Enabling Exact Delay Synthesis

Luca Amarú*, Mathias Soeken†, Patrick Vuillod*, Jiong Luo*, Alan Mishchenko‡, Pierre-Emmanuel Gaillardon§, Janet Olson*, Robert Brayton‡, Giovanni De Micheli†

*Synopsys Inc., Design Group, Sunnyvale, California, USA
†Integrated Systems Laboratory, EPFL, Lausanne, Switzerland
‡Department of EECS, UC Berkeley, Berkeley, California, USA
§LNIS, University of Utah, Salt Lake City, Utah, USA

***Abstract*—Given (i) a Boolean function, (ii) a set of arrival times at the inputs, and (iii) a gate library with associated delay values, the exact delay synthesis problem asks for a circuit implementation which minimizes the arrival time at the output(s). The exact delay synthesis problem, with given input arrival times, relates to computing the communication complexity of a Boolean function, which is an intractable problem. Input arrival times are variable and can take any value, thereby making the exact delay synthesis search space infinite. This paper presents theory and algorithms for exact delay synthesis. We introduce the theory of equioptimizable arrival times, which allows us to partition all arrival time patterns into a finite set of equivalence classes. Thanks to this new theory, we create for the first time exact delay circuit databases covering all Boolean functions up to 5 variables and all possible arrival time patterns. We describe further arrival time compression techniques which enable the creation of larger databases. We propose an enhanced delay synthesis flow capable of dealing with large circuits, combining exact delay logic rewriting and Boolean optimization techniques, attaining unprecedented results. We improve 9/10 of the best known results in the EPFL arithmetic delay synthesis competition, outperforming previous best results up to 3×. Embedded in a commercial EDA flow for ASICs, our exact delay synthesis techniques reduce the total negative slack by 12.17%, after physical implementation, at negligible area and runtime costs.**

I. Introduction

Logic circuits with minimum delay are of paramount interest to *Electronic Design Automation* (EDA). Delay oriented logic synthesis, or delay synthesis in short, aims at reducing the output arrival time of a logic circuit given (i) a target Boolean function, (ii) arrival times at the inputs, and (iii) available library gates. As timing requirements in modern designs are hard to meet, delay synthesis most commonly targets minimum results. Unfortunately, obtaining exact (minimum) delay results in synthesis is a difficult problem. Exact delay synthesis, with given input arrival times, relates to the intractable problem of computing the communication complexity of a Boolean function[1] [1].

Because of the intrinsic complexity of exact delay synthesis, modern design flows rely on heuristic methods such as *Sum Of Products* (SOP) balancing and *And Inverter Graph* (AIG) techniques [2], [3], high-effort collapsing and restructuring [4], Shannon's decomposition [5], BDD-based techniques [6], *Majority Inverter Graph* (MIG) algebraic and Boolean techniques [7], and many others [8]–[11]. However, heuristic techniques may miss optimization opportunities. While it is widely accepted that exact delay synthesis results cannot be obtained for large design blocks in practice, other *locally-optimal* approaches are still viable in principle. Exact delay circuits can be computed offline for practical (small) cases, stored in a database and re-used online during synthesis. Assuming the gate library technology is an input to the synthesis problem, a database for exact delay synthesis is a $k \times m$ matrix of logic circuits, where k is the number of considered Boolean functions and m is the number of arrival time patterns at the inputs. For n input variables, we have $k \leq 2^{2^n}$, but m is unbounded as the number of possible arrival time patterns is infinite.

In this paper, we present the theory of *equioptimizable arrival times*. We prove that for any Boolean function, two equioptimizable arrival time patterns share the same exact delay circuit. Equioptimizable arrival time patterns allow us to partition all infinite possible arrival times into a *finite* number of equivalence classes containing equivalent arrival time patterns. This enables the use of exact delay synthesis in practice. For example, considering 4 inputs and a representative CMOS technology, including standard NAND, NOR, XNOR, MUX and INV gates, all (infinite!) arrival times can be mapped into just 280 equioptimizable arrival time patterns (see Section IV-A). We create exact delay databases covering all Boolean functions up to 5 variables and all possible arrival time patterns. We describe further arrival time compression techniques that enable the creation of larger databases. We propose an enhanced delay synthesis flow, combining exact delay logic rewriting and Boolean optimization techniques, attaining unprecedented quality of results. Experimental results over all the 65 536 4-variable Boolean functions reveal that heuristic synthesis techniques do not hit the minimum delay for more than 40% of the cases, being up to 3 delay units off the exact result. Employed in a logic optimization flow followed by LUT-mapping, our exact delay synthesis techniques improve the best known results for 9/10 of the EPFL arithmetic benchmarks [12], with up to 3× reduction in number of logic levels. We also demonstrate a much smaller gap between our new synthesis techniques and the recently introduced exact depth benchmarks [13]. Complete ASIC design results, using a commercial design flow and 51 benchmarks, show that our synthesis techniques reduce the worst negative slack by 3.04% and the total negative slack by 12.17%, after physical implementation, at negligible area and runtime costs.

II. Background and Motivation

Exact Logic Synthesis: Synthesis algorithms to find optimum network realizations for a given metric can be divided

[1]Karchmer and Wigderson [1] observed that the *minimum* circuit depth of a Boolean function, over the basis AND/OR/NOT, can be characterized as the complexity of a corresponding communication game.

into three categories [14]: (i) algorithms based on functional decomposition (see, e.g., [15], [16]), (ii) algorithms based on network enumeration (see, e.g., [17]–[19]), and (iii) hybrid approaches that are based on both functional and structural properties (see, e.g., [14], [20], [21]). Explicit network enumeration algorithms, up to 5 input variables, and implicit network enumeration algorithms, beyond 5 input variables, are considered the most practical ones. Further information on the other algorithms is given in [14]. One of the first algorithms to find optimum circuit realizations with implicit enumeration was proposed by Muroga and Ibaraki [18]. They present a method based on integer linear programming. Although they put an emphasis on the synthesis of multi-level NOR networks, their approach is generic and can take into account several other network restrictions. Ernst has thoroughly investigated the problem of optimal multi-level logic synthesis [14]. Her algorithm is implemented using a branch-and-bound method similar to the ones proposed by Culliney et al. [21]. The algorithm allows her to handle different synthesis options and can be adapted to various cost criteria. Also, being based on branch-and-bound, the algorithm can easily be adapted to relax the optimality guarantee and find networks of near-optimal cost. Knuth has shown a SAT encoding to find size-optimum networks consisting of binary logic gates [17]. His formulation requires the input function to be represented as a truth table and he shows how symmetry breaking and the restriction to normal functions can reduce the search space, borrowing previous ideas presented for explicit network enumeration. Along these lines, Soeken *et al.* [22], [23] recently revisited SAT-based exact synthesis to find logic networks for different logic representations that are optimum in size or depth.

Motivation: It is a widely accepted fact that exact synthesis, whether targeting area, delay or power, cannot be applied to a monolithic design block in practice. Consequently, it is customary to partition the design into smaller logic cones to find exact circuits with today's computational capabilities. As running exact synthesis online can be runtime prohibitive—even for small and medium logic blocks—using a database with precomputed exact circuits is considered an affordable solution. A constant database look-up time keeps the runtime overhead small, but does not compromise quality-of-results. Such methodology is already a standard for area-oriented synthesis, in both academia and industry [24]. However, it is not known how to apply such exact synthesis approach for delay, another crucial optimization objective in modern digital circuits. In this paper, we propose solutions to major open problems, allowing us to use exact delay circuit databases in a commercial design flow, thus unlocking remarkable delay reductions.

One of the main challenges in exact delay synthesis, preventing us to build efficient circuit databases, is the infinite number of arrival time patterns. In principle, each input can take any arrival time between 0 and $+\infty$. The current solution to deal with this problem is not to consider arrival time information, explicitly, during the database construction [3]. In this scenario, as we target exact results, we need to store all possible circuit structures, for each Boolean function in the database, which are not delay-dominated by any other [25]. Note that there may be a superexponential number of non-delay-dominated circuits for a n-variable function. During synthesis, as we access this type of database, all non-delay-dominated circuits for a function need to be probed given the input arrival times, in order to retrieve an exact delay result. As a consequence, the runtime complexity of this approach is not tractable. Interestingly enough, the impediment here derives from a missing theoretical link between minimum delay circuits and input arrival times.

III. Theory of Equioptimizable Arrival Patterns

In this section, we present the theory of equioptimizable arrival time patterns. In the following, we introduce notations, definitions, and theorems for equioptimizable arrival time patterns. We assume the reader is familiar with basic concepts on logic optimization and *Binary Decision Diagrams* (BDDs, [26]). For a review on these topics, we refer the interested reader to [8].

A. Notations and Basics

In the first part of this section, we consider single output functions and integer valued delay units. We successively extend the theory to multiple output functions and rational valued delay units. We start with general definitions on exact circuits and related properties.

Definition 1 (Exact delay circuit): Let f be an n-variable Boolean function, L be a library of gates with associated delay values, and $T = (t_1, \dots, t_n)$ be input arrival times. An *exact delay circuit* is a logic circuit, composed of gates from L, that implements f with the minimum arrival time at its output. We refer to the minimum output arrival time as $\min_{\mathrm{t}}(f, L, T)$.

Next, we provide a strong notion of critical paths.

Definition 2: Let C be a (not necessarily exact) circuit that implements a Boolean function f, and let T be input arrival times. Then, an *Essential Critical Path* (ECP) of C is a path from an input x_i to C's output such that for any $\varepsilon > 0$, there exists an input assignment to C such that by replacing t_i with $t_i + \varepsilon$, the output arrival time of C increases.

Note that there may exist multiple ECPs in the same circuit. Also, note that the ECP's location strongly depends on the input arrival times and the logic structure of C. It is easy to see that the output arrival time of a circuit is the sum of t_i and the delay of the ECP, where x_i is the starting node of the ECP.

B. Bounds on Essential Critical Paths

Next, we show an upper bound on the delay of essential critical paths, which are later used for the analysis of exact delay circuits.

Theorem 1: Let L be a gate library and let $\Delta(n, L)$ be n times the best delay for a MUX operation implemented using gates in L. For every n-variable function f, there always exists a circuit such that the delay of its ECP is bounded by $\Delta(n, L)$.

Proof: We can construct f as a BDD in which the longest path visits at most n non-terminal nodes that can be realized as a MUX. ■

This result yields an upper bound on the minimum output arrival time.

Corollary 2: The minimum output arrival time $\min_{\mathrm{t}}(f, L, T)$ is bounded by $\Delta(n, L) + \max T$.

```
Input  : Library $L$, arrival time pattern $T = (t_1, \dots, t_n)$
Output : Compressed arrival time pattern
1  set $d \leftarrow \max T - \Delta(n, L)$;
2  foreach $t_i \in T$ do
3      if $t_i < d$ then
4          set $t_i \leftarrow d$;
5  end
6  set $m \leftarrow \min T$;
7  foreach $t_i \in T$ do
8      set $t_i \leftarrow t_i - m$;
9  end
10 return $T$;
```

Algorithm 1: Arrival time lossless compression algorithm

C. Arrival Time Pattern Compression

With the help of the upper bounds on the output arrival time, we can compress arrival time patterns. We do so by limiting individual input arrival times to a fixed range, which results in a partitioning of all arrival time patterns into a finite number of classes.

Theorem 3: Let $T = (t_1, \dots, t_n)$ be an input arrival time pattern such that there exists a t_i with $t_i < \max T - \Delta(n, L)$. Let $T' = (t_1, \dots, t'_i = \max T - \Delta(n, L), \dots, t_n)$. Then $\min_{\mathrm{t}}(f, L, T) = \min_{\mathrm{t}}(f, L, T')$ for every n-variable function f.

Proof: (*by contradiction*) Let $t = \min_{\mathrm{t}}(f, L, T)$ and let $t' = \min_{\mathrm{t}}(f, L, T')$. We assume that $t \neq t'$. Note that, since $t'_i > t_i$, we also have $t' > t$. We need to consider two cases depending on whether t'_i is originating in an ECP. If it is not, then t'_i does not affect t' and the contradiction is evident. If t'_i is originating in an ECP, then we know that $t' = d_{\mathrm{ECP}} + t'_i$, where d_{ECP} is the delay of the ECP. Therefore, $t' = d_{\mathrm{ECP}} + \max T - \Delta(L, n)$. Using Theorem 1, we have that $t' \leq \Delta(n, L) + \max T - \Delta(n, L) = \max T$. Consequently, we have $t < t' \leq \max T$, which means that the output arrival time is smaller than the input arrival time, leading to a contradiction. ■

As a consequence of not changing the output arrival time in Theorem 3, T' also leads to the same exact delay circuit as T.

Corollary 4: An exact delay circuit for (f, L, T) is also an exact delay circuit for (f, L, T'), under the conditions of Theorem 3.

We use Theorem 3 to define an arrival time compression algorithm, illustrated in Algorithm 1. The driving principle is to reduce the maximum difference between the latest and earliest arrival times to $\Delta(n, L)$.

Note that Algorithm 1 can take an infinite number of possible arrival time patterns as input. On the other hand, the number of distinct patterns achievable as output of Algorithm 1, with unit-discretized delay information, is $(\Delta(n, L) + 1)^n$. On top of the infinite to finite space reduction, it is left to show that Algorithm 1 preserves the exactness property for delay synthesis.

Theorem 5: An exact delay circuit for (f, L, T) is also an exact delay circuit for (f, L, T'), where T' is obtained by running Algorithm 1 on L and T.

Proof: We need to prove that operations in Algorithm 1 do not alter the optimality of the original exact delay circuit. In Algorithm 1, there are two operations on t_i: (i) $t_i \leftarrow d$ and (ii) $t_i \leftarrow t_i - m$. Setting $t_i \leftarrow d = \max T - \Delta(n, L)$, if $t_i < d$, does not change the original exact delay circuit according to Theorem 3. Subtracting a common offset $m = \min T$ from all input arrival times t_i does not affect the circuit structure. ■

Consequently, for a given library L, both arrival time patterns T and T', where T' is obtained from Algorithm 1, are equioptimizable patterns. The exact delay optimization process therefore finds the same circuit structures for both patterns.

D. Generalizations

1) Multiple Output Functions: The theory of equioptimizable patterns is also valid for multiple outputs. To notice this, it is sufficient to consider the latest arriving output among all outputs of a circuit. All other arguments follow when treating this output as the single output in previous considerations. The proof to Theorem 1 will use shared BDDs, but since the number of variables does not change, also the path length in the BDD stays the same.

2) Rational Valued Delays: Real physical delays take values in $\mathbb{R}_{\geq 0}$, because of the continuous nature of time. However, technology libraries have delay values defined on a minimum delay precision, so they take values in $\mathbb{Q}_{\geq 0}$.

We now consider $T = (t_1, \dots, t_n) \in \mathbb{Q}^n_{\geq 0}$. We also consider library gates with delay values in $\mathbb{Q}_{\geq 0}$. We can transform any exact synthesis problem defined in $\mathbb{Q}_{\geq 0}$ into an equivalent problem defined in $\mathbb{N}_{\geq 0}$ by normalizing all values using an appropriate factor. One obtains integer values by dividing all delays for library gates and all arrival times by $d_{\mathrm{p}}(L)$, which is the minimum delay precision of library L.

It is worth noticing that our study is based on a load-independent abstraction of the delay information. This is because, during logic optimization and mapping, we do not have accurate information on the physical fanouts of a given gate. Indeed, this information becomes available only during physical design. Thus, we defer the use of load-dependent delays at later stages in the design flow, e.g., placement-aware sizing and buffering, where the real fanouts and their relative distance is known.

Please note that the primary purpose of Section III-D is to show that, in theory, we can deal with physical libraries and achieve exact results still having finite countable sets. In practice, using a scale factor of $1/d_p(L)$ leads to delay sets too large to be stored and processed. Lossy quantization is a commonly accepted solution leading to tractable delay sets.

IV. Building Exact Delay Databases

The previous section showed how to obtain a finite set of equioptimizable arrival time patterns. This enables the creation of databases of exact delay circuits. One of the remaining challenges in building, and storing, such databases is to keep the memory footprint small. As both the number of functions and number of equioptimizable patterns grow superexponentially with the number of inputs n, various compression techniques are needed to make a database scalable. On the number of functions, an immediate size saving derives from storing only input permutation classes. This does not compromise the results exactness after retrieval, as it will be shown later on in this section. On the number of equioptimizable patterns,

```
Input  : n-variable Boolean function $f$, library $L$, pattern $T$
Output : Exact delay circuit $C$ from $\mathrm{edd}(n, L) = (C_{ij})$
1 set $D \leftarrow \Delta(n, L)$;
2 set $\hat{f}, \pi \leftarrow$ classify($f$);
3 set $i \leftarrow$ index($\hat{f}$);
4 apply Algorithm 1 to $L$ and $T$;
5 set $T \leftarrow (t_{1\pi}, \ldots, t_{n\pi})$;
6 set $j \leftarrow \sum_{i=1}^{n} t_i \cdot (D+1)^{i-1}$;
7 set $C \leftarrow C_{ij}$;
8 permute $C$'s inputs according to $\pi$;
9 return $C$;
```

Algorithm 2: Exact circuit retrieval from the EDD

sharing of equivalent patterns, still leading to the same circuit structures, further helps reducing the size of a database.

Formally, an *Exact Delay Database* (EDD), for gate library L and n variables, is a matrix of logic circuits:

$$\mathrm{edd}(n, L) = \begin{bmatrix} C_{11} & C_{12} & \cdots & C_{1M} \\ C_{21} & C_{22} & \cdots & C_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ C_{N1} & C_{22} & \cdots & C_{NM} \end{bmatrix}$$

having N rows, where N is the number of *permutation* classes of n-input Boolean functions, and M columns, where M is the number of equioptimizable patterns over n variables and library L. Each entry C_{ij} of the EDD is an exact delay circuit for (i) the Boolean function representative of the i^{th} P class and (ii) the j^{th} indexed equioptimizable arrival pattern.

While there are known techniques to enumerate and index permutation classes [17], for quick look-up time purposes in the database, the same solutions are not directly applicable for equioptimizable patterns. To address this issue, we develop unique encoding algorithms to address equioptimizable patterns.

A. Encoding Equioptimizable Patterns

Equioptimizable arrival patterns can be encoded into an interval of integers starting at 0, using a simple explicit equation. The insight is the following. Every arrival time in the equioptimizable pattern can take integer values from 0 to $D = \Delta(n, L)$. We have n arrival times, with distinct and non-switchable positions. We interpret such arrival time pattern as an integer in base $(D + 1)$, having n digits. Each digit is associated with a specific arrival time in the pattern. Therefore, given an arrival time pattern $T = (t_1, \ldots, t_n)$, one gets a unique integer in the range $[0, (D + 1)^n - 1]$ by:

$$\sum_{i=1}^{n} t_i \cdot (D+1)^{i-1} \tag{1}$$

Using Eq. 1, one can compute a column index for an EDD.

B. Exact Delay Circuit Retrieval

Algorithm 2 shows a procedure to retrieve an exact delay circuit from the EDD. We describe Algorithm 2 by means of an example.

Example 1: Let us consider an $\mathrm{edd}(3, L)$, with gate library $L = \{\mathrm{NAND}, \mathrm{NOR}, \mathrm{XNOR}, \mathrm{MUX}, \mathrm{INV}\}$. The timing information is $t_{\mathrm{NAND}} = t_{\mathrm{NOR}} = t_{\mathrm{INV}} = 1$ and $t_{\mathrm{XNOR}} = t_{\mathrm{MUX}} = 2$. The $\mathrm{edd}(3, L)$ has 80 rows and 343 columns. Each row points to a permutation class, and we have 80 different permutation classes of 3 variables [27], [28]. Each column points to an equioptimizable pattern, and we have 343 equioptimizable patterns because $D = \Delta(3, L) = t_{\mathrm{MUX}} \cdot 3 = 6$, and the number of patterns is $(D + 1)^n = (6 + 1)^3 = 343$. Please note that the number of equioptimizable patterns can be further reduced as it will be discussed later on in this paper. In this example, we want to retrieve from $\mathrm{edd}(3, L)$ an exact delay circuit for function $f = a \oplus b + \bar{c}$ and arrival times $t_a = 12$, $t_b = 0$ and $t_c = 1$. With Algorithm 2, we first find the permutation class representative $g = \bar{x}_1 + x_2 \oplus x_3$. The permutation is $\pi = (2, 3, 1)$. The index i of permutation class g is 16, but this number can be arbitrarily changed as rows in the EDD can be swapped with no effect on the results. We then apply Algorithm 1 to $T = (12, 0, 1)$, obtaining $T = (6, 0, 0)$. We apply permutation π to T, obtaining $T = (0, 6, 0)$. The index of the equioptimizable pattern is computed as $j = 0 + 6 \cdot 7 + 0 \cdot 7^2 = 42$. At this point, we look for entry $C_{16,42}$ in the EDD. The circuit contained in $C_{16,42}$ is $C = \mathrm{MUX}(x_2, \mathrm{NAND}(x_1, \mathrm{INV}(x_3)), \mathrm{NAND}(x_1, x_3))$. After applying π to C's inputs, we obtain the exact delay circuit $C = \mathrm{MUX}(a, \mathrm{NAND}(c, \mathrm{INV}(b)), \mathrm{NAND}(c, b))$. The returned circuit achieves $t_f = 14$ which is the minimum output arrival time possible given the function and input arrival times.

C. Computing The Entries of An Exact Delay Database

Computing an entry of $\mathrm{edd}(n, L)$ involves finding an exact synthesis solution for a given Boolean function and arrival time pattern. This problem can be practically solved with explicit circuit enumeration for $n \leq 5$. For larger Boolean functions, e.g., $n \geq 6$, implicit circuit enumeration techniques based on Boolean satisfiability (SAT) are more practical. We discuss both approaches hereafter.

Please note that the exact delay synthesis problem is an intractable problem [1], [17]. Exact delay solutions are increasingly hard to compute with growing n, and may require the database creation to happen offline with respect to the rest of the synthesis flow. Please also note that, with $n \geq 6$, only partial EDDs are currently feasible.

1) Explicit Circuit Enumeration: Explicit circuit enumeration techniques explore the logic representation space exhaustively. This is convenient when a complete exact delay database needs to be populated. Algorithm 3 depicts the high-level flow for explicit circuit enumeration for exact delay synthesis. It operates as follows. We first store trivial circuits for the logic constants and input variables. These circuits, which are simple wires, are delay optimal by construction. Then, we start the enumeration loop and we try to add a new gate from L, in increasing delay order, having as fanin some of the already stored functions, also in increasing arrival time order. If the generated function is not already stored, we save it. Otherwise, we already have a better delay implementation stored for the generated function. We keep iterating this procedure until we have stored circuits for all the 2^{2^n} functions. It can be easily seen that this procedure only stores exact delay circuits. Please note that the procedure in Algorithm 3 can be sped up by taking into account library considerations and function filtering. On the library side,

```
Input  : Arrival time pattern $T = (t_1, \ldots, t_n)$, gate library $L$
Output : Exact delay circuits for all $2^{2^n}$ Boolean functions
store circuits for constant (0, 1) and projection functions $x_i$;
set $t \leftarrow n + 2$;
sort $L$ by increasing delay;
while $t < 2^{2^n}$ do
    foreach $g \in L$ do
        set $m \leftarrow$ fanin count of $g$;
        foreach set of pins $(o_1, \ldots, o_m) \in$ stored circuits do
            foreach permutation $\pi$ of $(o_1, \ldots, o_m)$ do
                set $f \leftarrow g(o_{1\pi}, \ldots, o_{m\pi})$;
                if $f$ is not already stored then
                    store circuit for $f$;
                    set $t \leftarrow t + 1$;
                end
            end
        end
    end
end
```

Algorithm 3: Explicit enumeration of exact delay circuits

we can filter based on the gate properties, e.g., functional symmetry, delay dominance and decomposition, etc. On the function side, we can filter based on considerations on NPN-class properties [14] of the already stored functions. With all filtering, explicit enumeration runs quite fast. It takes less than *2 minutes* to generate $\text{edd}(4, L)$ for a typical L in CMOS technology, such as the one described in Example 1. On the other hand, SAT-based methods can take more than *3 hours* to complete a 4 variables delay-optimal database [22]. As previously mentioned, the advantage of explicit enumeration over SAT quickly vanishes with $n > 5$.

We refer the interested reader to [14], [17] for an in-depth discussion on circuit enumeration techniques.

Fig. 1 shows a sample entry for an exact delay database, generated by the aforementioned explicit circuit enumeration algorithm. The minimum delay precision is set to 0.5 delay units for this example, for practical reasons. The gate delays are extracted from a characteristic CMOS library in a 45nm technology node.

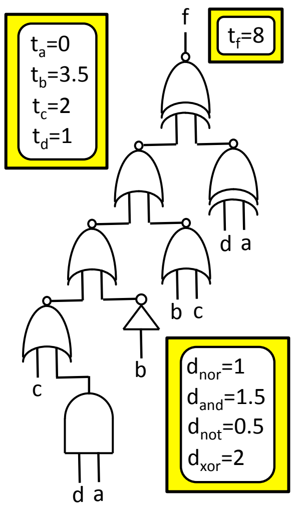

Fig. 1. Sample entry of an exact delay database

```
Input  : Function $f$, arrival time pattern $T = (t_1, \ldots, t_n)$,
         library $L$
Output : Exact delay circuit for $f$
set $t \leftarrow \min T$;
while true do
    foreach gate size $r$ do
        set $C \leftarrow \text{hasCircuit}(f, L, r, T, t)$;
        if $C$ is a circuit then return $C$;
    end
    set $t \leftarrow t + d_{\text{p}}(L)$;
end
```

Algorithm 4: SMT-based exact delay synthesis

2) Boolean Satisfiability: We can solve the task of exact delay synthesis by encoding the problem as in instance of the *Boolean satisfiability* problem. For this purpose, one encodes the problem "Does there exists a circuit with r gates from L that implements function f with input arrival times T respecting an output delay of t?" Let us refer to a SAT encoding of this problem as $\text{hasCircuit}(f, L, r, T, t)$. This procedure either yields to a circuit satisfying the constraints if it exists, or *unsatisfiable* otherwise.

We can solve the task of exact delay synthesis by encoding the problem as an instance of the *Satisfiability Modulo Theories* (SMT) problem. Algorithm 4 depicts the high-level SMT procedure for exact delay synthesis. We start by setting t to $\min T$, the minimum physical output arrival time possible. We try to find a circuit with r gates by choosing suitable values for r. For example, we can increase r until an upper bound of gates is reached. If no satisfying solution and therefore no circuit can be found, t is incremented by the library delay precision $d_{\text{p}}(L)$. This procedure is guaranteed to hit the exact delay circuit eventually. We refer the reader to [22] for an in-depth discussion of SAT formulation for exact synthesis and [23] for a detailed discussion on how to encode delay constraints.

D. Reducing The Size of Exact Delay Databases

The size of an exact delay database grows very quickly with the number of inputs n, and the complexity of the gate library L. We can reduce the size of an EDD using some of the following techniques:

1) Tighten $\Delta(n, L)$: The original bound on $\Delta(n, L)$ is quite loose. For small cases, i.e., $n < 5$, it is possible to compute a much tighter bound with brute force methods, e.g., exhaustive exploration of the search space. A tighter bound leads to fewer equioptimizable patterns to store.

2) Merge Equioptimizable Patterns: Even with a tight bound on $\Delta(n, L)$, there may exist equioptimizable patterns still leading to the same exact synthesis solution. Indeed, the equioptimizable patterns theory is agnostic of the specific function considered. A post processing pass on an EDD can identify pattern candidates for merging.

3) Disregard Inverters: While inverters carry delay information, there are applications where they can be neglected as first approximation. This is the case, for example, for LUT synthesis or other technology independent optimization flows. With this assumption, the EDD can consider NPN classification instead of P classification. This leads to much fewer classes and therefore to fewer rows in the EDD.

4) Partial Databases: In practical synthesis problems, only a fraction of all possible functions over n variables are encountered. The larger n is, the smaller the fraction of encountered functions becomes. Exact delay databases can be optimized to contain only such frequently appearing function classes, and their equioptimizable patterns. This leads to a better control on the EDD size.

V. Logic Synthesis with Exact Delay Databases

Exact delay databases find natural application in logic synthesis flows. Logic rewriting techniques [24] directly support the use of an EDD. The rationale behind logic rewriting is to partition the network into smaller blocks, for example with the help of cut computation, and try to rewrite each block in topological order using precomputed solutions from a database [24]. In this context, exact delay databases enable delay-oriented logic rewriting. Please note that remapping is not necessary after such synthesis flow, as the solutions retrieved from the EDD already consists of library gates.

In its simplest implementation, logic rewriting with $\mathrm{edd}(n, L)$ requires cut computation of size n. However, we can achieve even stronger results by combining exact delay solutions of size n with Boolean delay optimization of higher order $m > n$. In this case, we compute larger cuts of size m and process each logic block as follows. We consider the latest arriving variable, say x_j, and try the following operations on the cut Boolean function, in order: (i) 2-operand disjoint support decomposition wrt. x_j and (ii) Shannon decomposition wrt. x_j. If the first operation is successful, we are left with one logic block of $(m-1)$ variables. Otherwise, Shannon decomposition creates two logic blocks of $(m-1)$ variables. Both DSD decomposition and Shannon decomposition are immediately implemented in terms of library gates. Blocks created by Shannon decomposition can be simplified by constant propagation and gate sharing. If the current blocks have size equal to n, we can directly retrieve an exact delay implementation for them from the EDD. Otherwise, we continue the delay-driven decomposition.

The goal of this combined *Boolean-exact* delay optimization is to decompose a large logic block into a delay-optimal tree where the leaves are minimum solutions from the EDD.

We call such synthesis flow *Enhanced Delay Synthesis* (EDS) with parameters m and n: $\mathrm{eds}(m, n)$. The setup $\mathrm{eds}(n, n)$ corresponds to plain logic rewriting with an EDD. More aggressive setups are $\mathrm{eds}(n+k, n)$ with $k = 1, 2, 3, 4, 5, 6$. While it is in principle possible to explore even more aggressive delay setups, the area overhead would start to be too large and possibly absorb the achieved delay gain during physical design.

VI. Experimental Results

In this section, we evaluate the efficacy of exact delay synthesis. We first run experiments for all 4 variables functions and show the gap between exact delay synthesis and state-of-the-art synthesis heuristics. Then, we employ the EDS flow, empowered with exact delay databases, to improve the best known results of the EPFL benchmark suite [12]. We also show a much smaller gap with respect to the recently introduced exact depth multi-level benchmark [13]. Finally, we integrate the EDS flow in a complete industrial design flow and show sensible timing gain measured after place & route.

A. Methodology

We implemented exact delay synthesis as part of a commercial design automation solution. Generation of exact delay databases can be performed offline, but also online. In practice, we embed by default a generic $\mathrm{edd}_{\mathrm{g}}(4, L)$, with a simple gate library consisting of NAND (delay 1), NOR (delay 1), XNOR (delay 2), MUX (delay 2), and INV (delay 1) gates. Note that the number of equioptimizable patterns, after compression, is only 280, leading to a memory footprint below 8 MB. Then, during the initial steps of the design compilation, we refine this EDD based on the provided library. In our x86-64, 16 cores, 2.6 GHz, 20GB RAM machine environment, generation of library specific $\mathrm{edd}_{\mathrm{s}}(4, L)$ takes less than 2 minutes. Generation of a complete EDD for more than 4 variables must be carried over offline, for runtime reasons.

We also generated a complete, generic $\mathrm{edd}_{\mathrm{g}}(5, L)$, but its size exceeds 600 GB. The reason is that we have $\approx 36 \cdot 10^6$ P classes of 5 variables [28] and about 1000 equioptimizable patterns, after high-effort compression. To make this database practical, compromises on the delay-exactness, e.g., use of NPN classes, or database-completeness, e.g., store only frequently appearing functions, need to be made. In our experiments, we were able to find a 400 MB quasi-exact delay database of 5 variables still leading to promising results. However, the size of the database still represents a challenge, which require loading the database in form of an external library. In commercial applications, the use of large external logic libraries, which exceed the size of the tool executable, may be hard to adopt. The best compromise we found is to have a complete $\mathrm{edd}_{\mathrm{g}}(4, L)$, statically embedded in the tool and refined dynamically during execution, and attempt online at partial EDD of higher order.

We employ the EDD within an EDS flow, as described in the previous section. We use $\mathrm{eds}(n, n)$ as core timing optimization engine, with calls to $\mathrm{eds}(n+k, n)$ to escape local minima. More details on the configuration is given below for each experiment.

B. Measuring the Exact-Heuristic Delay Gap

In this first experiment, we aim at showing the gap between exact delay results and heuristic delay techniques, already for 4 variables Boolean functions. We employ the generic gate library described in the previous subsection. We consider hard to synthesize input arrival time patterns, which are the ones challenging modern synthesis tools. In particular, with the arrival time pattern $(0, 0, 4, 4)$, traditional tools face difficulties in improving the input circuit structures for delay. We consider a required output arrival time of 0 for all 65 536 functions of 4 variables. On the one hand, we generate circuits using exact delay synthesis. On the other hand, we also generate circuits using a commercial delay-oriented heuristic approach, starting from a minimized SOP. Fig. 2 depicts the results graphically. The delay scale is normalized wrt. to unit INV delay. It is worth noticing that the synthesis heuristic does not hit the exact results more than 42% of the cases and can be up to 3 delay units off.

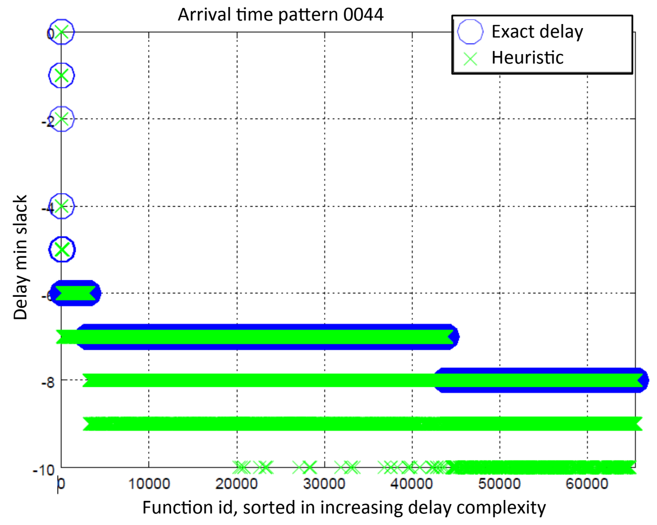

Fig. 2. Exact delay synthesis *vs.* heuristic delay synthesis. Results span all 65536 functions of 4-variables. The output required time is 0 for all cases.

On top of this experiment, we also tried to compare exact delay results to ABC [29] delay command "*if -g*", for the arrival time pattern $(0, 0, 0, 0)$. Also for this "easy" input arrival pattern, ABC results do not hit the exact numbers more than 18% of the cases and can be up to 2 delay units off.

C. Improving the Best Known Results for the EPFL Suite

The EPFL benchmark suite project keeps track of the best synthesis results, mapped into LUT-6, generated by EDA research groups [12]. In this work, we challenge the delay category of the EPFL suite, whose previous best results are held by UC Berkeley, EPFL, and Cornell research groups. We focus on the 10 arithmetic benchmarks of the EPFL suite, which are considered the hardest to synthesize.

We optimize these benchmarks using the EDS flow, with the most aggressive timing setup possible, i.e., using 0 as output(s) required time. We first use $\mathrm{eds}(4, 4)$, fed with $\mathrm{edd_g}(4, L)$, as long as we see delay improvement. Then, we try to escape local minima with $\mathrm{eds}(4 + k, 4)$, starting from $k = 1$ and increasing k up to 6 until a delay improvement is obtained. At this point, we go back to $\mathrm{eds}(4, 4)$. We iterate this routine until a delay improvement is possible or a runtime limit of 3 hours is reached. We did not impose any area constraints, as the EPFL competition for delay do not require so. For this reason, consecutive calls to $\mathrm{eds}(4 + k, 4)$, which involve continuous logic collapsing, may sensibly increase area. Nevertheless, we included in our flow a delay-bounded area-recovery phase.

We map the optimized circuits using an in-house delay-oriented LUT-6 mapper. Please note that we did not include optimization techniques [31], [32] directly in the LUT mapper, so further improvements may be reached on top of our results. We leave this to future work on improving LUT mapping.

Table I shows the results. We were able to improve 9/10 of the previous best results, in the arithmetic section. We either got (i) both fewer number of levels and fewer number of nodes (2/10), (ii) fewer number of levels but larger number of nodes (3/10) or (iii) the same number of levels but fewer number of

TABLE I
NEW BEST DELAY RESULTS FOR THE EPFL SUITE

Benchmark	I/O	LUT-6 count	Level Count	Improv.
adder	256/129	**470**	5	**yes**
barrel shifter	135/128	512	4	**no**
divisor	128/128	**26914**	**228**	**yes**
hypotenuse	256/128	151442	**567**	**yes**
log2	31/32	**9210**	55	**yes**
max	512/130	**880**	10	**yes**
multiplier	128/128	**7274**	**27**	**yes**
trigonometric sin	24/25	487941	**12**	**yes**
square-root	128/64	27265	**241**	**yes**
square	64/128	**3967**	11	**yes**

nodes (4/10)[2]. It is worth noticing that we reduced the depth of all benchmarks with > 100 levels. Also, we obtained about $3\times$ depth improvement over the previous best trigonometric sin implementation. The only benchmark which we have not improved is the barrel shifter. Please note that no research group has been able to improve the original depth nor the size of this benchmark. Thus, it is believed that this benchmark is already optimal, even though a formal proof is still missing. Our circuit implementations can be downloaded at [33].

D. Exact Depth Benchmarks Experiments

On top of challenging general benchmark suites, we also consider the recently introduced exact depth benchmarks [13]. We compare to the synthesis setup proposed in [13], using the EDS optimization flow described in the previous subsection. In place of LUT-6 mapping, we simply strash the optimized gate-level circuits into an AIG, for the sake of fair comparison.

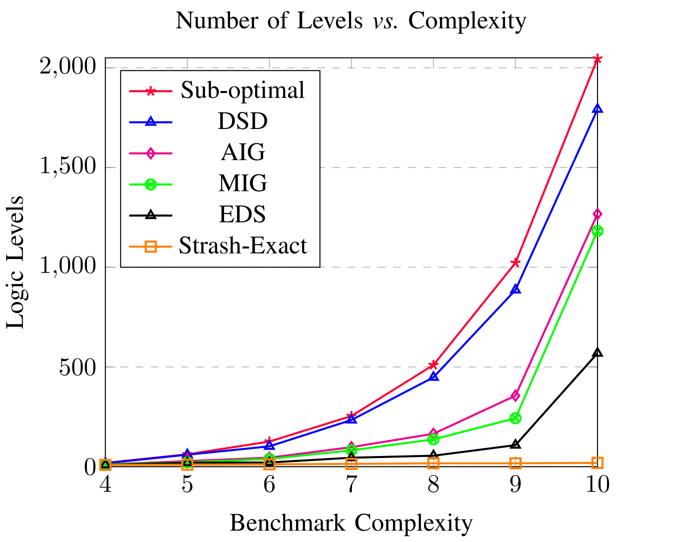

Fig. 3. Synthesis experiments over exact depth benchmarks. The *benchmark complexity* metric is computed as $log_2(\#inputs)$.

Fig. 3 shows the EDS results w.r.t. the exact depth bechmarks experiments presented in [13]. As we can notice from the plot, the gap between the orange line (Strash-exact) and the black line (EDS) is remarkably smaller than the best optimization results previously seen [13]. Indeed, the depth

[2]The tie breaking rules for the EPFL best results have been decided by the IWLS community.

is about $2\times$ smaller on average as compared to AIG/MIG results. Our results can be downloaded at [33]. Please note that, despite strongly improving previous results, an exponential gap still appears between the known minimum depth and the EDS flow. This is because the EDS flow obtains exact results on subnetworks and not on the original monolithic logic block.

E. Complete Design Experiments

In order to evaluate the potential of exact delay synthesis in commercial EDA, we integrated the EDS flow in a complete design solution, down to physical design. According to the latest trends for synthesis R&D in industry [30], we aim at sensible delay reductions, greater than 10% *Total Negative Slack* (TNS) reduction, coming at negligible area and runtime overhead, i.e., less than 1%. In order to achieve this challenging goal, we tuned the EDS flow described in the previous subsections to keep the area overhead under tight control. Runtime of EDS synthesis is intrinsically very small, because the database solutions $\mathrm{edd_g}(4, L)$ are precomputed offline and cut computation followed by replacement is runtime friendly. Table II shows the complete design results, post place & route, for 51 state-of-the-art ASICs, coming from major electronics industries. We cannot provide details on each ASIC benchmark because of non-disclosure agreements. However, we present average results w.r.t. a baseline flow without our EDS techniques. The results are summarized in Table II. Our complete design flow, embedding exact delay synthesis, enables sensible worst negative slack reduction, -3.04% on average, and strong total negative slack reduction, -12.17% on average, at only $+0.41\%$ area cost and $+0.98\%$ runtime overhead.

TABLE II
POST PLACE&ROUTE RESULTS ON 51 INDUSTRIAL DESIGN

Flow	WNS	TNS	Area	Runtime
baseline	1	1	1	1
exact delay flow	**-3.04%**	**-12.17%**	+0.41%	+0.98%

VII. CONCLUSIONS

In this paper, we presented theory and practical methods for exact delay synthesis. We introduced, for the first time, the concept of equioptimizable arrival time patterns, which enabled the creation of exact delay databases. When employed in an enhanced synthesis flow, our exact delay synthesis techniques showed excellent results. Nine out of ten best known results in the EPFL arithmetic synthesis competition have been improved, exceeding previous best results up to $3\times$. We also showed a much smaller gap with respect to the recently introduced exact depth multi-level benchmark. Embedded in a commercial design automation flow for ASICs, the exact delay synthesis techniques reduced the total negative slack by 12.17%, after physical implementation, at negligible area and runtime costs.

ACKNOWLEDGMENTS

This work was partly supported by H2020-ERC-2014-ADG 669354 CyberCare and by Grant 2016016 from the United States-Israel Binational Science Foundation.

REFERENCES

[1] M. Karchmer, A. Wigderson, *"Monotone circuits for connectivity require super-logarithmic depth"*, SIAM J. Discrete Math., 3(2):255-265, 1990.
[2] A. Mishchenko, et al. *"Delay optimization using SOP balancing"*, Proc. ICCAD, 2011.
[3] W. Yang, L. Wang, A. Mishchenko, *Lazy mans logic synthesis,* ICCAD, 2012, pp. 597604.
[4] R.K. Brayton, G.D. Hachtel, A.L. Sangiovanni-Vincentelli, *"Multilevel logic synthesis"*, Proc. IEEE 78.2 (1990): 264-300.
[5] C. L. Berman, D. J. Hathaway, A. S. LaPaugh, L. Trevillyan. *"Efficient techniques for timing correction"*, ISCAS, pp. 415419, 1990.
[6] N. Vemuri, *et al.*, *"BDD-based logic synthesis for LUT-based FPGAs"*, ACM TODAES 7.4 (2002): 501-525.
[7] L. Amaru, P.-E. Gaillardon, G. De Micheli, *"Majority-inverter graph: A new paradigm for logic optimization"*, IEEE TCAD-IC 35.5 (2016): 806-819.
[8] G. De Micheli, *"Synthesis and Optimization of Digital Circuits"*, McGraw-Hill, New York, 1994.
[9] V. Bertacco and M. Damiani, *"The disjunctive decomposition of logic functions"*, ICCAD 1997.
[10] V. N. Kravets and P. Kudva, *"Implicit enumeration of structural changes in circuit optimization"*, DAC 2004.
[11] M. Elbayoumi, M. Choudhury, V. N. Kravets, A. Sullivan, M. S. Hsiao, and M. Y. ElNainay, *TACUE: A timing-aware cuts enumeration algorithm for parallel synthesis*, DAC 2014.
[12] L. Amaru, P.-E. Gaillardon, G. De Micheli, *"The EPFL Combinational Benchmark Suite"* International Workshop on Logic & Synthesis (IWLS), 2015.
[13] L. Amaru, et al., *"Multi-level logic benchmarks: An exactness study"*, ASPDAC'17, Tokyo, Japan, January 2017.
[14] E. A. Ernst, *Optimal combinational multi-level logic synthesis,* PhD thesis, The University of Michigan, 2009.
[15] R. M. Karp, F. E. McFarlin, J. P. Roth, J. R. Wilts, *A computer program for the synthesis of combinational switching circuits,* in Symp. on Switching Circuit Theory and Logical Design, 1961, pp. 182194.
[16] J. P. Roth, R. M. Karp, *Minimization over Boolean graphs,* IBM Journal of Research and Development, vol. 6, no. 2, pp. 227238, 1962.
[17] D. Knuth, *"The Art of Computer Programming"*, Volume 4A, Part 1
[18] S. Muroga, T. Ibaraki, *Design of optimal switching networks by integer programming,* IEEE Trans. on Computers, vol. 21, no. 6, pp. 573582, 1972.
[19] A. Kojevnikov, A. S. Kulikov, G. Yaroslavtsev, *Finding efficient circuits using SAT-solvers,* in Intl Conf. on Theory and Applications of Satisfiability Testing, 2009, pp. 3244.
[20] E. S. Davidson, *An algorithm for NAND decomposition under network constraints,* IEEE Trans. on Computers, vol. 18, no. 12, pp. 10981109, 1969.
[21] J. N. Culliney, M. H. Young, T. Nakagawa, S. Muroga, *Results of the synthesis of optimal networks of AND and OR gates for four-variable switching functions,* IEEE Trans. on Computers, vol. 28, no. 1, pp. 7685, 1979.
[22] M. Soeken, L. Amaru, P.-E. Gaillardon, G. De Micheli, *"Exact Synthesis of Majority-Inverter Graphs and Its Applications"*, IEEE TCAD-IC, 2017.
[23] M. Soeken, G. De Micheli, A. Mishchenko, "Busy Mans Synthesis: Combinational Delay Optimization With SAT," Design, Automation & Test in Europe Conference (DATE), Lausanne, Switzerland, 2017.
[24] A. Mishchenko, S. Chatterjee, R. Brayton, *DAG-aware AIG rewriting a fresh look at combinational logic synthesis*, Proc. DAC 2006.
[25] S. Hassoun, T. Sasao. *Logic Synthesis and Verification*, Springer, 2001
[26] Randal E. Bryant. *"Graph-Based Algorithms for Boolean Function Manipulation"*. IEEE Transactions on Computers, C-35(8):677691, 1986.
[27] V. Correia, A. Reis. *"Classifying n-input Boolean functions"*, Proc. IWLS 2001.
[28] M. Harrison, *"The number of equivalence classes of Boolean functions under groups containing negation"*, IEEE Transactions on Electronic Computers 5 (1963): 559-561.
[29] ABC synthesis tool: https://bitbucket.org/alanmi/abc.
[30] L. Amaru, P. Vuillod, J. Luo, J. Olson, *"Logic Optimization and Synthesis: Trends and Directions in Industry"*, Design, Automation & Test in Europe Conference (DATE), Lausanne, Switzerland, 2017.
[31] B. Schmitt, A. Mishchenko, R. Brayton, *"SAT-based area recovery in technology mapping"*, IWLS'17.
[32] A. Mishchenko, R. Brayton, A. Petkovska, M. Soeken, *"SAT-based optimization with don't-cares revisited"*, IWLS'17.
[33] http://lsi.epfl.ch/benchmarks