp. 438–441. (2004)
10. Kuehlmann, A., Paruthi, V., Krohm, F., Ganai, M.: Robust Boolean reasoning for equivalence checking and functional property verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **21**(12), 1355–1394 (2002)
11. Li, Y.L., Li, J.Y., Chen, W.B.: An efficient tile-based eco router using routing graph reduction and enhanced global routing flow. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **26**(2), 345–358 (2007)
12. Lin, C.C., Chen, K.C., Marek-Sadowska, M.: Logic synthesis for engineering change. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **18**(3), 282–292 (1999)
13. Ling, A.C., Brown, S.D., Zhu, J., Safarpour, S.: Towards automated ECOs in FPGAs. In: Proceedings of the International Symposium on FPGAs, Monterey, CA, pp. 3–12. (2009)
14. Mishchenko, A., Chatterjee, S., Brayton, R.: Dag-aware AIG rewriting: A fresh look at combinational logic synthesis. In: Proceedings of the Design Automation Conference, San Francisco, CA, pp. 532–536. (2006)
15. Mishchenko, A., Chatterjee, S., Jiang, R., Brayton, R.: Fraigs: A unifying representation for logic synthesis and verification. ERL Technical Report, EECS Department, UC Berkeley, March 2005.
16. Muroga, S.: Logic Design and Switching Theory, John Wiley, New York (1979)
17. Osler, P.: Personal communication (2009)

18. Roy, J., Markov, I., Eco-system: Embracing the change in placement. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **26**(12), 2173–2185 (2007)
19. Safarpour, S., Mangassarian, H., Veneris, A.G., Liffiton, M.H., Safarpour, S., Mangassarian, H., Veneris, A.G., Liffiton, M.H.: Improved design debugging using maximum satisfiability. In: Proceedings of Formal Methods in Computer-Aided Design, Austin, TX, pp. 13–19. (2007)
20. Shinsha, T., Kubo, T., Sakataya, Y., Ishihara, K.: Incremental logic synthesis through gate logic structure identification. In: Proceedings of the Design Automation Conference, Las Vegas, NV, pp. 391–397. (1986)
21. Stok, L., Kung, D.S., Brand, D., Drumm, A.D., Sullivan, A.J., Reddy, L.N., Hieter, N., Geiger, D.J., Chao, H.H., Osler, P.J.: Booledozer: Logic synthesis for ASICs. IBM Journal of Research and Development **40**(4), 407–430 (1996)
22. Swamy, G., Rajamani, S., Lennard, C., Brayton, R.K.: Minimal logic re-synthesis for engineering change. In: Proceedings of the International Symposium on Circuits and Systems, Hong Kong, pp. 1596–1599. (1997)
23. Trevillyan, L., Kung, D., Puri, R., Reddy, L.N., Kazda, M.A.: An integrated environment for technology closure of deep-submicron IC designs. IEEE Design and Test Magazine **21**(1), 14–22 (2004)
24. Visweswariah, C., Ravindran, K., Kalafa, K.,Walker, S., Narayan, S.: First-order incremental block-based statistical timing analysis. In: Proceedings of the Design Automation Conference, San Diego, CA, pp. 331–336. (2004)
25. Werber, C., Rautenback, D., Szegedy, C.: Timing optimization by restructuring long combinatorial paths. In: Proceedings of the International Conference on Computer-Aided Design, San Jose, CA, pp. 536–543. (2007)
26. Xiang, H., Chao, K.Y., Wong, M.: An ECO routing algorithm for eliminating coupling capacitance violations. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **25**(9), 1754–1762 (2006)
27. Zhu, Q., Kitchen, N., Kuehlmann, A., Sangiovanni-Vincentelli, A.L.: SAT sweeping with local observability don't-cares. In: Proceedings of the International Conference on Computer-Aided Design, San Jose, CA, pp. 229–234. (2006)