Intuitive ECO Synthesis for High Performance Circuits

Haoxing Ren[1], Ruchir Puri[1], Lakshmi Reddy[2], Smita Krishnaswamy[5], Cindy Washburn[2], Joel Earl[3], Joachim Keinert[4]
[1] IBM T. J. Watson Research Center, Yorktown Heights, NY, USA
[2] IBM System and Technology Group, Fishkill, NY, USA
[3] IBM System and Technology Group, Rochester, MN, USA
[4] IBM System Technology Group, Boeblingen, Germany
[5] Columbia University, New York, NY, USA

ABSTRACT
In the IC industry, chip design cycles are becoming more compressed, while designs themselves are growing in complexity. These trends necessitate efficient methods to handle late-stage engineering change orders (ECOs) to the functional specification, often in response to errors discovered after much of the implementation is finished. Past ECO synthesis algorithms have typically treated ECOs as functional errors and applied error diagnosis techniques to solve them. However, error diagnosis methods are primarily geared towards finding a single change, and moreover, tend to be computationally complex. In this paper, we propose a unique methodology that can systematically incorporate human intuition into the ECO process. Our methodology involves finding a set of directly substitutable points known as *functional correspondences* between the original implementation and the new specification by using name-preserving synthesis and *user hints,* to diminish the size of the ECO problem. On average, our approach can reduce the size of logic changes by 94% from those reported in current literature. We then incorporate our logic ECO changes into an incremental physical synthesis flow to demonstrate its usability in an industrial setting. Our ECO synthesis methodology is evaluated on high-performance industrial designs. Results indicate that post-ECO worst negative slack (WNS) improved 14% and total negative slack (TNS) improved 46% over pre-ECO.

Keywords
Engineering Change Order, Logic Synthesis, Physical Synthesis

1. INTRODUCTION
In IC design, ECO synthesis refers to the process of realizing late-stage functional changes to a design, by directly and minimally modifying the implementation, instead of re-invoking the entire design process from scratch. As a result, design cost is reduced and design stability is maintained. Producing a high-quality implementation of a high-performance design is a painstaking process that involves simultaneously optimizing numerous timing, power, and reliability metrics [4]. An ECO synthesis process can aid in maintaining the stability of such design metrics because the changes inserted are designed to be minimally invasive. On the manufacturing end, processing the new logic from scratch may result in discarding the multi-million dollar front-end-of-line (FEOL) mask sets. With ECO synthesis, the logic changes can often fit into spare cells provided on the chip, and the wiring changes are implementable on cheaper back-end-of-line (BEOL) masks.

Existing logic ECO methods can be classified into two major categories: error-detection-and-correction-based approaches, and matching-based approaches. Error-detection-and-correction-based approaches [3][10][11][12][29] view an ECO as an error correction problem and borrow techniques from literature in diagnosis and verification. The matching-based methods [2][13][18][1] acknowledge that ECOs are small changes, and that a large part of the design remains equivalent. After matching these equivalent parts, the differences can be extracted out.

We note that the key task in logic ECO synthesis is the determination of points in logic at which minimal substitutions can be made, in order to rectify the difference between the ECO and original logic. We call such points the *output-side boundary* of the changes. This boundary is difficult to establish because there are no longer any functionally equivalent points to guide the methods. In lieu of functional equivalence, [1] uses a laborious method of recursive matching that is severely limited due to the difficulty of both subcircuit enumeration and Boolean matching. [10] uses MAX-SAT to find changes, but synthesizing the substitution logic can still be a challenging task.

Despite these proposals, ECO synthesis is often done manually, since designers seem able to intuitively derive smaller changes than the respective tools in many cases. In this work, we study the process by which designers manually synthesize ECOs and incorporate their methods into an efficient industrial-strength design flow. We significantly reduce the complexity of finding the output-side boundary of the change by observing that if we generate gate-level netlists from both the original and ECO specifications, we can often find pairs of points that directly form a *functional correspondence* between the ECO and original netlists. Thus, in our approach, error detection and correction are performed simultaneously. In this paper we:

1. Formulate the notion of a functional correspondence between the ECO and original designs, which form the output-side boundary.
2. Provide a method for the generation of pairs of points in logic that constitute functional correspondences.
3. Provide a general method that can incorporate guessed or otherwise generated correspondences, to derive verifiable full functional correspondences that result in small logic changes.
4. Incorporate our ECO methodology into an industrial physical synthesis flow that places the gates produced by logic ECO synthesis and optimizes them to improve timing.

978-3-9815370-0-0/DATE13/©2013 EDAA

The rest of this paper is organized as follows: Section 2 describes previous work in this area, Section 3 provides background in equivalence checking and previous ECO methods, Section 4 and 5 present the new logic ECO process; Section 6 describes our ECO physical synthesis flow and presents the empirical results on a set of industry designs, and, Section 7 concludes the paper.

2. PREVIOUS WORK

Several previous papers propose error-detection-and-correction-based approaches [3][10][11][12][29]. The error detection step tries to find signals that can correct the original netlist in order to make it equivalent to the new functional specification. Then, the error correction step constructs substitution circuits to correct the identified signals. Existing methods in this category differ in how these two steps are conducted. The early work [3] uses BDD-based Boolean quantification based on a formula originally proposed in [21] to identify single-signal errors. Error correction is then done by functional decomposition. [12] uses a SAT formulation originally proposed in [20] to detect signals to be replaced, and then constructs the replacement logic with nearby signals using a set of error signatures. The authors in [10] formulate the signal error detection problem as MAX-SAT problem originally proposed in [14] and then derived the function based on SAT based function dependency check [22] and functional decomposition targeted for FPGA [23]. Recent work in [12] proposes a modified SAT-based fault identification procedure built on the proposals of [12][20] to include the potential fix via MUX modeling. Solving the SAT problem identifies and corrects the error at the same time. This technique also relies on extensive simulation to narrow down the scope of searching and SAT interpolation to construct ECO logic, similar to [10], if the MUX modeling fails. One difficulty with the error detection and correction based approaches are the difficulty of locating the signals if there are multiple points changed. Other difficulties include building the right support set, and synthesizing the correcting logic. Latest works [11][29] propose to use SAT interpolation technique to construct partial fixes for ECO and claim good results on both single and multiple error circuits.

The matching-based methods [2][13][18][1] attempt to find portions of the design to re-use. The pioneering work in [2] proposes to directly reconnect the ECO logic with the original netlist on key signals, where the ECO logic is either functionally or structurally correspondent to the original netlist. Authors of [2] propose an ATPG-based approach to check functional equivalence. The authors of [13][18] extend the work in [2] by name-based matching between the ECO logic and original netlist, and BDD-based equivalence checking.

After logic ECO, the new logic gates have to be placed and optimized using ECO physical synthesis. ECO physical synthesis is an incremental CAD paradigm [8] [17], where design perturbation is an optimization criterion, in addition to the conventional wirelength, timing, and power constraints. Research in ECO physical synthesis is geared towards making each physical synthesis step (placement, buffer insertion, gate sizing) ECO aware. The authors of [6] present an ECO placement system, [15] introduce an ECO buffer insertion algorithm using spare cells, [26] presents a method to perform technology remapping with spare cells, and [16] introduces an ECO routing algorithm. Incremental legalization schemes such as [7][9] can also be used in ECO placement.

3. BACKGROUND

In this paper we build on the framework of DeltaSyn [1], which utilizes a two-phase process to reduce the logic difference or *delta,* between the ECO and original designs. As shown in Figure 1, the first phase finds functional equivalences between original netlist and ECO logic from the primary inputs forward, forming the input-side boundary of the changes. Functional equivalences can be generated by sat-sweeping as in [27], which is summarized in Figure 2.

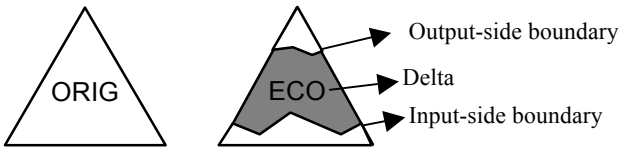

Figure 1. Summary of the DeltaSyn Method from [1].

```
find_functional_equivalence( netlist A, netlist B)
{
    simulate A and record signature(i) on each net i
    create hash table for signature(i) on A
    connect primary inputs of A and B together
    simulate B and record signature(i) on each net i
    for each net i in B
        for each net j in A where signature(j)=signature(i) using hash table
            construct a SAT miter between i and j
            if (unsat) record functional equivalence (i==j) on i
}
```

Figure 2: Functional Equivalence

The second phase uses Boolean matching starting from corresponding primary outputs and proceeds recursively backwards as it matches small chunks of logic, thus forming the output-side boundary of the logic change. One problem with this approach is that it returns large sets of changes to implement ECOs when the changes are close to the primary inputs or drive a large fanout. This is due to limitations of subcircuit enumeration and Boolean matching method.

4. LOGIC ECO SYNTHESIS

We observe that designers are often able to perform ECO synthesis quickly because they come up with intelligent guesses about correspondences between the VHDL and the original implementation and then verify them. In this section, we formalize the notion of a *user hint* in the framework of functional correspondence, which can be utilized in logic eco synthesis. Then, we discuss how to automatically derive intelligent functional correspondences. We begin with some notations and problem formulation:

1. The original functional specification is denoted *original VHDL*.
2. The modified functional specification containing the ECO is denoted *ECO VHDL.*
3. The original implemented netlist is denoted *original netlist.*
4. A preliminary synthesized gate-level version of the ECO VHDL is known as the *ECO netlist*.

Problem statement: *The problem of logic ECO synthesis is to determine a minimal logic delta, i.e., a list of gates and connections to be inserted into existing logic, such that the ECO and original netlists are rendered equivalent.*

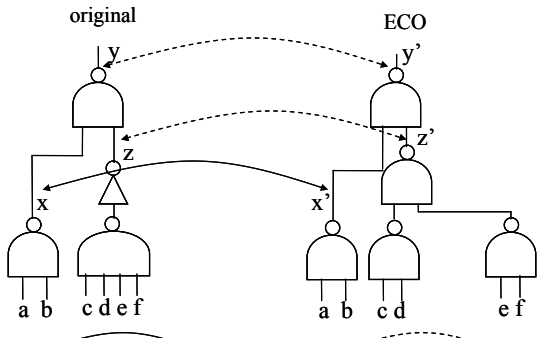

Figure 3: Functional Equivalence and Correspondence

Minimizing the *delta* also implies maximizing reuse of existing logic in the original netlist. As in [1], we also begin by marking and eliminating functionally equivalent signals from the *delta*. However, functional equivalence can only find signals upstream of logic changes. For example, in Figure 3, functional equivalence can eliminate the signal *x* and its fanin-cone from the logic changes. It cannot detect that the gate driven by *z* can also be re-used in synthesizing the ECO. This requires the recognition of a different concept which we define and explore in this section.

4.1 Functional Correspondence

Definition A **functional correspondence** between two logic circuits, *ckt_orig and ckt_eco*, is a 2-tuple *(S, S'),* where *S* and *S'* are sets of signals $S=\{s_1, s_2,..., s_n\}$, $S' = \{s_1', s_2', ..., s_n'\}$, from *ckt_orig* and *ckt_eco* respectively, such that substituting signals in *S* from *ckt_orig* with those of *S'* (including the gates that drive signals in S') results in *ckt_orig* becoming functionally equivalent to *ckt_eco*.

The individual signal pairs $(s_1, s_1'), (s_2, s_2') \ldots (s_n, s_n')$ are said to be **correspondence pairs**. For example, in Figure 3, the set of signals $S=\{z\}$ and $S'=\{z'\}$ form a functional correspondence, so do $S=\{y\}$ and $S'=\{y'\}$. Figure 4 gives the algorithm for verifying a functional correspondence based on the definition.

```
verify_functional_correspondence ( S in ckt_orig, S' in ckt_eco)
{
    connect primary inputs of ckt_orig and ckt_eco together
    for each correspondence pair (si, si')
        connect sinks of si in S to si' in S'
    check equivalence beween ckt_orig and ckt_eco
    if (equivalent) return true
    else return false
}
```

Figure 4: Verifying a Functional Correspondence

Definition A **well-formed functional correspondence** *(S,S')* has the property that it is no longer a functional correspondence if any correspondence pair *(s,s')* is dropped from *(S,S')*.

For example, in Figure 3, $S=\{y,z\}$, $S'=\{y',z'\}$ is not a well-formed functional correspondence because after removing *(y,y')*, the result, $S=\{z\}$, $S'=\{z'\}$, is still a functional correspondence. Generally, this means that in traversing the circuit along any path from a primary input to a primary output, only one corresponence pair is reached. From here on, when we refer to functional correspondences, we mean well-formed correspondences.

Functional correspondences can form the output boundary of the logic ECO changes, with the substitutable logic being directly apparent in the ECO netlist. The number of gates to be inserted, i.e. the *delta* size, is the number of gates in the fanin cones of functional correspondences *S'* that are not marked by functional equivalence step.

The trivial functional correspondence for any ECO is formed by $S=\{o_1,o_2,...,o_n\}$ and $S'=\{o_1',o_2',...,o_n'\}$, where *o* and *o'* are primary outputs not matched after functional equivalence. The resulting *delta* is usually not very good unless the changes are close to primary outputs. The problem of deriving a minimal set of changes can be formulated as the problem of finding the best functional correspondences. A functional correspondence *(S, S')* is *better* than another correspondence *(F, F')* if it results in a smaller set of logic changes. In the remainder of this section, we derive efficient methods by which functional correspondence can be determined and verified.

4.2 Compute Functional Correspondence

In this section, starting from *any* list of correspondence pairs, we show how to generate a functional correspondence. We emphasize that this can include guessed and potentially invalid pairs, i.e., pairs that do not belong to any functional correspondence.

In order to solve this problem optimally, we can use a branch-and-bound approach, which branches on the number of intermediate equivalences created upon immediate replacement of the correspondence pair in question. For instance, in Figure 3, if *(x, y')* were entered as a potential correspondence pair, it would not create any additional equivalences between the designs. However, if *(x, x')* were used then it would create an additional equivalence at *(y, y')*. Equivalences do not have to be proven and simulation can generate new potential equivalences for the purposes of branch and bound.

In practice, if the correspondence pairs we generate are primarily valid pairs (or if invalid pairs are topologically ahead of valid pairs), then we find that a greedy approach gives good results. The greedy algorithm is illustrated in Figure 5. Here, we traverse the set of corresponding pairs, N, in topological order. For each corresponding pair *(n,n')*, we first evaluate whether *n* is equivalent to *n'*. If not, we add *n* into *S* and *n'* into *S'* and remove any corresponding pairs that are in the *maximum fanout free cone* (MFFC) of *n* and *n'* from *S* and *S'* respectively. At each step, we also speculatively connect the sinks of *n* to *n'*. This essentially allows *(n,n')* to be part of a speculative functional correspondence. When a node is removed from *(S,S')*, these connections are restored. Eventually, it will reach the correspondence pairs on the primary outputs, if the existing pairs are not equivalent, they will be inserted into *(S,S')*. Therefore *(S,S')* will always produce a valid functional correspondence.

```
functional_correspondence_greedy ( correspondence pairs
N={ (n1,n1'), (n2, n2'), … (nk, nk')})
{
    Insert default correspondence pairs on primary outputs
    sort N topologically
    assign S and S' as empty set
    foreach (n,n') in N
        if ( n and n' are equivalent) ) then next
        (C,C') = {(m,m') : s∈MFFC (n)  && s'∈MFFC (n')
                          && {m,m'}∈(S,S') }
        for each pair {m,m'} in (C,C')
            restore sinks of m in S from m' in S'
        S  =  (S – C) ∪ {n}
        S' =  (S' –C') ∪  {n'}
        connect sinks of n in S to n' in S'
    return (S,S')
}
```

Figure 5: Compute Functional Correspondence

For example, in Figure 3, *N = {(z,z'), (y,y')}*. The algorithm will traverse *(z,z')* first since it is topologically before *(y,y')*. Since *z* is different to *z'*, it will add *(z,z')* to *(S,S')* and connect the sinks of *z* to *z'*. Then, it will process *(y,y')* and *y* and *y'* are actually equivalent due to the reconnection on *(z,z')*. Thus, the algorithm will not add *(y,y')* into *(S,S')*. In the end, it will return functional correspondence *(S={z},S'={z'})*. If the gate driving *y'* is a NOR gate instead, *y* and *y'* will not be equivalent even after we reconnected *z* and *z'*. Then pair *(y,y')* will be added to *(S,S')* when processing *(y,y')*; and *(z,z')* will be removed and sinks restored since it is in the MFFC of *(y,y')*.

To complete our method, we feed the smaller design consisting of the gates between the derived functional equivalences and derived functional correspondences to the algorithm from [1] which can further reduce the *delta* by Boolean matching. The problem size given to the Boolean matching phase is drastically reduced after applying functional correspondence.

5. GENERATING CORRESPONDENCES

In the previous sections, we described the main ideas of how functional correspondences can reduce the *delta* size and how to compute the best functional correspondence based on a given set of correspondence pairs. In this section, we describe how to generate likely correspondence pairs.

In practice, most of the VHDL signal names exist in both the original and the ECO VHDL. Naturally, these names are the candidates for partial correspondence. For example, compare following two lines of VHDL determined by a program such as xdiff:

Original: *rst <= ls_need AND ex_hit;*
ECO: *rst <= core AND ls_need AND ex_hit;*

If we were able to find the *rst* signal in the ECO and original netlists, then it would constitute a functional correspondence that can be used to reduce the ECO. The difficulty lies in the fact that the signal *rst* itself is not always visible in the gate-level netlists due to synthesis transformations merging several signals.

To address this issue, we propose to find equivalences between signals in the RTL and the gate-level netlists, to find functional correspondences upstream of primary outputs by taking advantage of the speed of modern logic synthesis tools. First, the ECO and original VHDLs are mapped into gate-level implementations such that signal names are preserved in the mapped netlist. We use the term "map" to mean that it is not necessary to perform tedious redundancy removal or other complex steps to generate these versions [24]. These mapped versions are called ECO_B and ORIG_B netlists. Then, we identify names that are common to both ECO_B and ORIG_B signals. Next, we detect and prove further equivalences between the pairs of netlists (ECO_B, ECO), and (ORIG_B, original) through the algorithm in Figure 2. Then, we can find functional correspondence between ECO and original by matching the functional equivalent signals based on the name-matched signals in ECO_B and ORIG_B. The detailed algorithm is given in Figure 6.

For example, Figure 7 shows four netlists: the original, the ECO, the mapped ORIG_B and ECO_B sharing the same PI: *a*, *b*, *c*, *d*, *e*, and *f*. We can see that there are no internal functional equivalences between the original netlist and the ECO netlist. The names of the internal signals are also quite different. But we can locate the functional equivalence (*x,o1*), (*z,o2*) and (*y,o3*) between ORIG_B and original; also (*e0,x*), (*e4,z*) and (*e2,y*) between ECO and ECO_B. Since *x*, *y* and *z* are signal names preserved in both ORIG_B and ECO_B, we can use the algorithm described in Figure 6 to derive correspondence pairs (*o1,e0*), (*o2,e4*) and (*o3,e2*) between original and ECO netlists. Note, that this method requires no alteration to the original synthesis method. It does require simple synthesis runs, which are very fast compared to physical design runs, and result in physical design efforts saved by the production of a smaller logic *delta*.

```
find_named_functional_correspondences(all VHDLs, netlists)
{
    ORIG_B = map(original VHDL)
    ECO_B = map(eco VHDL)
    find_functional_equivalences(original netlist, ORIG_B)
    find_functional_equivalence(ECO netlist, ECO_B)
    locate_name_based_correspondences(ECO_B, ORIG_B)
    for each name_based_correspondence (s, s')
      if (exists_equivalent_signal(s, original netlist))
        n=equivalent_signal(s, original netlist)
        if (exists_equivalent_signal(s', ECO netlist))
          n'=equivalent_signal(s', ECO netlist)
          add (n, n') to list of correspondences pairs
}
```

Figure 6: Name-based Functional Correspondences

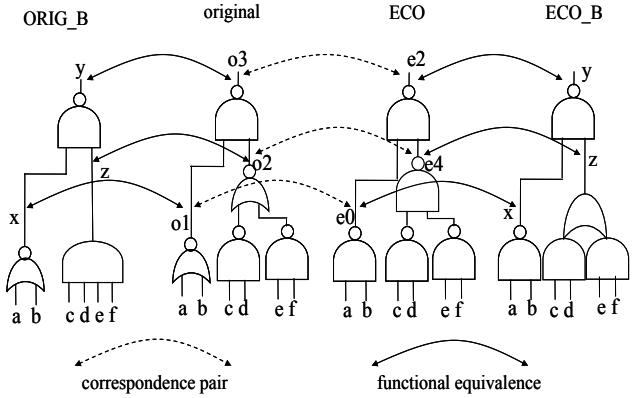

Figure 7: Name-based Functional Correspondence Example

Correspondence pairs can also be generated by designers who can intuitively determine signals，which correspond in any part of the logic. These *user hints* do not have to form a complete functional correspondence, as any correspondence pair can be utilized in the algorithm of Figure 5. Thus, our method is able to directly and systematically utilize any user intuition about ECO changes.

6. AN INDUSTRIAL ECO FLOW

Logic ECO process described in previous sections produces functionally correct circuits. It is up to ECO physical synthesis tools to place and optimize those circuits to satisfy timing constraints. The physical synthesis process must disturb a minimal amount of logic beyond the logic delta provided by the logic ECO process. Our ECO physical synthesis flow is illustrated in Figure 8. The first step is to place the ECO gates (the *Delta* netlist) produced the logic ECO flow. To maintain design stability, all the existing gates in the original netlist, excluding those deleted during ECO, will be kept fixed in their location. Existing gates are considered fixed blockages by the placer and other algorithms. First, ECO gates will be placed in the remaining empty space. Then, the placer will optimize total wirelength among the ECO gates and their connected original gates. After ECO gates are placed, the second step is to size and buffer these gates to fix electrical violations [4] such as max slew and max driving capacitance violations. These violations are fixed by gate sizing and buffering. During this step,

only the ECO gates are allowed to change under an in-place-optimization (IPO) framework that it minimizes the disturbance to the existing placement.

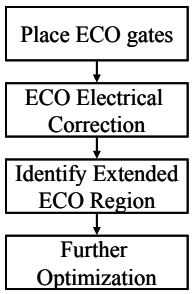

Figure 8: ECO Physical Synthesis

For high performance designs, the ECO circuit often has to be extensively optimized to meet tight timing constraints. Introducing ECO gates might affect the timing of other paths. A larger region than the ECO gates will need to be optimized if necessary. The algorithm to identify extended ECO region is given in Figure 9. Once the extended ECO region is identified, many IPO transforms [5] will be applied to circuits in the region to further optimize the design.

```
identify_extended_ECO_region (ECO gates E, K)
{
  for each gates in E
      forward traverse up to K levels and mark gates
      backward traverse up to K levels and mark gates
  for each marked gates
      forward traverse up to K levels and mark gates
  unmark gates with positive slacks
  assign marked gates as extended ECO region
}
```

Figure 9: Identify ECO Region

When the FEOL masks are produced, ECO needs to be implemented with spare cells. Spare cells in our design methodology are gate array cells that are distributed in the space not occupied by standard cells. These cells can be rewired to implement a few basic logic functions during ECO. We are required to provide spare cell support in our ECO physical synthesis flow. During placement step, it is the spare cells that are placed. The spare cells are also placed at certain placement intervals. During optimization, we can not change the size or the location of any existing gate, instead, we can only create spare cell clones for these gates if they are underpowered, or buffer the existing nets with spare gates.

Next, we show the empirical results our ECO logic and physical synthesis flows real ECOs on a set of IBM microprocessor macros. We will compare the results of the new logic ECO flow to the algorithm from [1]. For simplicity we will refer the new logic ECO flow as INTUIT in the rest of this section. Our code is written in C++/TCL and run on IBM P570 servers (L1 i64K d64K, L2 4M).

6.1 ECO Logic Synthesis Results

We are given the original netlist, original VHDL and ECO VHDL for these macros. The total number of VHDL line changes is listed in the second column (V#) of Table 1. Although there is often only one line of VHDL that is changed, it can affect many gates in the circuit as shown in "Cone Gates" column. We run both DeltaSyn and INTUIT on these macros and report the *delta* size in the "delta" column. In M1-M5, where the design changes are close to primary outputs, DeltaSyn is capable in producing small *deltas*. It is clear, however, that the *delta* sizes produced by INTUIT are significantly smaller in M6-M12, where the design changes are not close to primary outputs. Overall, the *delta* size produced by INTUIT is only 6% as large as those from DeltaSyn: a 94% reduction! The microprocessor design team has verified that the sizes of these *deltas* are similar to what is produced manually [28].

Table1: Logic ECO Statistics on IBM Benchmark

macro	V#	Total Gates	Cone Gates	DeltaSyn		INTUIT	
				delta	CPU	delta	CPU
M1	1	7145	37	17	113	16	113
M2	6	3427	169	34	64	30	64
M3	1	8351	8	1	94	1	93
M4	4	14115	1867	44	139	11	138
M5	1	7929	317	4	101	4	101
M6	3	4025	963	265	72	12	72
M7	3	4632	1482	456	75	8	79
M8	4	19263	4180	861	226	10	245
M9	6	17135	1603	102	204	14	200
M10	3	10729	2777	605	108	27	111
M11	2	105620	2680	271	1332	58	1509
M12	1	25458	7384	1086	682	32	703
SUM				**3746**		**223**	

The runtimes (in seconds) of both DeltaSyn and INTUIT are reported on "CPU" columns. The runtime includes both the front end synthesis [24] and the ECO algorithm. We observed that more than 90% of runtime is synthesis runtime. Since synthesis runtime dominates the entire flow, we can see the INTUIT runtime is about the same as much as the DeltaSyn runtime.

6.2 ECO Physical Synthesis Results

We run ECO physical synthesis flow on both the DeltaSyn generated *delta* and the INTUIT-generated *delta*. During physical synthesis we set K in Figure 9 to 2, which extends the ECO region by 2 levels. The results are shown in Table 2.

To evaluate quality of result (QoR) impact, we report the worst negative slack (WNS) and total negative slack (TNS) before and after ECO physical synthesis. The sums of WNS and TNS on all macros are reported in the last row. Note that on M7 and M10 post-ECO timing with DeltaSyn degraded significantly due to large *delta* sizes produced by DeltaSyn. In comparison, post-ECO timing with INTUIT is better than pre-ECO on the majority of the macros. Overall WNS is 14% better and TNS is 46% better.

In Table 2 we also report the design perturbation introduced by ECO physical synthesis. A# column reports the number of gates added during ECO physical synthesis, D# reports the number of gates deleted and C# reports the number of gates changed, which includes resized or moved. Again, comparing the results of INTUIT and DeltaSyn, we can see that ECO physical synthesis introduced much less design changes with INTUIT *deltas*. For example, INTUIT resulted in an 84% reduction in total added gates, as compared to DeltaSyn.

K in Figure 9 can be adjusted to balance between design perturbation and QoR improvement. A larger K opens up an extended region for optimization. Therefore, better QoR is possible. However, an extended physical ECO region also allows more gates changed, which means bigger perturbation. A larger perturbation often results in longer runtime and worse QoR for backend tools such as ECO routing. The tradeoff between QoR improvement and perturbation is shown in Figure 10. For example, using K=0 as the baseline, extra 24% of WNS and 31% of TNS improvement is achievable when K=5 at the cost of 70% more gates added and 246% more gates changed.

Table 2: ECO Physical Synthesis Statistics on IBM Benchmark

macro	WNS (ps)	TNS (ps)	ECO physical synthesis from DeltaSyn						ECO physical synthesis from INTUIT					
			WNS	TNS	A#	D#	C#	CPU	WNS	TNS	A#	D#	C#	CPU
M1	-32.6	-2073.7	-22.1	-1023.2	155	5	28	201	-22.2	-1059.9	163	4	26	186
M2	-2.6	-2.6	-2.6	-2.6	40	27	32	165	-2.7	-2.7	52	38	24	150
M3	-1.3	-1.3	-1.3	-1.3	10	0	25	191	-1.4	-1.4	10	0	25	183
M4	-18.8	-87.9	-18.8	-71.2	104	291	225	278	-18.8	-85.9	19	19	231	259
M5	-4.3	-4.31	-9.7	-14.0	4	48	30	198	-4.3	-4.3	3	35	30	190
M6	-35.4	-1502.6	-17.9	-451.2	552	209	154	219	-17.9	-673.5	118	7	24	177
M7	1.9	0	-33.7	-871.3	1184	361	1103	999	8.7	0	58	17	115	488
M8	-46.3	-116.0	-46.3	-126.0	1035	763	238	500	-46.3	-116.0	14	8	29	271
M9	-15.7	-64.3	-20.6	-69.2	147	106	39	291	-15.7	-64.	35	12	39	287
M10	2.6	0	-129.1	-4069.6	1019	665	21	889	-12.8	-30.1	36	20	21	245
M11	-45.5	-121.2	-45.5	-124.7	716	423	132	1224	-45.5	-121.2	212	47	32	1153
M12	-6.1	-54.6	3.3	0	2920	1593	638	1457	3.3	0	536	349	179	767
SUM	**-204.1**	**-4028.51**	**-344.3**	**-6824.3**	**7886**	**4491**	**2665**	**6612**	**-175.6**	**-2159**	**1256**	**556**	**775**	**4356**

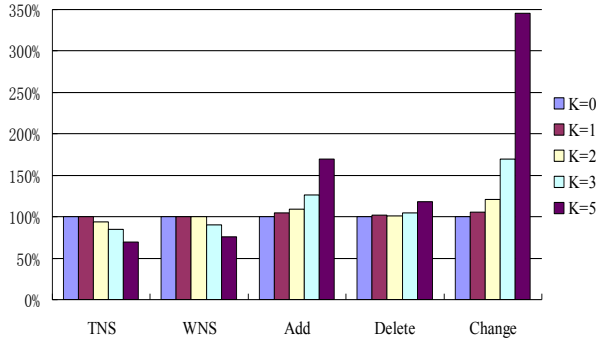

Figure 10: QoR and Perturbation Tradeoff on K

7. Conclusion

We presented a novel logic ECO approach that uses synthesized versions of the ECO and original netlists to find potentially substitutable pairs of points in logic known as correspondence pairs. We presented methods by which good functional correspondences can be derived using these or other guessed correspondence pairs to drastically reduce the logic changes needed to synthesize an ECO. We then incorporated our method into an industrial ECO physical synthesis flow. Results show that our method produces improved timing and significantly smaller design perturbation than previous methods.

8. REFERENCES

[1] S. Krishnaswamy, H. Ren, S. Modi, and R. Puri, "DeltaSyn: An efficient logic difference optimizer for ECO synthesis," *ICCAD* 2009, pp.789-P796.

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

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

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

[5] C. J. Alpert, S. Karandikar, Z. Li, G. J. Nam, S. Quay, H. Ren, C. Sze, P. Villarrubia, and M. Yildiz, "The Nuts and Bolts of Physical Synthesis," *International Workshop on System Level Interconnect Prediction* 2007, pp. 89-94

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

[7] H. Ren, D. Z. Pan, C. J. Alpert and P.Villarrubia, "Diffusion-based Placement Migration," *DAC 2005*, pp. 515-520.

[8] J. Cong and M. Sarrafzadeh, "Incremental Physical Design", *ISPD 2000*, pp. 84-92.

[9] M. Cho, H. Ren, H. Xiang, R. Puri, "History-based VLSI Legalization Using Network Flow," *DAC* 2010, pp.286-291

[10] A. C. Ling, S. D. Brown, J. Zhu, S. Safarpour, "Towards Automated ECOs in FPGAs," *FPGA* 2009, pp. 3-12.

[11] B.-H. Wu, C.-J. Yang, C.-Y. Huang, J.-H. R. Jiang, "A Robust Functional ECO Engine by SAT Proof Minimization and Interpolation Techniques," *ICCAD* 2010

[12] K.-H. Chang, I.L. Markov and V. Bertacco, "Fixing design errors with counterexamples and resynthesis," *ASPDAC* 2007, pp. 944-949.

[13] S.-Y. Huang, K.-C. Chen and K.-T. Cheng, "Error correction based on verification techniques," *DAC* 1996, pp. 258-261

[14] S. Safarpour, H. Mangassarian, et al., "Improved design debugging using maximum satisfiability," *Formal Methods in Computer-Aided Design* 2007, pp. 13–19.

[15] Y. Chen, J. Fang and Y. Chang, "ECO timing optimization using spare cells," *ICCAD* 2007, pp. 530-535.

[16] Y. Li, J. Li and W. Chen, "An efficient tile-based ECO router using routing graph reduction and enhanced global routing flow," *TCAD*, vol.26, no.2, pp. 345-357, Feb. 2007.

[17] O. Coudert, J. Cong, S. Malik, and M. Sarrafzadeh, "Incremental CAD," *ICCAD* 2000, pp. 236-243.

[18] S.-Y. Huang, K.-C. Chen and K.-T. Cheng, "AutoFix: A Hybrid Tool for Automatic Logic Rectification", *TCAD*, pp. 1376-1384, Sep. 1999.

[19] Y. Watanabe and R. K. Brayton, "Incremental synthesis for engineering change," *ICCAD* 1991, pp. 40–43.

[20] A. Smith, A. Veneris and A. Viglas, "Design Diagnosis Using Boolean Satisfiability", *ASPDAC* 2004, pp. 218-223.

[21] H.-T. Liaw, J.-H. Tsaih, and C.-S. Lin, "Efficient automatic diagnosis of digital circuits," *ICCAD* 1990, pp. 464–467.

[22] C.-C. Lee, J.-H. R. Jiang, C.-Y. R. Huang, and A. Mishchenko, "Scalable exploration of functional dependency by interpolation and incremental SAT solving," ICCAD 2007, pp. 227–233.

[23] V. Manohararajah, D. P. Singh, and S. D. Brown, "Post-placement BDD-based decomposition for FPGAs," *International Conference on Field-Programmable Logic and Application*, Aug. 2005, pp. 31–38.

[24] L. Stok, D.S. Kung, D. Brand, et al., "BooleDozer: Logic Synthesis for ASICs," *IBM Journ. of R&D*, vol 40, no. 4, July 1996.

[25] N. Sorensson and N. Een, "Minisat v1. 13-a sat solver with conflict-clause minimization," in *SAT Competition*, 2005.

[26] K.-H. Ho, J.-H. R. Jiang, Y.-W. Chang, "TRECO: Dynamic Technology Remapping for Timing Engineering Change Orders", *ASPDAC* 2010, pp. 331-336.

[27] A. Kuehlmann, "Dynamic transition relation simplification for bounded property checking," *ICCAD* 2004, pp. 50-57

[28] D. Rude, *Personal Communication*, Design Engineer, IBM Systems & Technology Group, August, 2010

[29] K-F. Tang, C-A. Wu, P-K. Huang, C-Y. Huang, "Interpolation-based incremental ECO synthesis for multi-error logic rectification", DAC 2011, pp.146-151