Facilitating Timing Debug by Logic Path Correspondence

Oshri Adler, Eli Arbel, Ilia Averbouch, Ilan Beer, Inna Grijnevitch
IBM Research Lab, Mount Carmel, Haifa, Israel, 31905
{oshria,arbel,iliaa,beer,innag}@il.ibm.com

***Abstract*—Synthesis tools for high-performance VLSI designs employ aggressive logic optimization techniques in order to meet physical requirements such as area and cycle time. During these optimizations, the original structure of the design, which is usually written in a hardware description language (HDL), is lost. It is difficult, and often impossible, to relate signals after synthesis to the original signals in the HDL code. Some signals only lose their names while for others there are no equivalent counterparts in the design after synthesis. Debugging timing problems is based on timing reports which are usually represented in terms of the post-synthesis design. Hence, it is difficult to relate critical paths in the timing reports to the relevant paths in the HDL code when a logic fix is needed. In this paper, we propose a different approach for dealing with the correspondence problem: instead of trying to relate signals we relate paths. Given a critical path in a post-synthesis representation, our method is able to find all corresponding paths in the pre-synthesis (HDL) representation. As a result, locating the parts in the HDL which are relevant to the given timing problem becomes trivial. A novel Sat-based algorithm for dealing with the path-correspondence problem is described. Experimental results on various industrial high-end processor designs show the effectiveness of our algorithm in substantially reducing the amount of paths in the HDL which one will have to consider when debugging a given critical path.**

I. Introduction

VLSI design is a multi-dimensional problem in which one tries to optimize a given design against multiple objectives such as area, power and performance. The purpose of a given design dictates how each of the metrics the design is measured against are weighed. For example, an SoC targeted for a mobile device will be highly optimized for low power consumption while possibly trading off performance. On the other hand, in the area of high-end processor design, performance is one of the major objectives. As high-performance hardware entails very short cycle times, a major challenge in designing such systems is meeting timing constraints. This usually involves synthesizing the logic, analyzing and debugging timing results and fixing timing issues. Hence, one of the most time consuming phases in high-end processor design methodology is the one of *timing closure* [1].

In a typical timing closure methodology, a circuit designer first analyzes post-synthesis timing reports in an attempt to solve as many timing violations as possible at the circuit level. This may involve looking at the RTL (Register Transfer Level) as well, if the problem is suspected to be in the way the logic was written. In the realization that fixing the timing problem calls for a logic change, the critical path is then passed to the logic designer owning that piece of logic. A fundamental issue in this methodology is that the timing results are expressed in terms of post-synthesis netlist, while the logic is expressed in terms of RTL. This poses a real challenge: finding the correlation between synthesis results and the RTL. This is especially true when aggressive synthesis optimizations and netlist transformations are used, in which case usually little to no effort is spent in maintaining the original signal names or logic structure. Thus, a way to relate the post-synthesis results to the RTL representation is required. This will allow the circuit designer, which is not familiar with the logic, to easily identify the parts in the RTL which are relevant to the current timing problem. Similarly, the logic designer, which is also not familiar with all the physical constraints and the way synthesis was run, can greatly benefit from such mapping.

978-3-9815370-2-4/DATE14/

Mapping synthesis results back to the RTL can be done manually. However, in a reality where design sizes are getting increasingly larger while time to market is shrinking, automated techniques which facilitate timing closure are required. One approach for dealing with the problem is to rely on signal correspondence, that is, back-annotate timing information on a per-signal basis. The problem with this approach is that commonly used logic optimization techniques, such as those described in [2], [3] often destroy the netlist structure and change signal names to the extent that signal correspondence becomes very hard and even impossible. Another approach, described by Mishchenko and Brayton in [4] suggests a framework for recording history of synthesis transformations in order to facilitate verification tasks. This approach may be used as a basis for automatic back-annotation, however, many current design flows are yet to employ such capabilities.

In this paper we suggest a new approach for finding correlation between a post-synthesis and pre-synthesis netlist of the same design. Instead of relying on signal correspondence or structural similarities to perform back-annotation, our method works in the path level. Given a path in one representation level of a design, our proposed algorithm automatically finds the corresponding logic paths in the other representation level. The proposed analysis is based solely on functional properties of the logic, which are invariant to logic structure changes and name mangling. The algorithm is based on analyzing the functional behavior of the design in its different representation levels and finding paths which have similar sensitization properties. Our algorithm is implemented using a Sat solver, thus it is highly efficient, even on large and complex designs. Experimental results show that our algorithm is able to substantially reduce the number of paths which are needed to be considered, compared to all possible paths a timing problem may involve with.

AND	0	1	$\bar{v}$	v
0	0	0	0	0
1	0	1	$\bar{v}$	v
$\bar{v}$	0	$\bar{v}$	$\bar{v}$	0
v	0	v	0	v

XOR	0	1	$\bar{v}$	v
0	0	1	$\bar{v}$	v
1	1	0	v	$\bar{v}$
$\bar{v}$	0	v	0	1
v	0	$\bar{v}$	1	0

Fig. 1. Examples of truth tables of simple logic operator for symbolic representation

The rest of the paper is structured as follows. In Section II we give some preliminaries. A high-level description of our path correspondence algorithm is given in Section III followed by the details of implementing the algorithm with a Sat solver in Section IV. Results of applying it on some real high-end processor design examples are given in Section V and conclusions are drawn in Section VI.

II. Preliminaries

In this section we briefly describe some of the concepts used in this paper. A *netlist* $C = (V, E)$ is a graph representation of the logic which consists of gates (V) and wires (E). A netlist may describe a combinational circuit, in which case V consists of Boolean gates only (AND, OR, XOR etc.), or it can describe a sequential circuit where V also includes sequential elements. We use the term *latch* to describe a memory element (e.g. flip flop). The edges E of the graph represent wires and can be classified as internal (gate to gate), primary inputs or primary outputs. The terms wires, nets and signals are interchangeably used in this paper. A *pre-synthesis* netlist is the netlist generated after compilation and elaboration of the RTL (e.g. VHDL or Verilog). The pre-synthesis netlist may be comprised of high-level gates such as muxes and adders and its structure is similar to the RTL structure. Furthermore, as the pre-synthesis netlist usually undergo very minor optimizations, it is fairly easy to correlate the signals and structures in the pre-synthesis netlist to their RTL counterparts.

A *logic path* in a given netlist is an ordered list of signals $(s_1, s_2, ..., s_n)$ where s_i is an input signal of some combinational gate whose output is s_{i+1}, for each $i \in [1, n-1]$. The end-points of the path, s_1 and s_n, are called the starting point and the ending point of the path, respectively. In the context of timing debug, the starting point of a given path corresponds to either a latch output or a primary input, and the ending point of the path corresponds to either a latch input or a primary output.

Boolean Satisifiability is the problem of deciding whether there exists a truth assignment to a given Boolean formula. A *Sat-solver* is able to find truth assignments to Boolean formulae, commonly given in CNF (Conjunctive Normal Form). A CNF formula is a conjunction of clauses, where each clause is a disjunction of literals, and each literal is an instance of a variable or its negation. If a given Boolean formula has a truth assignment, we say it is *satisfiable* (SAT), otherwise it is *unsatisfiable* (UNSAT).

Symbolic evaluation is a method for representing multiple valuations of a given Boolean formula by assigning several of its variables with symbols instead of concrete Boolean values. A netlist representing a Boolean network can be extended to support symbolic evaluation by mean of *dual-rail encoding* [5]. Each net defined over the Boolean domain $\{0, 1\}$ is represented by 2 nets in the dual-rail encoding for holding the symbolic domain values $\{0, 1, v, \bar{v}\}$, where v and $\bar{v}$ are symbols representing a variable and its negation, respectively. Given a net n in the Boolean netlist, we denote n_d and n_v as n's domain and value bits, respectively, allowing us to represent a symbolic value as a combination of the domain and value bits. For example, consider $\delta : \{0, 1\}^2 \rightarrow \{0, 1, v, \bar{v}\}$, an encoding function from the Boolean domain to the symbolic domain. Then one possible encoding may be: $(0, 0) \rightarrow (0), (0, 1) \rightarrow (1), (1, 0) \rightarrow (\bar{v}), (1, 1) \rightarrow (v)$. In this encoding the first bit acts as the domain bit, signifying whether the encoded value is Boolean or symbolic, and the second bit signifies the polarity, or the concrete value under the given domain bit. The truth tables of logic gates are also extended to support correct computation over the symbolic domain as can be shown in Figure 1.

Finally, the *Boolean difference* of a function f w.r.t a variable x is defined as $\frac{\partial f}{\partial x} \equiv f|_{x=0} \oplus f|_{x=1}$, that is the negative cofactor of f w.r.t x Xor'ed with the positive cofactor of f w.r.t x. A *Path sensitization function* [6] is a Boolean function representing the conditions (e.g. the set of all inputs vectors) under which a signal can propagate from the starting point of the path to its ending point through the signals comprising the path.

III. Path Correspondence Algorithm

In this section we provide a high-level description of our path-correspondence algorithm. Details of our Sat-based implementation are given in Section IV. We describe the algorithm in the context of classical timing debug methodology, that is considering a path given in post-synthesis netlist (a critical path) and finding its corresponding paths in pre-synthesis netlist (RTL).

In order for our path-correspondence algorithm to work properly, it is assumed that combinational equivalency [7], [8] holds between the pre-synthesis and the post-synthesis netlist representations of the design to be processed. This assumption imposes one-to-one latch mapping, as well as primary input and output mapping between the given representation levels of the design. This is a fair assumption in many synthesis flows where the methodology does not permit sequential optimizations in order to ease ECO [9] and post-silicon validation, and to reduce verification complexity [6].

A. Extracting the Path Sensitization Function

The properties of Boolean functions of signals in a given netlist remain the same, regardless of the combinational transformation used to optimize the logic. For example, the conditions under which a given function evaluates to true, the number of truth assignments and the probability of the function being one, are all invariants of the function. In particular, the observability conditions of an input variable w.r.t a given function do not depend on logic implementation. More formally, given a function f and some input variable of f denoted as x, $\frac{\partial f}{\partial x}$ is invariant to the logic structure implementing f. Hence, path sensitization functions lie in the heart of our path-correspondence algorithm and are key in making our algorithm robust to netlist transformations.

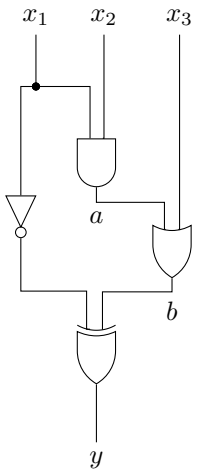

Fig. 2. An example path: (x_2, a, b, y)

Given a critical path in the post-synthesis level of the design, the first step of our algorithm is to extract the path sensitization function of this path. The path sensitization function of a given path can be calculated by conjuncting the Boolean difference functions of each path signal w.r.t the variable corresponding to starting signal of the path.

Consider the path (x_2, a, b, y) in Figure 2. We compute the Boolean difference functions of the signals along the path as follows: $\frac{\partial x_2}{\partial x_2} = 1$, $\frac{\partial a}{\partial x_2} = (x_1 \wedge 0) \oplus (x_1 \wedge 1) = x_1$, $\frac{\partial b}{\partial x_2} = ((x_1 \wedge 0) \vee x_3) \oplus ((x_1 \wedge 1) \vee x_3) = x_1 \wedge \neg x_3$ and $\frac{\partial y}{\partial x_2} = ((x_1 \wedge 0) \vee x_3) \oplus ((x_1 \wedge 1) \vee x_3) = x_1 \wedge \neg x_3$. Conjuctiong the Boolean difference functions, we get $h^{post} = \frac{\partial x_2}{\partial x_2} \wedge \frac{\partial a}{\partial x_2} \wedge \frac{\partial b}{\partial x_2} \wedge \frac{\partial y}{\partial x_2} = x_1 \wedge \neg x_3$.

Note that although in the example above $h^{post} = \frac{\partial y}{\partial x_2}$, the Boolean difference of y w.r.t x_2 only represents the conditions that x_2 is observable at y. However it does not constrain x_2 to propagate through a particular path, but rather through any possible path from x_2 to y. This is why we use the conjunction of the Boolean difference functions of all the signals along the path. In addition, the function h^{post} represents the necessary conditions for activating the given path, however it does not constrain signal x_2 from propagating to y through other paths.

Recall that our goal is to find which paths in the pre-synthesis level are sensitized by h^{post}. However, h^{post} is expressed in terms of the post-synthesis netlist signal variables (x_1, x_2 and x_3 in the example given in Figure 2). Thus h^{post} should be mapped to be over the pre-synthesis netlist variables. Recall that the pre-synthesis netlist is a graph representation of the RTL in which signal names are preserved and its logic structure is similar to the RTL structure. Since combinational equivalence holds between the 2 representation levels, this mapping is done trivially by replacing each occurrence of a post-synthesis variable (corresponding to either a primary input or a latch output) with its corresponding pre-synthesis variable. We denote the mapped path sensitization function as h^{pre}.

B. Finding Corresponding Paths

Given the path sensitization function h^{pre}, expressed in terms of the pre-synthesis netlist according to the previous section, the algorithm begins by checking which paths in the pre-synthesis netlist are sensitized by h^{pre}. Denote S and T as the pre-synthesis signals which correspond to the starting and ending points of the path, respectively. Note that since the two representation levels of the design are combinationally equivalent, S and T can be easily located. Only paths starting with S and ending in T are candidates for being corresponding to the given path. Other paths need not be checked at all. Thus, only a small portion of the logic, called the *diamond* of S and T, is analyzed in this step. The diamond of S and T is defined as a set of signals obtained by taking the intersection of the combinational fanout signals of S and the combinational fanin signals of T, both in the pre-synthesis netlist. See Figure 3 for an illustration.

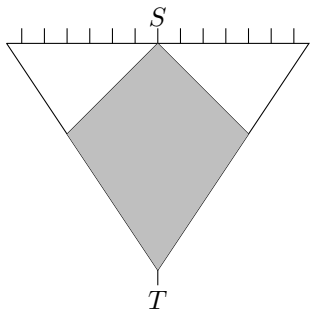

Fig. 3. Diamond of S and T: Intersection of fanout cone of S and fanin cone of T (gray area)

For a path to be sensitized by h^{pre}, it means that S is observable to T through all the path signals. Thus in order to check which paths are sensitized, the algorithm checks at which signals in the diamond of S and T, S is observable under h^{pre}. This is accomplished as followed: for each net n in the diamond of S and T, $f(n)$, the Boolean function representing n is computed. Then the algorithm checks whether:

$$\frac{\partial g}{\partial S} \neq 0 \tag{1}$$

where $g \equiv f(n) \wedge h^{pre}$.

In case Equation 1 evaluates to *false*, S cannot propagate to n under h^{pre}, thus all paths starting with S and passing through n cannot correspond to the original path. Otherwise, there exists a condition under which S is observable at n, making n a candidate net for being part of some pre-synthesis path which correspond to the post-synthesis one. In rare cases the check of Equation 1 may result *false* for all the nets in the diamond of S and T. This can occur due to S not being observable at T, i.e. there are some redundancies in the circuit, or because h^{post} has no truth assignments. In the latter case the algorithm finds a candidate false path, i.e. the path is functionally false, but not necessarily a real false path as our path correspondence algorithm does not consider wire delays [10].

Once all the nets in the diamond of S and T are classified according to the check given in Equation 1, the final step is to extract the actual paths which start with S and end in T which correspond to the original critical path. This is done by traversing the pre-synthesis netlist in a DFS order starting from S and continuing the traversal only on nets for which Equation 1 evaluates to *true*. Each time we encounter T during

the DFS traversal, the list of signals of the current DFS stack comprise one corresponding path. We collect all paths this way until the DFS traversal is complete.

IV. Sat-based Implementation

The path-correspondence algorithm described in Section III heavily relies on Boolean reasoning. In particular, it employs the notion of symbolic evaluation with the use of symbolic path sensitization function and the Boolean difference operator. The algorithm can be readily implemented using Binary Decision Diagrams (BDDs) [11]. However, in order to achieve better efficiency and scalability, we describe in this section how those ideas can be implemented using a Sat solver.

The main challenge in implementing the algorithm using a Sat solver is to efficiently encode the path sensitization function, as well as the conditions which express the checks according to Equation 1. Naively, one could implement the Boolean difference operations which are involved in the computation directly according to definition, that is by Xoring the two cofactors of each Boolean function. This is, however, highly inefficient, especially when a distinct application of the Boolean difference operator is required for each path signal and for each net in the diamond of S and T. Instead, in our implementation we use dual-rail encoding [5] for describing the observability constraints expressed by the Boolean difference formulation. Both the pre-synthesis netlist and the post-synthesis netlist are first transformed to dual-rail encoding over the domain $\{0, 1, \bar{v}, v\}$ to efficiently support symbolic evaluation using a Sat solver.

The general flow of our Sat-based path-correspondence algorithm is depicted in Algorithm 1. The inputs to the algorithm are the two netlist representations already in dual-rail encoding, a mapping between netlists input and latch points (M) and the path to be analyzed (P). The algorithm produces a set containing all nets in the diamond of S and T which pass the check of Equation 1. The fanin cone in the post-synthesis netlist is first encoded to CNF in lines 3-4. We used the logic clausification scheme described by Tseitin in [12] in our implementation. The constraints expressed by the path sensitization function are encoded in lines 5-7, by constraining the dual-rail domain bit (see Section II) of each path signal to be 1, that is enforcing it to be a symbol. Note that S, the starting point of the path is also included in P. Constraining all the domain bits of the path signals to 1 is equivalent to requiring that S will propagate through all those path signals, equivalently to constructing the conjunction of Boolean difference functions of the path signals. The polarity of S (i.e. its value bit) remains unconstrained.

Next, the fanin cone of the signal T in the pre-synthesis netlist is encoded to CNF in line 10. Now we have two sets of CNF clauses: the first describes the path-sensitization function and the second originated from the pre-synthesis fanin cone of T. We correlate the two sets using the input and latch output signals map M in lines 13-15, by encoding a pairwise equivalence constraint for each logic input which appears in the two cones of logic. Note that $preIns \triangle preOuts$ (the symmetric difference) may not necessarily be empty, that is some inputs may appear in one cone of logic but not in the other. In this case those input nets are left uncorrelated and

Algorithm 1 Sat-based path correspondence

Input: *preNL*, *postNL*, P : critical path, M : $\{E \times E\}$ /* input and latch boundary mapping */
Output: σ : a set of nets

1: $\sigma \leftarrow \emptyset$
2: *CNF* $\leftarrow \emptyset$
3: $T_{post} \leftarrow$ *endingPointOf*$(P, postNL)$
4: *CNF* $\leftarrow$ *CNF* $\cup$ *encodeToCnf*(T_{post})
5: **for all** $n \in P$ **do**
6: *CNF* $\leftarrow$ *CNF* $\cup$ (*domainBitOf*$(n) = 1$)
7: **end for**
8: $S_{pre} \leftarrow$ *startingPointOf*$(P, preNL, M)$
9: $T_{pre} \leftarrow$ *endingPointOf*$(P, preNL, M)$
10: *CNF* $\leftarrow$ *CNF* $\cup$ *encodeToCnf*(T_{pre})
11: *preIns* $\leftarrow$ *inputsOf*(T_{pre}) /* combinational input nets */
12: *postIns* $\leftarrow$ *inputsOf*(T_{post}) /* combinational input nets */
13: **for all** $i \in$ *preIns* $\cap$ *postIns*) **do**
14: *CNF* $\leftarrow$ *CNF* $\cup$ *correlateInput*$(i, M, preNL, postNL)$
15: **end for**
16: **for all** $i \in$ (*preIns* $\cup$ *postIns*)$\backslash S$) **do**
17: *CNF* $\leftarrow$ *CNF* $\cup$ (*domainBitOf*$(i) = 0$)
18: **end for**
19: **for** $n \in$ *diamondOf*(S_{pre}, T_{pre}) **do**
20: *res* $\leftarrow$ *callSat*(*CNF*, *domainBitOf*$(n) = 1$)
21: **if** *res* $=$ *SAT* **then**
22: $\sigma \leftarrow \sigma \cup n$
23: **end if**
24: **end for**

treated as non-deterministic inputs. The equivalence constraint is encoded only for the value bit of each input.

All input nets which are not S are encoded to be over the Boolean domain only, by setting their domain bit to be 0 (lines 16-18). This way we ensure that only S is encoded as symbolic, which guarantees that the symbolic evaluation is modeled correctly using the dual-rail encoding with its extended gate operators.

The last step of the algorithm is to check which nets in the diamond of S and T can observe the symbol of S under the path-sensitization constraints. This check, shown in lines 19-24, is performed using a Sat solver. In our implementation we used MiniSat [13]. In each iteration we ask the solver to find an assignment in which the domain bit of the current net is constrained to 1. Since S is the only symbol in the given CNF (all other inputs were constrained to be Boolean), finding a true assignment for the given CNF instance means that S is observable at this net under the path-sensitization function. In our implementation we used MiniSat incrementally by sending the CNF only once before the last for loop, and calling it each iteration with an assumption vector containing the positive literal of the domain bit of the current net. All the nets adhering to the check of Equation 1 are collected in line 22 into σ and returned by the procedure. Finally, the actual corresponding paths are constructed based on the nets in σ as was described in the end of Section III-B.

V. Results

To evaluate the effectiveness of our algorithm, we applied it on synthesis results of 21 different IBM design blocks,

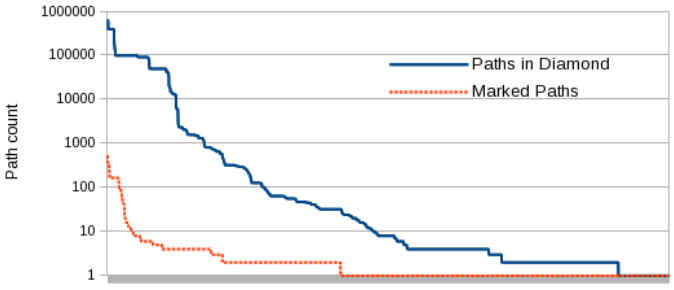

Fig. 4. Total paths in diamond vs. paths marked by our algorithm

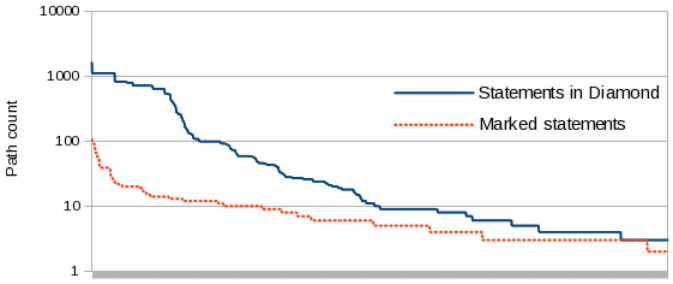

Fig. 6. Total statements in diamond vs. statements marked by our algorithm

belonging to two high-end processors. The examples were chosen to represent designs at various project levels, such that we would have both mature designs with relatively low number of timing violations and designs at their early stages that may have a higher number of longer critical paths. In order to make sure that our algorithm implementation is correct, we hand-validated the path-correspondence results of many timing problems to make sure that they point to the correct parts in the RTL. This section, however, contains quantitative validation of our algorithm. The algorithm was incorporated into an in-house synthesis flow. In this flow, the pre-synthesis netlist was saved just after the RTL elaboration, i.e a technology independent netlist. The post-synthesis netlist, technology mapped, was also used for the analysis. For each of the 21 designs, an end-point report was generated by the synthesis tool (in terms of the post-synthesis netlist), listing the worst slack paths up-to some given slack threshold. Overall our benchmark set has total of 13351 critical paths. A pre-synthesis netlist diamond is considered as trivial if it consists of a single path connecting its endpoints. Obviously there is no need to run our path correspondence algorithm on these cases. In our measurements, about a half of the diamonds were trivial. It is worth noting that about 6% out of the total paths were classified as functionally false paths.

Figure 4 shows an overall view of the path counts for all the non-trivial diamonds before and after applying our path-correspondence algorithm. The solid line shows the path count in the original pre-synthesis diamonds while the dashed line shows the number of paths marked by our algorithm for each case. Each point on the x axis represents a critical path and the actual count numbers are given in the y axis, in logarithmic scale. Note that the values of the original path counts and the number of marked paths were separately sorted, so the two curves on the diagram do not correlate, but rather demonstrate the quantitative comparison between the two parameters over the entire data set. In the vast majority of the non-trivial cases, our algorithm managed to significantly reduce the number of paths to examine: out of 3649 cases where the number of paths to examine exceed 5, 3087 cases (84%) were reduced to have only 5 or fewer paths to examine by our algorithm. In 3418 cases, our algorithm managed to mark a single path out of a non-trivial diamond.

Figure 6 shows similar quantitative comparison in terms of the source code RTL statements. From user perspective, it is convenient to show the corresponding paths in terms of a collection of RTL statements. Since the pre-synthesis netlist retain the RTL signal names and most of its structure, it is fairly straight forward to perform back-annotation from it to the RTL itself. Thus, one of our path-correspondece algorithm outputs is the RTL statements (e.g. concurrent assignments) related to the signals of the corresponding paths. From our experience, it is fairly difficult to visualize (and understand) a logic diamond that contains more than a dozen of RTL statements. Before applying the algorithm there are 3177 cases having more than 15 statements in the diamond out of which the algorithm was able to reduce 2511 to have 15 or less statements (79%).

To further understand how our algorithm behaves on different types of designs, we show, for each design, the average reduction in the number of paths and in the number of statements in Figure 5 and Figure 7, respectively. For example, designs number 2 and 3 include complex arithmetic computation logic written using low-level statements. In these cases, our algorithm was able to find very few corresponding paths in each diamond. On the other hand, designs 8-9 are data

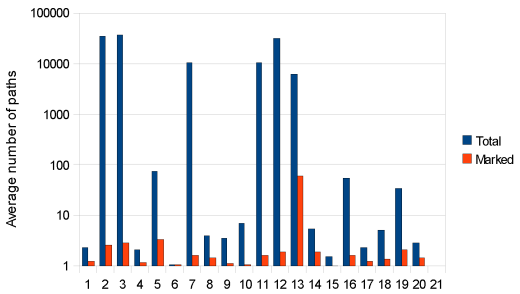

Fig. 5. Average number of paths in the logic diamond vs. number of marked paths, per design

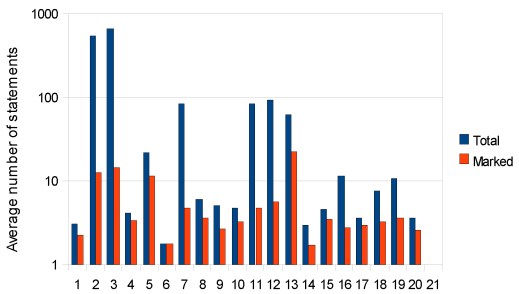

Fig. 7. Average number of RTL statements in the logic diamond vs. number of marked RTL statements, per design

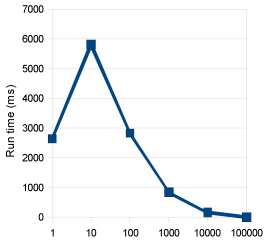

Fig. 8. Run time distribution

path logic written using higher level statements which exhibit small number of paths and statements in the pre-synthesis diamond.

Next, we examined the run time behavior of our algorithm. For our tests, we used a dedicated machine with Intel 2.33 GHz Core 2 Duo processor and 4GB of RAM, running 64 bits RedHat 5.6 Linux. Figure 8 shows the distribution of the run time, given in milliseconds, over the cases in our data set. In the vast majority of the cases, our path-correspondence algorithm took less than a second, which makes it very suitable for integration into an interactive timing debug environment. The fact the algorithm is very efficient, even on relatively large cones, i.e. on large CNF instances, is attributed to the incremental nature of the problem and the way MiniSat was used incrementally. Out of 13351 paths, only in 47 cases it took more than 5 seconds. The longest one took about 35 seconds.

Finally, in order to visually illustrate the merit of our path-correspondence algorithm, we show some non-trivial diamond in Figure 9 in form of a graph. The starting point of the path is the left square, the ending point is the right square and each internal round circle represents a concurrent RTL assignment statement. This logic diamond consists of 29 RTL statements which form 27 possible paths. Our path-correspondence algorithm found 1 corresponding path, marked in dark nodes.

VI. Conclusion and Future Work

In this paper we presented a novel approach for dealing with the problem of correlating a timing problem expressed in terms of post-synthesis netlist, to the pre-synthesis design representation (RTL). We described a Sat-based path-correspondence algorithm which is based on functional analysis of the design, thus making it robust to name mangling and structural changes a design may undergo during the synthesis process. Experimental results show that our algorithm is both efficient and effective in reducing the number of paths, and correspondingly the number of RTL statements, one would have to consider when debugging a given timing problem. The algorithm can be easily incorporated into existing synthesis flows as a post-processing step, requiring the pre-synthesis and post-synthesis netlists and primary input and latch output correspondence as its inputs.

In the future we plan to enhance even more the path reduction strength of our algorithm by adding constraints that can prune paths which do not correspond to the given critical path but are still sensitizable by the given path sensitization function. Furthermore, we plan to investigate how our algorithm can help solving other problems such as ECO localization and general back-annotation.

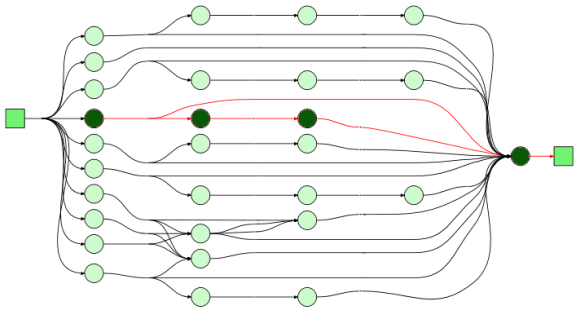

Fig. 9. Example of path-correspondence result: corresponding paths are marked in dark nodes

References

[1] R. Berridge, R. Averill, A. E. Barish, M. Bowen, P. Camporese, J. DiLullo, P. Dudley, J. Keinert, D. W. Lewis, R. D. Morel, T. Rosser, N. Schwartz, P. Shephard, H. Smith, D. Thomas, P. Restle, J. Ripley, S. Runyon, and P. Williams, "Ibm power6 microprocessor physical design and design methodology," *IBM Journal of Research and Development*, vol. 51, no. 6, pp. 685–714, 2007.

[2] R. Brayton, *Logic Minimization Algorithms for VLSI Synthesis*, ser. a Kluwer international series in engineering and computer science: VLSI, computer architecture, and digital signal processing. Springer, 1984. [Online]. Available: http://books.google.co.il/books?id=VT2kzZBPvz4C

[3] G. D. Hachtel and F. Somenzi, *Logic synthesis and verification algorithms*. Springer, 2006.

[4] A. Mishchenko and R. Brayton, "Recording synthesis history for sequential verification," in *Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design*, ser. FMCAD '08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 4:1–4:8. [Online]. Available: http://dl.acm.org/citation.cfm?id=1517424.1517428

[5] T. Schuele and K. Schneider, "Three-valued logic in bounded model checking," in *Formal Methods and Models for Co-Design, 2005. MEMOCODE '05. Proceedings. Third ACM and IEEE International Conference on*, 2005, pp. 177–186.

[6] H.-C. Chen and D.-C. Du, "Path sensitization in critical path problem," in *Computer-Aided Design, 1991. ICCAD-91. Digest of Technical Papers., 1991 IEEE International Conference on*, 1991, pp. 208–211.

[7] A. Kuehlmann and F. Krohm, "Equivalence checking using cuts and heaps," in *Design Automation Conference, 1997. Proceedings of the 34th*, 1997, pp. 263–268.

[8] Y. Matsunaga, "An efficient equivalence checker for combinational circuits," in *Design Automation Conference Proceedings 1996, 33rd*, 1996, pp. 629–634.

[9] S.-L. Huang, W.-H. Lin, P.-K. Huang, and C.-Y. Huang, "Match and replace: A functional eco engine for multierror circuit rectification," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 32, no. 3, pp. 467–478, 2013.

[10] J.-J. Liou, A. Krstic, L.-C. Wang, and K.-T. Cheng, "False-path-aware statistical timing analysis and efficient path selection for delay testing and timing validation," in *Design Automation Conference, 2002. Proceedings. 39th*, 2002, pp. 566–569.

[11] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Trans. Computers*, vol. 35, no. 8, pp. 677–691, 1986.

[12] G. S. Tseitin, "On the complexity of derivations in the propositional calculus," *Studies in Mathematics and Mathematical Logic*, vol. Part II, pp. 115–125, 1968.

[13] N. Eén and N. Sörensson, "An extensible SAT-solver," *Theory and Applications of Satisfiability Testing (SAT)*, vol. 2919, pp. 333–336, 2003. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-24605-3_37