Recovering Hierarchical Boundaries in a Flat Netlist

Kuo-Wei Ho[1], Yu-Wei Fan[1], Jie-Hong R. Jiang[1,2], Alan Mishchenko[3], Robert Brayton[3], Sean A. Weaver[4]
[1]*Graduate Institute of Electronics Engineering, National Taiwan University*
[2]*Department of Electrical Engineering, National Taiwan University*
[3]*Department of Electrical Engineering and Computer Sciences, University of California, Berkeley*
[4]*Laboratory for Advanced Cybersecurity Research, U.S. National Security Agency*

***Abstract*—In a typical integrated circuit design flow, a hierarchical design specification is optimized and mapped into a library by a synthesis tool, resulting in a flat netlist without hierarchical instances. However, in some practical scenarios, it may be important to reintroduce into the mapped netlist the boundary of some instances defined in the specification. This paper presents an automated recovery method that works even in those cases when synthesis completely removes the original nodes on the boundary. The paper also discusses several use cases of the recovered boundary in synthesis, verification, and engineering change orders.**

I. Introduction

The hardware design flow is a sequence of transformations automatically applied by a computer-aided design (CAD) toolchain to a hardware design description. The design is typically given as a register-transfer-level (RTL) specification, composed of modules combined together to implement specific functions in the given design. The toolchain working on the RTL specification applies transformation one by one, resulting in a sequence of intermediate representations. For example, the RTL specification may be internally flattened leading to the loss of the hierarchical information. Next, the word-level operators introduced after flattening are elaborated and replaced by functionally-equivalent bit-level circuits. These circuits are synthesized and mapped by other parts of the toolchain, resulting in a flat netlist, which may be written out into a file before applying physical design tools.

Although in a typical design flow described above, the toolchain produces a variety of intermediate representations, they are typically not visible to the user, who sees only the RTL specification at the beginning and the mapped netlist at the end of synthesis and mapping. The resulting netlist is "flat" in the sense that it does not contain hierarchical information present in the original specification. However, this hierarchical information may be important in a number of practical applications described below.

It may be observed that the boundaries between the instances can be maintained by customizing the synthesis script. However, in many cases, doing so compromises the quality of synthesis, since the logic network is not optimized across the fixed instance boundaries. The designer may also be working on previously synthesized designs, for which rerunning synthesis may be impractical or impossible, and yet recovering the boundaries may be desirable.

Before we further motivate the need for recovering hierarchical boundaries, it should be noted that this task is greatly simplified by assuming that the flat netlist contains nodes that are functionally equivalent to the nodes found on the boundary in the hierarchical specification. This assumption rarely holds in practice because logic synthesis often restructures the netlist in such a way that the intermediate nodes are removed while the functions of the primary outputs are preserved. Additionally, some instances may have outputs that do not have external drivers. After elaboration, such outputs without fanout are typically removed by the data-structure cleanup before logic synthesis begins.

The goal of this paper is to develop an automated method to recover hierarchical boundaries in the flat netlist after synthesis and mapping, without assuming that the mapped netlist has nodes equivalent to the boundary nodes in the specification.

The problem is solved in several steps. First, the hierarchical specification and the flat implementation are both transformed into an equivalent and-inverter-graph (AIG) to enable efficient simulation and SAT solving. Next, SAT sweeping of the AIG is used to identify equivalent node pairs across the implementation and the specification. Next, the instance in the implementation that corresponds to the given instance in the specification is localized by analyzing the available node equivalences, resulting in the "extended instance". The extended instance is a superset of the logic nodes found in the target instance, including logic nodes found between the target instance and the closest pairs of equivalent nodes. Finally, by replacing the extended instance in the implementation with the copy of the extended instance taken from the specification, the boundary of the original instance can be successfully recovered. The details of this procedure are presented in Sections IV and V.

Potential applications of the proposed method are as follows:

1) **Synthesis**: In some cases, the designer who runs industrial tools on the hierarchical design may be unsatisfied with the quality of the resulting synthesized/mapped netlist. It may be, for example, due the fact that the design has some novel features, which are not widely supported by the tools, or just an accidental quality degradation. At the same time, it may be obvious for the designer that some parts of the design should be synthesized differently, in particular, by using manual transformations or applying another tool on a part of the target design. However, typical industrial tools do not maintain hierarchical instances, and even if they do, they typically maintain only some but not all boundaries; otherwise, the quality of synthesis can be substantially reduced. In such cases, if the designer has a hierarchical RTL design and the flat netlist produced by the tool, we may recover the boundary of a module using the proposed method and insert the result of

manual synthesis. This may improve the quality of the design (delay, area, power, etc).

2) **Model checking**: Consider a flat synthesized/mapped design that needs to be verified and a verification tool that uses the abstraction-refinement methodology. It may be helpful to abstract specific functionalities that are present in the RTL but are not clearly visible in the flat netlist. If we can transfer the boundary to the flat netlist, abstraction-refinement can proceed with better guidance.

3) **Equivalence checking**: For example, when an instance is a large multiplier, SAT sweeping may time out in the transitive fanout of the multiplier, and as a result, the proposed boundary recovery method may not be applicable. In such cases, we may be able to use simulation to find equivalent node candidates, as opposed to exact equivalences found by SAT sweeping. If we succeed in this case in reconstructing the boundary based on candidates rather than exact equivalences, we can divide the equivalence-checking proof into two parts: (a) prove the multiplier to be equivalent to its specification using, for example, algebraic methods. (b) prove the netlist after logic grafting to be equivalent to its specification.

4) **Engineering change order (ECO)**: In a typical ECO scenario, the hierarchical specification changes due to a bug fix or a new feature addition. If this happens at the end of the design cycle, when the synthesized and mapped netlist is already generated, possibly after weeks of painstaking effort by the design team, it may be costly or impossible to rerun synthesis from scratch on the updated specification. However, if the change of the specification can be localized, we can define a hierarchical instance containing the change in the RTL. The boundary of this instance indicates the boundary of change due to the last-minute bug fix or new feature. In this context, the proposed method provides an efficient and simple solution. We first apply the proposed method to recover this instance in the implementation, by comparing it against the original specification before it is changed by ECO. If the boundary recovery is successful, we can transfer the changes from the modified specification directly, by replacing the recovered instance. This way we can update the implementation without the need to resynthesize it from scratch. Furthermore, defining the recovered instance to only contain the necessary changes also minimizes the patch.

5) **Logic block annotation**: It may be helpful to associate signals in the netlist after physical design with those in the original RTL. This association allows for annotating parts of RTL with the design information, such as timing, area, or power, derived by physical design. The RTL parts with labeled physical information can serve as the training data for machine learning. To perform this annotation, we consider the relevant region in the physical netlist as a target instance in the specification and treat the RTL netlist as the implementation. We can use the proposed boundary recovery method to identify an extended instance in the RTL netlist containing the region of the physical netlist, thus enabling the transfer of the physical design information to the RTL netlist.

The rest of the paper is organized as follows. Section II describes the preliminaries. Section III outlines the problem formulation. Section IV presents the method used to solve the problem. Section V presents an enhancement to our method with speculative reduction, followed by the experimental results in Section VI, a discussion in Section VII, and conclusions in Section VIII.

II. Preliminaries

A Boolean netlist is a directed-acyclic graph (DAG) with nodes corresponding to logic gates and edges corresponding to the interconnections between these gates. We use netlist and circuit interchangeably to refer to a Boolean netlist. Given a graph G, we use the notation $\textsc{Nodes}(G)$ to denote the set of nodes of G. The source nodes and the sink nodes of a netlist are called the *primary inputs* (PIs) and the *primary outputs* (POs), respectively. The *internal nodes* are all nodes excluding PIs and POs. And-Inverter Graph (AIG) represents a Boolean netlist consisting of only two-input AND gates and inverters, where the inverters are annotated on the edges. In this work, we consider a netlist as an AIG. We say n is a *fanin* of n' and n' is a *fanout* of n if (n, n') is an edge in the Boolean netlist. Also, (n, n') is a *fanout edge* of n and a *fanin edge* of n'. For a node n, we use the notations $\text{FO}(n)$ and $\text{FI}(n)$ to denote the set of fanouts and the set of fanins of n, respectively.

If there is a path from node n to node n', then n called the *transitive fanin* of n' and n' is called the *transitive fanout* of n. Given a node n, the *transitive fanin (fanout) cone*, denoted as $\text{TFI}(n)$ ($\text{TFO}(n)$), is a subgraph with the set of nodes being

$$N = \{n' \mid n' \text{ is a transitive fanins (fanouts) of } n\} \cup \{n\} \quad (1)$$

and the set of edges being

$$E = \{(n', n'') \mid n', n'' \in N\}. \quad (2)$$

A cut C for $(n, \text{TFI}(n))$ is a set of nodes, called *leaf nodes* or *leaves*, where every path from PIs to n passes through exactly one node in C and n is called *root node* of C. Symmetrically, C is a cut for $(n, \text{TFO}(n))$ if every path from n to POs passes through exactly one node in C. Given a set of nodes N, we say N is *sufficient* to form a cut for n and $\text{TFO}(n)$ ($\text{TFI}(n)$), if $N \cap N'$ forms a cut with N' being $\textsc{Nodes}(\text{TFO}(n))$ ($\textsc{Nodes}(\text{TFI}(n))$). In a netlist G, a *fanout-free netlist* G' is a subgraph where $\text{FO}(n) \subseteq G'$ for every node $n \in G'$.

For each node n, we associate it with a Boolean variable x_n to represent the logic value of the gate output. An *input pattern* is an assignment that maps the variables of all PIs to Boolean values. Given an input pattern, we can simulate the AIG by applying the following simulation rules to each and-node n in a topological order.

$$x_n = \begin{cases} \neg x_{n_1} \wedge \neg x_{n_2}, & \text{if both fanin edges have inverters,} \\ \neg x_{n_1} \wedge x_{n_2}, & \text{if only } (n_1, n) \text{ has an inverter,} \\ x_{n_1} \wedge \neg x_{n_2}, & \text{if only } (n_2, n) \text{ has an inverter,} \\ x_{n_1} \wedge x_{n_2}, & \text{otherwise,} \end{cases} \quad (3)$$

where n_1 and n_2 are the two fanins of n. This allows us to uniquely determine the values of all nodes under an input pattern. A node n is *equivalent* to another node n', and vice versa, if the value of x_n equals that of $x_{n'}$ under all possible input patterns.

A hierarchical netlist G is a netlist containing an *instance*. An instance is a fanout-free netlist $\text{TFO}(N_{\text{BI}}) \cap \text{TFI}(N_{\text{BO}}) \in G$ with two sets of nodes N_{BI} and N_{BO} being its *boundary inputs* and *boundary outputs*, respectively.

SAT sweeping [1] is a commonly used approach to perform netlist optimization and verification represented as AIGs through efficient detection of equivalent nodes. In this work, we use SAT sweeping to compute the set of equivalent node pairs, which are used by the proposed boundary recovery method.

III. Problem Formulation

The problem can be stated formally as follows: Consider two netlists, the hierarchical specification S and its flattened implementation I, and the boundary B_{S} in S composed of the input and output nodes of a target instance. The solution to the boundary recovery problem is the boundary $B_{\text{I}'}$ in I' composed of nodes that may not have existed in I, such that the Boolean functions of B_{S} and $B_{\text{I}'}$ are equivalent and the modified I' is equivalent to the original S. Also, we want to minimize the changes to the logic of I' needed to recover the boundary of the target instance.

IV. Methods

A. Identifying Extended Boundary

Given an implementation netlist I and its specification S with a hierarchical boundary, the boundary recovery is divided into two parts: The first part identifies the equivalent pairs of nodes belonging to I and S using SAT sweeping. The second part uses structural analysis on I and S using the node pair equivalence information as the guidance. This would include computing a set of nodes in I, where their corresponding equivalent nodes in S form a cut in the transitive fanin/fanout cone composed of boundary inputs/outputs.

The main procedure IDENTIFY-EXTENDED-BOUNDARY(I, S) is shown in Algorithm 1. Let N_{BI} and N_{BO} be the set of nodes belonging to the boundary inputs and boundary outputs in S, respectively. The sub-procedure IDENTIFY-EQ-NODES(I, S) then identifies the equivalent pairs of nodes in I and S using SAT sweeping. Let EQ-NODE(n) be the equivalent node of n in another netlist, and EQ-NODE(n) $= \emptyset$ if there are no equivalent nodes of n in another netlist. Note that a node can generally be equivalent to more than one node in another netlist. In this case, let EQ-NODE(n) be the first node detected as equivalent to n by SAT sweeping.

Once the equivalence information is computed, structural analysis is performed as follows. For each boundary output n, procedure FIRST-EQ-CUT-TFO(n) is used to find a set of nodes in S that forms a cut in TFO(n). These nodes become the *extended boundary outputs* (EBOs) N_{EBO}. Similarly, we can find a set of *extended boundary inputs* (EBIs) N_{EBI} in S using the procedure FIRST-EQ-CUT-TFI(n) for a boundary input n. The procedure FIRST-EQ-CUT-TFO(n) is shown in Algorithm 2. Given a root node n in S, the procedure performs DFS traversal from n toward the fanouts until a valid cut with all the nodes having equivalent nodes in I is found. We can always find such a cut since the POs form a trivial cut.

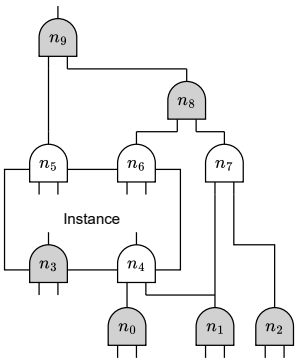


Fig. 1: A part of the specification netlist near the boundary.

Procedure FIRST-EQ-CUT-TFI(n) is the same as Algorithm 2, except that FO(n_{cur}) in Line 10 is replaced by FI(n_{cur}).

Recall that we found the nodes of N_{EBO} in $\text{TFO}(N_{\text{BO}})$ and the nodes of N_{EBI} in $\text{TFI}(N_{\text{BI}})$, respectively. Such a boundary may not be valid since the EBIs may not be sufficient to form a cut for $(n, \text{TFI}(n)$ with an EBO n. Fig. 1 shows part of the netlist S. $\{n_5, n_6\}$ and $\{n_3, n_4\}$ are the boundary outputs and inputs of the instance, respectively. The gray nodes are those that have an equivalent node in I. In this example, the current N_{EBO} is $\{n_8, n_9\}$ and the current N_{EBI} is $\{n_0, n_1, n_3\}$. It can be seen that for an EBO n_8, N_{EBI} is not sufficient to form a cut for n_8 and $\text{TFI}(n_8)$.

To resolve this issue, for each EBO n in S, procedure FIRST-EQ-CUT-TSI(n, N_{BI}) shown in Algorithm 3. computes a set of nodes N_{ret} that, with equivalent nodes in I, form a cut for $(n, \text{TFI}(n)$ along with the original boundary outputs N_{BO}. These nodes are included in the EBIs in the main procedure. Notice that in Line 8, we ensure that the additional EBI is not in $\text{TFO}(N_{\text{BO}})$. Otherwise, we may have an invalid EBI n with a boundary output in $\text{TFI}(n)$. Consider again the example in Fig. 1. After the first call of the procedure FIRST-EQ-CUT-TSI(n_8, N_{BO}), it returns the set of nodes $N_{\text{ret}} = \{n_1, n_2\}$ and the N_{EBI} will be updated by N_{ret} as $\{n_0, n_1, n_2, n_3\}$. It can be checked that the current N_{EBI} is sufficient to form a cut for both $(n_8, \text{TFI}(n_8))$ and $(n_9, \text{TFI}(n_9))$. After the second call FIRST-EQ-CUT-TSI(n_9, N_{BO}), N_{EBI} will remain the same.

B. Recovering Target Instance Boundary

After identifying the extended boundary in S, we can easily reintroduce the boundary of the target instance into the implementation by replacing the logic inside the extended boundary of I. That is, for each node n on the EBO in S, we substitute all its equivalent n' in I with n. And, for each node n on the EBI in S, n is substituted by one of its equivalent nodes in I.

However, before the substitution, we check the validity of the corresponding extended boundary in I. The boundary is invalid if there is an EBI n in the transitive fanout cone of an EBO n'. This could result in combinational loops after the substitution. If this is the case, our method cannot be used to reintroduce the logic in I. We found experimentally that this happens in some rare cases.

Algorithm 1 Identify the extended boundary in specification S using equivalent node pairs between implementation I and S.

procedure IDENTIFY-EXTENDED-BOUNDARY(I, S)
  $N_{\text{BI}}, N_{\text{BO}} \leftarrow \text{BI}(S), \text{BO}(S)$
  IDENTIFY-EQ-NODES(I, S)
  $N_{\text{EBO}}, N_{\text{EBI}} = \emptyset, \emptyset$
  for all node $n \in N_{\text{BO}}$ **do**
    $N_C \leftarrow$ FIRST-EQ-CUT-TFO(n)
    $N_{\text{EBO}} \leftarrow N_{\text{EBO}} \cup N_C$
  for all node $n \in N_{\text{BI}}$ **do**
    $N_C \leftarrow$ FIRST-EQ-CUT-TFI(n)
    $N_{\text{EBI}} \leftarrow N_{\text{EBI}} \cup N_C$
  for all node $n \in N_{\text{EBO}}$ **do**
    $N_C \leftarrow$ FIRST-EQ-CUT-TSI(n, N_{BO})
    $N_{\text{EBI}} \leftarrow N_{\text{EBI}} \cup N_C$
  return $(N_{\text{EBI}}, N_{\text{EBO}})$

Algorithm 2 Compute the set of nodes that, with equivalent nodes in implementation I, form a cut in TFO(n). Returned are the nodes on the first-met cut that satisfies this property.

procedure FIRST-EQ-CUT-TFO(n)
  $N_{\text{ret}} \leftarrow \emptyset$
  $N_{\text{stack}} \leftarrow$ empty_stack()
  N_{stack}.push(n)
  while N_{stack} is not empty **do**
    $n_{\text{cur}} \leftarrow N_{\text{stack}}$.pop()
    if EQ-NODE(n_{cur}) is not $\emptyset$ **then**
      $N_{\text{ret}} \leftarrow N_{\text{ret}} \cup \{n_{\text{cur}}\}$
    else
      for all $n_{\text{next}} \in \text{FO}(n_{\text{cur}})$ **do**
        if n_{next} is not visited **then**
          N_{stack}.push(n_{next})
    Mark n_{cur} as visited
  return N_{ret}

V. Enhancement with Speculative Reduction

One potential weakness of the proposed method is that it relies on SAT sweeping to compute the set of pairs of equivalent nodes spanning across the specification and the implementation. However, it is well known that, if the design contains arithmetic logic structured differently in the specification and in the implementation (for example, an array multiplier in the specification and a Booth multiplier in the implementation), the SAT solver may fail to prove some or all of the intermediate equivalences and the SAT sweeper will not find a substantial number of equivalent nodes pairs. In the worst case, when there are no equivalences or if no equivalences can be proved by SAT, the proposed method defaults to the trivial case when the inputs (outputs) of the extended box are the primary inputs (primary outputs) of the design. This is functionally correct but practically useless because we recover the boundary by replacing the implementation with the specification, without reusing any of the original logic from the implementation.

Speculative reduction (SR) is the method proposed in [2] and developed in [3] for simplifying a combinational or sequential verification miter during SAT sweeping by merging candidate equivalent nodes without proving them. All the node pairs merged without proof during speculative reduction are recorded in the form of a *speculative miter* (SM) to be proved later. The advantage of isolating hard equivalences in the speculative miter is that the miter can be proved independently by the same prover (the SAT solver) running with higher resource limits or by a different prover (for example, by an engine that proves the equivalence of arithmetic circuits by constructing arithmetic polynomials [4]). However, the downside of the speculative reduction is that, if any of the candidate equivalences are disproved, the process has to be redone from scratch while not merging disproved equivalence during sweeping.

We can use speculative reduction to enhance the proposed method as follows. We run SAT sweeping with speculative reduction enabled, resulting in the set of candidate equivalences (as opposed to the set of proved equivalences). These equivalences are used to perform boundary detection, as described in this paper. If the boundary recovery is successful, we use it to perform *logic grafting* by replacing the box logic in the implementation by that of the specification, which often simplifies the miter because both sides of the miter have the same circuit structure within the boundary. If the verification miter is proved under speculation, we still have the task of proving the speculative miter. If this is successful, the hard equivalence checking problem is solved.

Algorithm 3 Compute the set of nodes that makes N_{EBI} forms a complete cut in TFI(n). Returned are the nodes on the first-met cut that satisfies this property.

procedure FIRST-EQ-CUT-TSI(n, N_{BO})
  $N_{\text{ret}} \leftarrow \emptyset$
  $N_{\text{stack}} \leftarrow$ empty_stack()
  N_{stack}.push(n)
  while N_{stack} is not empty **do**
    $n_{\text{cur}} \leftarrow N_{\text{stack}}$.pop()
    if $n_{\text{cur}} \notin N_{\text{BO}}$ **then**
      if $n_{\text{cur}} \notin \text{TFO}(N_{\text{BO}})$ and EQ-NODE(n_{cur}) is not $\emptyset$ **then**
        $N_{\text{ret}} \leftarrow N_{\text{ret}} \cup \{n_{\text{cur}}\}$
      else
        for all $n_{\text{next}} \in \text{FI}(n_{\text{cur}})$ **do**
          if n_{next} is not visited **then**
            N_{stack}.push(n_{next})
    Mark n_{cur} as visited
  return N_{ret}

VI. Experimental Results

A. Boundary Recovery

The benchmarks used to evaluate the proposed methods come from several sources:

1) Designs from the OpenCores repository [5]. In each of the designs considered, one combinational instance or part of the Verilog code is isolated as an instance and used for boundary recovery. The design is read into Yosys [6] twice: to derive the specification, all instances are flattened while the boundary of one instance is maintained; to derive the implementation, all instances are flattened.

2) Arithmetic test cases consisting of additions, multiplications, and possibly some control logic. To generate such benchmarks, we write Verilog description for expressions similar to *(a+b)*c+d* and read them using ABC [7] to generate a hierarchical specification. We then flatten the specification netlist and apply synthesis scripts to generate the implementation.

3) ISCAS'85 hierarchical benchmarks [8]. The implementation is synthesized from the original ISCAS'85 gate-level netlist and the specification is derived from the hierarchical Verilog description. [1]

For the detailed description of each benchmark and the target instances, please refer to Table I. We will make all the used benchmarks publicly available after the double-blind review stage.

In each of the above cases, we synthesize the implementation using different synthesis scripts in ABC shown in Table II and report the results of boundary recovery after using each script.

In Table II, executing each script means running the corresponding "Commands" for "#Iter" times. For the command used, "*&dc2*" performs heavy AIG rewriting. "*&dc4*" and "*&dc3*" both perform AIG balancing and technology mapping/unmapping with different parameters. "*&dc4*" additionally performs shared logic extraction. The main idea is that *script2* applies stronger synthesis than *script1*, and *script3* applies even stronger synthesis than *script2*.

The results are shown in Table III, where the columns "#Spec," "#Inst," "#BI," and "#BO" represents the number of nodes in the specification, the number of nodes in the target instance, the number of boundary inputs, and the number of boundary outputs, respectively.

Then, for each script described in Table II, the columns "#Impl" and 'Equiv' represent the number of nodes in the synthesized implementation and the percentage of the nodes in the specification that have equivalent nodes in the implementation, respectively. In the specification, let the number of nodes, the number of nodes within the target instance, and the number of nodes within the extended instance be M_{spec}, M_{inst}, and M_{ext}, respectively. Then the column "Extend" is the ratio calculated by the following equation:

$$\frac{M_{\text{ext}} - M_{\text{inst}}}{M_{\text{spec}} - M_{\text{inst}}} \tag{4}$$

In other words, "Extend" is 0% if the extended instance is the original target instance and 100% if it includes the whole design.

For each specification netlist, we create three different test cases by specifying different target instances. The names of the test cases "*arith-i-j*," "*oc-i-j*," and "*iscas-i-j*," indicate that they are the j^{th} test cases derived from the i^{th} specification netlist based on the arithmetic, the OpenCores, and the ISCAS benchmarks, respectively.

The column "Extend" of each script shows that the size of the extended instance is larger when a stronger synthesis is applied. Although the extended instance is larger in *script3*, the average values of "Equiv" are both 33% in *script2* and *script3*. This result suggests that, after some synthesis, the number of equivalent pairs may not change a lot, but the changes in the logic structure can make the task of identifying the target instance more difficult.

The test cases *arith-3-1* and *arith-3-2* result in an invalid boundary (boundary containing combination loops) in the implementation synthesized using *script2* and *script3*. This issue is caused by the structural differences, as discussed in Section VII-A. Although further extending the boundary toward primary input is a simple solution, some structural analysis can be done to decide what part of the boundary should be extended. Improving the selection heuristics is deferred to future work.

B. Enhancement using Speculative Reduction

To demonstrate the runtime improvement in boundary recovery with speculative reduction, we perform the experiments on two different benchmarks. The first one contains a set of manually generated arithmetic circuits with structurally different target instances between the specification and the implementation. The second one contains selected designs used in [9] from IWLS 2005 benchmarks [10]. The description of these benchmarks is shown in Table IV. We note that, since the speculative miter is often much smaller than the original design, we perform additional synthesis ("*&syn2; &dc2;*") to reduce the size of the SM before proving it. We do not report the additional synthesis runtime since it is negligible, compared to the time spent on proving the SM.

Table V shows the results with and without speculative reduction (SR). The columns "Bitwidth," "#Spec," "Time" represent the input bitwidth of the target instance, the number of nodes in the specification, the runtime in seconds without SR, respectively The column "Time" and "EQ" under "SR" show the runtime in seconds with SR the equivalent of the designs with SR, respectively. The column "Time" and "EQ" under "SM" show the runtime in seconds of proving SM and the equivalence of SM, respectively. For notations, we use "EQ," "NEQ," and "TO" to represent "equivalent," "not equivalent," and "timeout", respectively. For each test case, the conflict limit of 1000 and the runtime limit of 1000 seconds are used, and the resulting implementation with an extended boundary is verified to be equivalent to the specification.

It should be noted that, in these cases, we could not formally prove the speculative miter since it contains hard arithmetic functions that are not verifiable by the current equivalence checking in ABC. However, we applied extensive random simulation (16 words, 10000 rounds) to the speculative miters and did not disprove them. This gives us some degree of confidence about their correctness because random simulation is powerful enough to find bugs in arithmetic logic. We plan to work on formal equivalence checking of hard arithmetic functions as part of future work.

It can be seen that, for circuits that are hard for SAT sweeping, SR can be used to obtain a solution faster. In particular, column "PO" shows that, except for case *n1* with bitwidth 32, all primary outputs are equivalent under speculation, and a boundary recovery solution can be found in all cases.

Although we currently validate the correctness of the majority of speculative miters only using simulation, the results for

[1]The hierarchical Verilog description can be found here https://web.eecs.umich.edu/~jhayes/iscas.restore/benchmark.html

TABLE I: The description of the benchmarks used in Table III

Design	Description	Case	Target Instance
arith-1	6-bit arithmetic expression, $(a+b)*c+d$	1	the multiplier
		2	the lowest 5 bits of the multiplier
		3	the first adder
arith-2	6-bit complex arithmetic expression	1	a multiplier
		2	the lowest 3 bits of a multiplier
		3	the lowest 5 bits of a multiplier
arith-3	16-bit complex arithmetic expression with control logics	1	a subtractor
		2	an arithmetic block with multiplication and addition
		3	an adder with some control logics
oc-1	module "dcach_write" from design "ao486", a processor	1	the lowest 2 bits of signal "line_merged"
		2	signal "line_merged"
		3	signal "write_burst_byteenable_1"
oc-3	module "crp" from design "des", a crypto core	1	module "sbox8"
		2	part of the module "sbox3"
		3	module "sbox4"
oc-4	module "CPE" from design "nova", a video controller	1	one of the module "CPE_base"
		2	one line in one of the module "CPE_base"
		3	one line in one of the module "CPE_base"
oc-5	module "Inter_pred_CPE" from design "nova", a video controller	1	one of the module "CPE"
		2	one line in one of the module "CPE"
		3	one line in one of the module "CPE"
iscas-1	"c2670," a 12-bit ALU and controller	1	instance "UM4_4" of module "CLA12_XY"
		2	instance "UM4_1" of module "MaskBus"
		3	instance "UM6_0" of module "Mux9bit_2_1"
iscas-2	"c5315," a 9-bit ALU	1	instance "M4" of module "CalcParity"
		2	instance "M5" of module "MuxesPar_4"
		3	instance "M11" of module "ZeroFlags"

TABLE II: The scripts for synthesizing implementations.

Script	Commands	#Iter
script1	*&dc2;*	2
script2	*&dc3; &dc2;*	2
script3	*&dc3; &dc2; &dc4; &dc2;*	3

examples *n4* and *n5* indicate that, at least in some cases, the synthesized SM is easier to prove, compared to SAT sweeping the original design.

C. A Case Study of Logic Block Annotation

This experiment is a case study of logic block annotation mentioned in Section I for the design *oc-1-1*, demonstrating the feasibility of annotating the relevant region in the specification netlist with the physical information. Several studies on placement-aware logic synthesis [11], [12], [13] have shown the importance of the placement information during logic synthesis. Therefore, as a proof of concept, we consider the physical information in placement, which includes cell positions and wirelength estimation of certain nets. We follow the procedure described in Section VI-A to derive the specification netlist G_1 and the implementation netlist G_2 with *script1*. G_2 netlist is then transformed into a physical netlist G_p in the Bookshelf format, where the and-gate is a standard cell with two input pins and one output pin. For each node n, its output pin and the connected input pins of FO(n) form a *net*. Note that we have to remember the mapping of the nodes and the cells between G_2 and G_p in order to map the information from G_p to G_2. G_p is then given to the placer NTUPlace3 [14] to derive the placement result. With the placement result and the mapping between G_2 and G_p, we are able to label the nodes in G_2 with their corresponding positions in G_p after placement. Last, given the relevant region in G_1, we perform boundary recovery on G_1 and G_2. The extended boundary nodes in G_1 are annotated with the physical locations based on the boundary nodes in G_2. The wirelength information in the region of G_1 can be computed by summing up the half-perimeter wirelength (HPWL) of all the nets in the extended boundary in G_2.

VII. Discussion

In this section, we discuss several limitations of the current boundary recovery method and their possible solutions.

A. Structural Differences

The first limitation of our method stems from the assumption that the boundary nodes would be presented as the equivalent nodes between the implementation and the specification. In some scenarios, the netlist may be largely restructured such that the target instance still exists in the implementation but does not have equivalent boundary nodes. For example, in Fig. 3, the left-hand side is the specification netlist S and the right-hand side is the implementation netlist I with the multiplier (MULT) being the target instance. It can be checked that, although both netlists implement the same function and the boundary of the target instance present in both netlists, the BOs in S are not equivalent to that in I and the left part of the BIs in S is also not equivalent to that in I. If we apply the proposed boundary recovery in this case, we will detect the POs as the EBOs and the PIs as the EBIs, resulting in the trivial extended boundary, even if the exact boundary is present in S and I.

In other cases involving complex logic structures, we may end up with an EBI whose TFI contains an EBO, resulting in

TABLE III: Boundary recovery results and benchmarks statistics with different synthesis scripts.

Case	#Spec	#Inst	#BI	#BO	*script1*			*script2*			*script3*		
					#Impl	Equiv	Extend	#Impl	Equiv	Extend	#Impl	Equiv	Extend
arith-1-1	423	307	12	12	391	86%	2.59%	549	36%	37.93%	589	41%	60.34%
arith-1-2	509	96	10	5	391	83%	0.00%	549	36%	2.18%	589	40%	2.18%
arith-1-3	423	37	12	12	391	86%	0.00%	549	36%	0.00%	589	41%	0.00%
arith-2-1	752	307	12	12	642	72%	0.67%	850	32%	37.53%	995	35%	60.22%
arith-2-2	405	34	6	3	318	63%	0.00%	369	34%	2.43%	458	38%	2.43%
arith-2-3	501	96	10	5	407	63%	4.44%	474	32%	10.12%	569	36%	10.12%
arith-3-1	4529	107	32	16	3301	68%	1.42%	3932	32%	8.75%	4073	36%	8.41%
arith-3-2	4628	698	32	16	3301	68%	0.08%	3932	32%	7.74%	4073	35%	5.37%
arith-3-3	4529	125	35	16	3301	68%	0.00%	3932	32%	0.00%	4073	36%	1.45%
oc-1-1	22475	124	32	2	5159	49%	0.00%	4704	44%	0.00%	3475	34%	0.00%
oc-1-2	22359	21981	195	128	5059	20%	0.00%	4765	11%	0.00%	2901	8%	0.00%
oc-1-3	22357	23	5	4	5059	20%	0.00%	4765	11%	0.00%	2901	8%	0.00%
oc-3-1	2047	239	6	4	1523	35%	0.00%	1305	10%	0.00%	1315	13%	0.00%
oc-3-2	1995	16	3	2	1449	32%	0.00%	1264	10%	0.00%	1305	15%	0.00%
oc-3-3	2047	238	6	4	1523	35%	0.00%	1305	10%	0.00%	1315	13%	0.00%
oc-4-1	1773	441	16	14	1767	95%	0.83%	2386	45%	8.56%	2686	43%	23.12%
oc-4-2	1841	69	9	9	1828	94%	0.34%	2484	44%	2.09%	2800	43%	2.71%
oc-4-3	1834	62	12	9	1812	94%	0.56%	2449	44%	1.81%	2965	42%	2.88%
oc-5-1	6749	1773	46	8	5663	97%	0.00%	7830	50%	0.00%	9031	45%	0.00%
oc-5-2	6775	372	14	14	6006	97%	1.42%	8254	51%	1.58%	9562	47%	1.58%
oc-5-3	6473	70	22	14	5729	97%	1.42%	7986	50%	1.95%	9329	48%	3.40%
iscas-1-1	794	82	24	1	553	33%	3.79%	562	23%	4.49%	535	23%	5.76%
iscas-1-2	707	14	15	12	553	35%	13.71%	562	24%	14.43%	535	24%	15.58%
iscas-1-3	707	27	19	9	553	34%	0.00%	562	23%	0.00%	535	24%	12.65%
iscas-2-1	1810	307	45	2	1364	34%	0.00%	1336	27%	0.00%	1295	25%	0.00%
iscas-2-2	1724	46	15	4	1364	34%	0.00%	1336	27%	0.00%	1295	24%	0.00%
iscas-2-3	1724	32	36	4	1364	34%	0.00%	1336	27%	0.24%	1295	24%	0.47%
Average						68%	0.66%		33%	5.84%		33%	8.77%

TABLE IV: The benchmarks for speculative reduction.

Design	Specification	Implementation
n1	"$(((a+b)*c)+d)$"	"$((c*(a+b))+d)$" synthesized using "*&dc3; &dc2*"
n2	"$((((e+f)*(a+b))+((e+f)*c))+d)$"	"$(((e+f)*((a+b)+c))+d)$" synthesized using "*&dc3; &dc2*"
n3	"$((m?0:((a+b)*c))+d)$"	"$(((m?0:(a+b))*c)+d)$" synthesized using "*&dc3; &dc2*"
n4	Design "pci_bridge32" synthesized using "*&dc2; &syn2;*"	Design "pci_bridge32" synthesized using "*ifraig -s;*"
n5	Design "netcard" synthesized using "*&dc2; &syn2;*"	Design "netcard" synthesized using "*ifraig -s;*"

TABLE V: The result of boundary recovery with speculative reduction.

Design	BW	#Spec	Time	SR		SM	
				Time	EQ	Time	EQ
n1	8	678	56.52	0.36	EQ	68.96	EQ
n1	16	2218	TO	1.17	EQ	TO	N/A
n1	32	7794	TO	4.12	NEQ	N/A	N/A
n2	8	1356	TO	0.37	EQ	TO	N/A
n2	16	4436	TO	1.02	EQ	TO	N/A
n2	32	15588	TO	3.11	EQ	TO	N/A
n3	8	886	0.45	0.02	EQ	0.85	EQ
n3	16	2250	109.61	1.2	EQ	TO	N/A
n3	32	7858	TO	4.54	EQ	TO	N/A
n4	N/A	16392	441.17	0.14	EQ	0.01	EQ
n5	N/A	519206	TO	8.42	EQ	0.35	EQ

a combinational loop. The issues due to structural differences can be alleviated by relaxing the EBI towards the PIs or POs.

B. Satisfiability Don't Cares

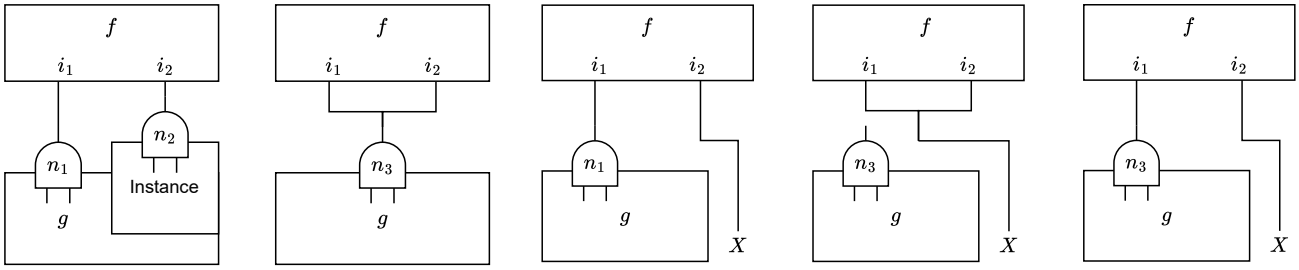
When the proposed method for boundary recovery is used in the ECO applications, the correct solution is guaranteed when the logic cones outside the target instance in the specification and in the implementation are equivalent. Therefore, our method may not be applicable when the implementation is synthesized using *satisfiability don't-cares* (SDCs) produced by the target instance.

This happens when certain assignments never appear at the BOs, becoming SDCs for the TFO cones of the BOs, providing the synthesis tool with flexibility for optimization.

An example is shown in Fig. 4. In these two netlists, the transitive fanout cone of the boundary outputs are an XOR gate and an OR gate, respectively. While the XOR gate and the OR gate are in general not equivalent, the two netlists could be equivalent if the assignment $(1, 1)$ never appears on the BOs. If such a condition happens, although we could still identify the boundary using the equivalent nodes, the two netlists may not be equivalent after replacing the logic inside the extended boundary of the implementation with the patch. As mentioned in the previous subsection, this issue can also be tackled by further extending the boundary towards the primary input or the primary outputs.

C. Logic Sharing

Logic sharing in the netlist may lead to non-equivalent logic cones in the implementation and in the specification outside of the target instance. This happens when one boundary node in the implementation is equivalent to multiple nodes in the specification. Because the implementation is assumed to undergo some optimization and is usually more compact than the specification, some equivalent nodes will likely be merged in the implementation. When we change the functionality of



(a) Specification S (b) Implementation I (c) L_S from S (d) L_I from I (e) L'_I from I

Fig. 2: An example of non-equivalent logics outside the target instances caused by logic sharing. L_S (L_I) is derived from S (I) by replacing the boundary outputs with additional free inputs X. While L_I is not equivalent to L_S, L'_I keeps the original connection for i_i and is equivalent to L_S.

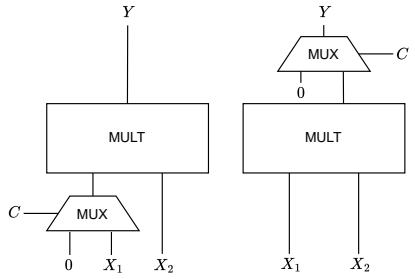


Fig. 3: The specification and implementation netlists with non-equivalent boundary nodes.

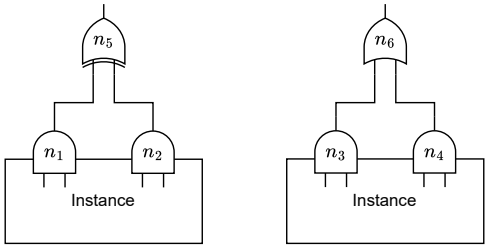


Fig. 4: An example of the specification netlist and the synthesized implementation netlist after exploiting satisfiability don't-cares.

the target instance for the specification, we expect that only the fanin cones of the equivalent nodes at the boundary should change. However, since all the equivalent nodes are merged into one node in the implementation, our approach patches the changed logic of that node in the implementation. Alternatively, the patch will change the logic of all those equivalent nodes, resulting in the non-equivalence between the specification and the implementation.

This is illustrated in Fig. 2 where Fig. 2a is the specification and Fig. 2b is the implementation. If nodes n_1 and n_2 in the specification and n_3 in the implementation are equivalent, when we choose n_3 as the boundary output, the logic cones on the outside of the boundary are not equivalent, as shown in Fig. 2c and Fig. 2d. To make these logic cones equivalent, the connection between the nodes n_3 and i_1 should be kept in the implementation, as shown in Fig. 2e.

This issue can be alleviated by considering structural information when choosing the corresponding equivalent node.

VIII. Conclusions

The paper presents a novel method for post-processing of a flat netlist to recover a hierarchical boundary that existed in the specification but was lost during elaboration and further obfuscated during logic optimization. The method works even in the case when the flat netlist does not have nodes corresponding to those originally present on the boundary in the specification. The computation uses a robust resource-aware implementation of SAT sweeping and does not rely on other reverse engineering techniques [15], [16], [17].

It can be observed that the boundary detection tends to be harder in the following cases: when a more aggressive synthesis script is used, when a more global restructuring is performed, and when the logic of the implementation is relatively deep and/or highly redundant. In all of the above cases, it is likely that logic optimization leads to a substantial simplification across the boundary, which obfuscates the boundary and makes the recovery harder.

Experimental results confirm that the proposed method can successfully recover hierarchical boundaries in many cases. The method can be useful in a number of practical scenarios, in particular, when a designer needs to update the result of synthesis for a specific module or when a verification engineer tries to simplify equivalence checking by using additional constraints, such as uninterpreted function constraints, attached to the cut points present in the specification but missing in the implementation.

Future work may focus on increasing the robustness of the method, which at present still fails on some examples when high-effort synthesis makes significant structural changes. In particular, the situation when one node in the implementation is equivalent to several nodes in the specification can be addressed. Another extension is make the method work for sequential circuits by relying on sequential SAT sweeping for detecting node equivalences [18].

Acknowledgments

This work was supported in part by the National Science and Technology Council of Taiwan under grant NSTC 111-2923-E-002-013-MY3, and the NTU Center of Data Intelligence:

Technologies, Applications, and Systems under grant NTU-113L900903. This research at UC Berkeley was supported in part by the NSA grant "Novel methods for synthesis and verification in cryptanalytic applications", the SRC Contract 3173.001 "Standardizing Boolean transforms to improve quality and runtime of CAD tools", and donations from AMD, Siemens, and Synopsys.

REFERENCES

[1] A. Mishchenko, S. Chatterjee, R. Jiang, and R. K. Brayton, "FRAIGs: A unifying representation for logic synthesis and verification," UC Berkeley, Tech. Rep., 2005.

[2] H. Mony, J. Baumgartner, V. Paruthi, and R. Kanzelman, "Exploiting suspected redundancy without proving it," in *Proceedings of Design Automation Conference*, 2005, pp. 463–466.

[3] H. Mony, J. Baumgartner, A. Mishchenko, and R. Brayton, "Speculative reduction-based scalable redundancy identification," in *Proceedings of Design, Automation & Test in Europe Conference*, 2009, pp. 1674–1679.

[4] D. Kaufmann, A. Biere, and M. Kauers, "Verifying large multipliers by combining sat and computer algebra," in *Proceedings of Formal Methods in Computer-Aided Design*, 2019, pp. 28–36.

[5] *Opencores*. [Online]. Available: https://opencores.org/.

[6] C. Wolf, *Yosys open synthesis suite*. [Online]. Available: https://yosyshq.net/yosys/.

[7] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Proceedings of the International Conference on Computer-Aided Verification*, 2010, pp. 24–40.

[8] M. C. Hansen, H. Yalcin, and J. P. Hayes, "Unveiling the iscas-85 benchmarks: A case study in reverse engineering," *IEEE Design & Test of Computers*, vol. 16, no. 3, pp. 72–80, 1999.

[9] S.-Y. Lee, H. Riener, A. Mishchenko, R. K. Brayton, and G. De Micheli, "A simulation-guided paradigm for logic synthesis and verification," *IEEE Transactions on CAD of Integrated Circuits and Systems*, vol. 41, no. 8, pp. 2573–2586, 2022.

[10] C. Albrecht, *Iwls 2005 benchmarks*. [Online]. Available: https://iwls.org/iwls2005/benchmarks.html.

[11] H. Zheng, S. T. Gurumani, K. Rupnow, and D. Chen, "Fast and effective placement and routing directed high-level synthesis for fpgas," in *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2014, pp. 1–10.

[12] J. Nam and D. Hyun, "Bayesian optimization for parameter tuning in placement-aware logic synthesis," in *2023 20th International SoC Design Conference (ISOCC)*, 2023, pp. 353–354.

[13] D. Hyun, Y. Fan, and Y. Shin, "Accurate wirelength prediction for placement-aware synthesis through machine learning," in *2019 Design, Automation and Test in Europe Conference (DATE)*, 2019, pp. 324–327.

[14] T.-C. Chen, Z.-W. Jiang, T.-C. Hsu, H.-C. Chen, and Y.-W. Chang, "Ntuplace3: An analytical placer for large-scale mixed-size designs with preplaced blocks and density constraints," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 7, pp. 1228–1240, 2008.

[15] W. Li, Z. Wasson, and S. A. Seshia, "Reverse engineering circuits using behavioral pattern mining," in *Proceedings of the International Symposium on Hardware-Oriented Security and Trust*, 2012, pp. 83–88.

[16] M. Soeken, B. Sterin, R. Drechsler, and R. Brayton, "Simulation graphs for reverse engineering," in *Proceedings of Formal Methods in Computer-Aided Design*, 2015, pp. 152–159.

[17] A. Mishchenko, B. Sterin, and R. Brayton, "Structural reverse engineering of arithmetic circuits," UC Berkeley, Tech. Rep., 2017.

[18] A. Mishchenko, M. Case, R. Brayton, and S. Jang, "Scalable and scalably-verifiable sequential synthesis," in *Proceedings of the International Conference on Computer-Aided Design*, 2008, pp. 234–241.