

Timing-driven optimization using lookahead logic circuits

Mihir Choudhury and Kartik Mohanram

Department of Electrical and Computer Engineering, Rice University, Houston

{mihir, kmram}@rice.edu

Abstract

This paper describes a timing-driven optimization technique for the synthesis of multi-level logic circuits. Motivated by the parallel prefix problem, the proposed timing-driven optimization produces logic circuits with “lookahead” properties due to the inherent parallelism among the synthesized sub-circuits. Lookahead logic circuits are synthesized using global critical path sensitization information to decompose and reduce the Boolean functions of the nodes in the technology-independent representation of the logic circuit. Unlike prior timing-driven optimization techniques, where synthesis of the decomposition functions is potentially expensive, the proposed technique has the advantage that the decomposition functions are discovered in the synthesized form. On average, the proposed technique reduces the number of logic levels (mapped delay) of 15 benchmark circuits by 40%, 56%, and 22% (21%, 56% and 10%) over the best results of SIS, ABC, and an industry-standard synthesizer, respectively.

Categories and Subject Descriptors: B.6.3 [LOGIC DESIGN]: Design Aids—Automatic synthesis

General Terms: Algorithms, Design, Performance

Keywords: Logic synthesis, timing optimization, lookahead

1. Introduction

Timing-driven optimization during multi-level logic synthesis is a well-researched area, and several solutions have been proposed in literature [1–9]. These techniques either restructure the critical paths or perform decomposition-based resynthesis of the circuit. Restructuring techniques, such as [1–4], are computationally efficient but the improvements from these techniques are limited because restructuring is restricted to cutsets of nodes on the critical path. On the other hand, decomposition-based resynthesis techniques, such as [5–7], have immense scope for optimization because the space of possible transformations is vast. However, the algorithms proposed in literature are computationally intensive and the improvements achieved are limited by available computational resources.

Motivated by the parallel prefix problem [10, 11], this paper describes a timing-driven optimization technique for the synthesis of multi-level logic circuits. The prefix problem is one of the funda-

mental approaches to build parallel algorithms and has been extensively studied in literature, with successful applications to problems including sorting, parallelizing compilers, task scheduling, etc. The classic example of the application of the prefix problem to logic circuits is in the design of the tree-structured carry lookahead adder (CLA), where it is used to reduce the delay of carry propagation in an n -bit ripple carry adder from $O(n)$ to $O(\log(n))$.

The basic property that allows the application of prefix theory to these problems is the identification of intermediate computation that can be performed in parallel. For instance, in an n -bit binary adder, the generate bit (g_i) and propagate bit (p_i) for each bit-slice can be computed in parallel and the carry bit (c_i) is a prefix computation defined over the pair (p_j, g_j) , $1 \leq j \leq i$. The regular modular structure of the adder makes it easy to identify parallel computation of the pairs (p_j, g_j) for each bit-slice in the adder. Once the pairs are computed, parallel prefix theory is used to synthesize a fast implementation for the carry, such as the CLA. For an adder, parallel intermediate computation are simple functions with disjoint support, i.e., (g_i, p_i) and (g_j, p_j) , $i \neq j$ do not have any common inputs. However, in general multi-level logic circuits such as control logic in microprocessors, intermediate computations are complex functions with non-disjoint support due to logic sharing. Hence, identifying parallel computation to apply the principles of the prefix problem to general multi-level logic circuits is significantly more challenging.

In this paper, we propose a timing-driven optimization technique to identify intermediate computation that can be parallelized in general multi-level logic circuits. The optimized logic circuits exhibit “lookahead” properties due to the inherent parallelism among the synthesized sub-circuits, and are hence called lookahead logic circuits in this paper. When applied to a ripple carry adder, our technique can systematically derive different realizations of high-performance adders including the carry lookahead, carry skip, and carry select adders. The main advantage of our technique is that it synthesizes decomposed circuits with smaller delays in the form of lookahead logic circuits, instead of searching for decomposition functions. Lookahead logic circuits are synthesized by decomposing and reducing the Boolean functions corresponding to the internal nodes in the technology-independent representation of the original logic circuit. The lookahead logic circuits are then combined using Shannon’s decomposition and its implication-based simplifications to reconstruct the original logic circuit. Unlike prior timing-driven optimization techniques, where the synthesis of the decomposition functions is potentially expensive, our technique has the advantage that the decomposition function is discovered in the synthesized form of the lookahead logic circuit.

The performance of lookahead logic circuits is compared to the state-of-the-art academic tools SIS and ABC, and an industry-standard synthesizer. First, a case study of the n -bit ripple carry adder is used to compare our technique with these tools. Results indicate that our technique discovers several interesting decompositions with fewer levels of logic. Next, 15 circuits from the MCNC

The authors would like to acknowledge Prof. Peter Varman at Rice University and Prof. Adnan Aziz at the University of Texas, Austin for helpful discussions and suggestions. This research was supported in part by NSF CAREER Award CCF-0746850 and in part by a gift from the Fujitsu Laboratories of America.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DAC’09, July 26–31, 2009, San Francisco, California, USA.

Copyright 2009 ACM 978-1-60558-497-3/09/07 ...\$5.00.

and ISCAS benchmark suites and the OpenSPARC T1 processor are used to compare our technique to the best results obtained using these tools. On average, our technique reduces the number of logic levels in the final circuit by 40%, 56% and 22% over the best results of SIS, ABC, and the industry-standard synthesizer, respectively. When mapped delays are evaluated, our technique achieves an average reduction of 21%, 56% and 10% over the best results of SIS, ABC, and the industry-standard synthesizer, respectively. Our approach is computationally efficient, with a runtime of 100 seconds on the largest circuit considered in this paper.

This paper is organized as follows. Sec. 2 provides a background on existing timing-driven optimization techniques. Sec. 3 introduces lookahead logic circuits and describes the proposed synthesis algorithm. Sec. 4 presents a case study for n -bit adders. Sec. 5 presents results. Sec. 6 presents conclusions.

2. Timing-driven decomposition

The timing-driven optimizations proposed in literature can be broadly divided into two classes: (i) structure-based and (ii) decomposition-based. The earliest techniques for timing-driven optimization were based on restructuring critical paths to reduce circuit delay [1–4]. Most structure-based techniques have used the transformation of a ripple carry adder into a fast implementation like the CLA, carry select adder or carry bypass adder as motivation for their techniques. The technique proposed in [1], called tree height reduction, uses a CLA as motivation to reduce the delay of the circuit by rescheduling computation along critical paths. The technique presented in [2], called the generalized select transform, uses a carry select adder as a motivating example and proposes a technique that identifies late arriving signals, performs computation using both 0 and 1 as the value for the signal, and then uses that signal to select the correct output through a multiplexer. In [3], the carry bypass adder is used as motivation to propose the generalized bypass transform that reduces the critical path delay by adding redundant bypass paths and turning the critical paths into false paths. The false paths can then be eliminated without increasing the delay of the circuit using a technique presented in [4].

Decomposition-based techniques fundamentally differ from structure-based techniques in that they do not directly restructure the circuit. Instead, the circuit structure is changed as a result of changing the functionality of the internal nodes, while maintaining functional equivalence at the primary outputs. Decomposition-based techniques are capable of exploring a much richer design synthesis space, at higher computational cost, as compared to structure-based techniques. A decomposition-based technique using partial collapsing and simplification of nodes to reduce the delay is proposed in [5]. The technique proposed in [6] uses permissible functions to resynthesize sets of nodes that lie on the critical path to reduce the delay. In [7], additional redundant circuitry is added to compute the output on input patterns that sensitize the critical paths. This approach includes features of structure-based techniques, but suffers the following drawbacks. Since redundant logic is added in the form of bypass paths to the original circuit, the technique leads to a circuit with a high area and/or power footprint. The improvements in delay are limited because the additional redundant logic is restricted to only implications (0-approximation or 1-approximation) of the original function. The scalability of this approach is also limited due to a bottom-up synthesis approach for the additional redundant logic starting from an incompletely specified Boolean function with a large don't care space. Finally, although not directly related to the present work, BDD-based decomposition techniques for timing optimization have also been proposed [9, 12–15] and to this day are an active area of research.

In this paper, we propose a decomposition-based timing-driven optimization technique using lookahead logic circuits. Unlike prior techniques, where the synthesis of the decomposition functions is potentially expensive, our technique has the advantage that the decomposition functions are discovered in the synthesized form. It can explain conversion of a ripple carry adder into several fast implementations including the carry lookahead, carry select, and carry bypass adders. Like most other timing-driven optimization techniques, it also complements existing logic optimization algorithms. The next section develops the theory of lookahead logic circuits and describes the synthesis algorithm for lookahead logic circuits.

3. Lookahead logic circuits

With the background on timing-driven optimization, we use binary addition to introduce the basic principles of prefix computation and then develop the theory of lookahead logic circuits. The most common approach to speed up carry computation in adders with large operand sizes is to exploit the observation that carry propagation in binary addition is a *prefix* problem [16].

Prefix problem: Given n values z_1, z_2, \dots, z_n and an associative binary operator \otimes , the prefix computation problem, or simply the prefix problem, is to compute the n values $z_i \otimes z_{i-1} \otimes \dots \otimes z_1$, $1 \leq i \leq n$. In the context of binary addition of two n -bit numbers a and b , the carry for the i th bit can be expressed as

$$\begin{aligned} \text{Prefix element : } z_i &= (g_i, p_i) \\ \text{Operator : } (u_1, v_1) \otimes (u_2, v_2) &= (u_1 + v_1 u_2, v_1 v_2) \\ (G_{i:j}, P_{i:j}) &= (g_i, p_i) \otimes (g_{i-1}, p_{i-1}) \otimes \dots \otimes (g_j, p_j), i > j \\ c_i &= G_{i:1} + P_{i:1} c_{in}, 1 \leq i \leq n \\ c_{out} &= g_n + p_n g_{n-1} + \dots + p_n p_{n-1} \dots p_1 c_{in} \end{aligned} \quad (1)$$

where $g_i = a_i b_i$ and $p_i = a_i \oplus b_i$ represent the generate and propagate bits. Since the prefixes g_i and p_i can be computed in parallel, the prefix problem reduces to efficient prefix computation and several tree structures, with size and depth trade-offs, have been proposed in literature to realize parallel-prefix adders [11].

We make the important observation that the parallel-prefix CLA can be thought of as an optimal timing-driven decomposition for carry computation and we generalize this as follows. Consider a Boolean function $f(x_1, x_2, \dots, x_n)$ of n inputs x_1, x_2, \dots, x_n . Consider the decomposition for the Boolean function f given by the identity

$$\begin{aligned} f &= \overline{\Sigma}_l f_l + \Sigma_l \overline{\Sigma}_{l-1} f_{l-1} + \dots + \\ &\quad \Sigma_l \Sigma_{l-1} \dots \overline{\Sigma}_1 f_1 + \Sigma_l \Sigma_{l-1} \dots \Sigma_1 f_0 \end{aligned} \quad (2)$$

where Σ_i (called the window function) and f_i are all functions of x_1, x_2, \dots, x_n . By drawing an analogy to the CLA representation from equation 1, we can interpret the CLA representation from equation 2 as a *lookahead* decomposition for the Boolean function f . Here, $\overline{\Sigma}_i f_i$ corresponds to the generate bit g_i and Σ_i corresponds to the propagate function p_i , $1 \leq i \leq l$. The interesting connection between the CLA representation and the timing-driven decomposition lies in the expressions for Σ_i and f_i . Let us look at the timing-critical computation for the carry bit, c_i , of each stage of the n -bit adder. Note that c_i can be computed without the carry, c_{i-1} , of the previous stage when $a_i = b_i = 0$ ($c_i = 0$) and when $a_i = b_i = 1$ ($c_i = 1$). Thus, the case $a_i = b_i$ is not a timing-critical computation at the i th bit-slice. However, when $a_i \neq b_i$ ($a_i \oplus b_i = 1$), the carry of the previous stage is necessary to compute c_i . Hence, $a_i \neq b_i$ is a timing-critical computation at the i th bit-slice. When Σ_i is set to $a_i \oplus b_i$ and f_i is set to the value of c_i for

$\bar{\Sigma}_i = 1$, i.e., $f_i = a_i$ or $f_i = b_i$, the timing-driven decomposition for c_{out} for an n -bit adder is given by

$$\begin{aligned} c_{\text{out}} &= (a_n \oplus \bar{b}_n) a_n + \dots + (a_n \oplus b_n) \dots (a_2 \oplus b_2) (a_1 \oplus \bar{b}_1) a_1 + \\ &\quad + (a_n \oplus b_n) \dots (a_1 \oplus b_1) c_{\text{in}} \\ &= a_n b_n + \dots + (a_n \oplus b_n) \dots (a_2 \oplus b_2) a_1 b_1 + \\ &\quad + (a_n \oplus b_n) \dots (a_1 \oplus b_1) c_{\text{in}} \\ &= g_n + p_n g_{n-1} + \dots + p_n p_{n-1} \dots p_1 c_{\text{in}}, \end{aligned} \quad (3)$$

which is equivalent to the expression for c_{out} obtained using the prefix problem in equation 1.

Thus, the key contribution of this paper for timing-driven optimization is the use of information about timing critical computation to identify window functions Σ_i that produce lookahead logic circuits f_i with fewer levels of logic. The regular modular structure of a binary adder makes it easy to identify a good timing-driven decomposition, Σ_i and f_i . However, applying this technique to the synthesis of multi-level control logic circuits is challenging for the following reasons:

1. Control logic is irregular with multiple critical paths. Due to logic sharing, control logic defies the easy modularity that makes it possible to write a CLA-like representation for the Boolean expression of the critical paths.
2. The Boolean expression for the critical-path in control logic is significantly more complex, i.e., it cannot be expressed as a simple expression such as that for the carry in adders.
3. Both Σ_i and f_i for an adder have a disjoint support set for $1 \leq i \leq n$, i.e., Σ_i and Σ_j as well as f_i and f_j ($i \neq j$) do not have common inputs in their support. However, for multi-level logic circuits, Σ_i and f_i may not have disjoint support sets. Hence, realizing them independently using separate logic circuits can be very expensive, and a good tradeoff would be to share logic between these functions.
4. Unlike an adder where the delay of each p_i and g_i term is equivalent to a single level of logic, the functions Σ_i and f_i may have different levels of logic and delays and hence combining them optimally is a challenge.

In the rest of this section, we will describe a synthesis technique for lookahead logic circuits (circuits for implementing Σ_i and f_i) that addresses these challenges.

Definitions

Decomposed logic circuit: A decomposed logic circuit is a directed acyclic graph (DAG) with nodes representing AND gates. The edge connecting a node i to another node j can be of two types: (i) complemented, when there is an inverter between the output of node i and input of node j and (ii) uncomplemented, when there is no inverter. Thus, a decomposed circuit uses AND and NOT gates as building blocks, and is referred to as an and-invert-graph (AIG).

Technology-independent network: A technology-independent network is an intermediate DAG representation of a circuit in which the internal nodes are arbitrary Boolean functions. An AIG can be converted into a technology-independent representation using clustering algorithms (“renode” command in the tool ABC [17]).

3.1 Synthesis of lookahead logic circuits

Given a decomposed circuit \mathcal{C} with n inputs, x_1, x_2, \dots, x_n , and m outputs, let $l_{\mathcal{C}}$ denote the number of levels of logic in \mathcal{C} . Although our implementation considers all outputs simultaneously, for ease of notation and without loss of generality, we refer to a primary output y containing at least one critical path, i.e., at least one

path with $l_{\mathcal{C}}$ levels of logic for the rest of this discussion. Consider the problem of obtaining a single level of timing-driven decomposition for the Boolean function, y , given by

$$y = \bar{\Sigma}_1 y_1 + \Sigma_1 y_0 \quad (4)$$

as proposed in equation 2 to improve the performance of the circuit by reducing the number of logic levels.

Attempting a function-based decomposition of the Boolean function y as shown in equation 4 has two major disadvantages. First, there is no knowledge of the circuit implementation of Σ_1 , y_0 , and y_1 . Hence, a function-based decomposition may result in a bad choice of Σ_1 , y_0 , or y_1 that may lead to a higher number of logic levels than the original circuit. Since the space of decompositions is vast, finding a good function-based decomposition based on equation 4 with lesser levels of logic than the original circuit is challenging. Second, even with the knowledge of the functions Σ_1 , y_0 , and y_1 that can potentially produce a good decomposition, directly synthesizing AIGs for these Boolean functions is a challenge and does not scale as the complexity of the function increases.

In this paper, we propose a novel approach to address the issues of finding the functions Σ_1 , y_0 , and y_1 and synthesizing their AIGs to have fewer logic levels than the original circuit. Our technique is based on two key ideas. First, we use transformations on the technology-independent network, \mathcal{T} , of the original decomposed circuit, \mathcal{C} , to synthesize the technology-independent networks for Σ_1 , y_0 , and y_1 . The transformations are made by simplifying the Boolean functions of the internal nodes in the technology-independent network to reduce the logic levels of the circuit. In this process, the functions Σ_1 , y_0 , and y_1 are derived dynamically during simplification. Second, we use global path sensitization information, extracted from the given decomposed circuit, \mathcal{C} , as a metric to guide the simplification. This ensures that the simplifications transform the functionality of the internal nodes significantly to reduce delay while preserving the functionality at the primary outputs. Our technique has two stages: (i) extracting global critical path sensitization information from \mathcal{C} and (ii) simplifying the technology-independent network \mathcal{T} to obtain the technology-independent representations of Σ_1 , y_0 , and y_1 . We will now describe each step in greater detail.

Path sensitization information: The aim of obtaining path sensitization information for output y is to identify minterms in the input space of y that are responsible for exercising *all* the speed-paths (critical or near-critical paths) in the decomposed circuit. These minterms are referred to as the timing-critical minterms or speed-path minterms in the input space of y . We shall refer to this set of minterms as the speed-path characteristic function (SPCF) for y . Thus, for a given delay Δ , the SPCF for y contains all minterms that sensitize paths of length greater than or equal to Δ . To compute the SPCF for a decomposed circuit, in which the delay is given by the number of levels of logic, Δ may be set to an integer value greater than 0. In that case, the SPCF will contain all minterms that sensitize paths with greater than or equal to Δ levels of logic.

Several algorithms have been proposed for the exact computation of the SPCF [7, 18]. These algorithms compute the exact set of minterms that sensitize paths with a delay greater than or equal to a desired value. These algorithms are path-based and require traversal of each critical path. Other algorithms that compute an approximation of the SPCF have also been proposed [19, 20]. These algorithms compute an over-approximation of the SPCF, i.e., minterms that do not sensitize critical paths may be included in the SPCF. The over-approximation algorithms are computationally more efficient than path-based algorithms because they are node-based and

require computation only at nodes that lie on the critical path. Note that the SPCF is used only as a metric to guide the synthesis of the lookahead logic circuit. Although our implementation computes the SPCF exactly, it is possible to use the over-approximation techniques to compute the SPCF for computational efficiency.

After the SPCF for output y is computed, simplifications are made to the technology-independent network \mathcal{T} . The simplification of \mathcal{T} is performed in two stages. The first simplification, referred to as the primary simplification, is used to synthesize the technology-independent networks for Σ_1 and y_0 and the second simplification, referred to as the secondary simplification, is used to synthesize the technology-independent network for y_1 . Both primary and secondary simplifications involve simplifying the Boolean expressions of the internal nodes in \mathcal{T} . As a result of the simplifications, the Boolean function for output y is transformed to y_0 in the primary simplification and to y_1 in the secondary simplification. In the primary simplification, additional logic for the technology-independent network of Σ_1 is also added to \mathcal{T} .

Primary simplification of \mathcal{T} : The pseudo-code for the primary simplification algorithm is shown in algorithm 2. The main goal of the primary simplification of \mathcal{T} is to reduce the number of logic levels by simplifying the Boolean function of the internal nodes in \mathcal{T} . When an internal node in \mathcal{T} is simplified, the original Boolean function at y is changed to y_0 . By adding additional logic to \mathcal{T} , the algorithm ensures that the window function, Σ_1 , is altered suitably (as described in algorithm 1) so that $y_0 = y$ when $\Sigma_1 = 1$. The algorithm ensures that the additional logic does not cancel the reduction in logic levels obtained as a result of the simplification of the internal node. Another goal of the primary simplification is to obtain a good window function Σ_1 . As we have seen in the carry lookahead adder example in equation 3, functions containing timing-critical minterms or speed-path minterms form good window functions. Thus, the SPCF for the output y is used as a metric to guide the simplification of the internal nodes as explained below.

Using the SPCF: Consider an internal node j in the fanin cone of output y in \mathcal{T} . Let b_j denote the Boolean function of this node. Thus, b_j is a typical Boolean function with 10–15 inputs. The SPCF contains the global critical-path sensitization minterms for output y . Let \tilde{b}_j denote the Boolean function obtained after simplification of b_j . In order to use the SPCF information for simplification of b_j , we assign a weight $w(c)$ to each prime implicant cube c in the off-set and on-set of b_j . The weight $w(c)$ is the fraction of minterms in the SPCF that will be covered in the window function Σ_1 if $\tilde{b}_j(c) = b_j(c)$. Thus, $w(c)$ is the metric based on which the Boolean function of the internal node is simplified. The cube weights can be easily computed for each node using the global Boolean functions of each node and the SPCF. Note that the cube weights for a node are computed only if the node is chosen for simplification. The function `reduce` in algorithm 2 describes the procedure for choosing nodes for simplification. The function `simplify` in algorithm 1 describes the procedure for simplifying the Boolean function of a node using the SPCF. At the end of the primary simplification, a technology-independent network for Σ_1 and y_0 is obtained for every output y with l_c levels of logic.

Secondary simplification of \mathcal{T} : The primary simplification determines the window function Σ_1 . In the secondary simplification, \mathcal{T} is reduced to generate the technology-independent network for y_1 . Thus, in the secondary simplification, the complement of the window function, $\bar{\Sigma}_1$, is used to assign cube weights for the internal nodes. However, unlike the primary simplification, where the nodes had to be carefully chosen for simplification in order

to obtain a good window function Σ_1 , the only objective of the secondary simplification is to generate the technology-independent network for y_1 . Hence, the objective is to reduce the levels of logic in \mathcal{T} as much as possible. This is done by replacing all cubes with zero weight by don't cares to simplify the Boolean function of every node. After the secondary simplification, the technology-independent network for y_1 is obtained for every output y with l_c levels of logic.

Reconstructing y : In general, equation 4 can be used to reconstruct y from Σ_1 , y_1 , and y_0 . However, there are several simplifications that can be applied when Σ_1 , y_1 , and y_0 satisfy implication properties with y . For example, consider $\bar{y}_0 \Rightarrow \bar{y}$ and $y_1 \Rightarrow y$. This means that y_1 is a 1-approximation for y and y_0 is a 0-approximation for y . This can be used to reduce y to $\Sigma_1 y_0 + y_1$. In this manner, 28 unique implication-based rules can be identified for the simplification of the Shannon decomposition in equation 4. We do not list them in the paper for brevity. In our optimization runs, we have observed that the implication-based rules are frequently used to reduce the number of levels of logic while reconstructing y . Finally, the technology-independent network for the reconstructed y is converted into a decomposed circuit by converting each node in the technology-independent network into an AIG. Area recovery is then performed using standard redundancy elimination algorithms.

Algorithm 1: `simplify(j)`

```

input   :  $j$  is a node in  $\mathcal{T}$  with Boolean function  $b_j$  and logic level  $l_j$ 
output  :  $\tilde{b}_j$ , the simplified Boolean function for node  $j$ 
 $S_0(S_1)$  is the minimum 0(1)-SOP of  $b_j$ 
 $w(c)$  is the weight of cube  $c$ ,  $c \in S_0$  or  $c \in S_1$ 
if  $w(c) = 0 \forall c \in S_0(S_1)$  then
     $\tilde{b}_j = 0(1)$ 
     $\mathcal{L}$  – Cubes of  $S_1$  ( $S_0$ ) in decreasing order of weight
    foreach  $c \in \mathcal{L}$  do
         $\tilde{b}_j(c) = 1(0)$ 
        Compute level(j) assuming  $\tilde{b}_j$  is the Boolean function of  $j$ 
        if level(j)  $\geq l_j$  then
             $\tilde{b}_j(c) = 0(1)$ 
     $\text{window}(j) = \tilde{b}_j(\bar{\tilde{b}}_j)$ 
else
    Both 0-SOP and 1-SOP for  $j$  have non-zero weights
    Initialize  $\tilde{b}_j = x$  /* don't care */
     $\mathcal{L}$  – Cubes of  $S_0$  and  $S_1$  in decreasing order of weight
    foreach  $c \in \mathcal{L}$  do
        Set  $\tilde{b}_j(c) = b_j(c)$ 
        Compute level(j) assuming  $\tilde{b}_j$  is the Boolean function of  $j$ 
        if level(j)  $\geq l_j$  then
             $\tilde{b}_j(c) = x$  /* don't care */
     $\text{window}(j) = \tilde{b}_j \oplus b_j$ 
mark(j)

```

Quantifying logic levels in \mathcal{T} : The logic levels for the nodes in a technology-independent network is used during the simplification of the technology-independent network in the proposed algorithm and is also used to keep track of the progress in the reduction of the logic levels. The logic level for a node j , `level(j)`, is computed using the minimum sum-of-products (SOP) representation of the off-set and on-set for the Boolean function of node j . The minimum logic level is computed for the Huffman AND tree of each prime-implicant cube in the off-set and on-set. The minimum logic level for the Huffman OR tree is then computed using the minimum logic level of each cube. The smaller logic level value, between the off-set and the on-set, is defined as the logic level for node j . In addition, to computing the level of each node, the critical inputs

Algorithm 2: $\text{reduce}(\mathcal{C}, \mathcal{T}, \text{SPCF}(l_c))$

input : Decomposed circuit \mathcal{C} with l_c levels of logic
input : $\text{SPCF}(l_c) \forall$ output $y \in \mathcal{C}$
input : Technology-independent network \mathcal{T} for \mathcal{C} with $l_{\mathcal{T}}$ levels of logic
output : Modified \mathcal{T} with y_0 and $\Sigma_1 \forall$ output $y \in \mathcal{C}$

foreach output y of \mathcal{T} **do**
 if $\text{SPCF}(y) = 0$ **then**
 continue /* output does not contain critical path */
 repeat
 $j =$ Unmarked node with highest logic level in $\text{fanin}(y)$
 $\bar{b}_j = \text{simplify}(j)$
 Recompute logic level of nodes in \mathcal{T}
 until $\text{level}(y) < l_{\mathcal{T}}$
 $y_0 = y$ /* output of the reduced network */
 $\Sigma_1 = \bigwedge_{\text{marked nodes } j} (\text{window}(j))$
 Unmark all nodes in \mathcal{T}

can also be identified for each node. An input to a node is critical if the reduction of its level is a necessary condition for reducing the level of the node. The critical inputs to a node are also used in the function `reduce` to explore candidate nodes for the function `simplify`.

4. Case study: n -bit adder

Historically, the adder has been an excellent example for evaluating various timing-driven optimization techniques primarily because of its regular prefix structure. Fast implementations of an n -bit adder include the (i) carry lookahead adder (CLA), (ii) carry select or conditional carry adder, and (iii) carry bypass or carry skip adder. In Sec. 2, we have described how existing timing-driven optimization techniques have used one of these adders as a motivating example to develop timing-driving optimizations for general multi-level logic circuits. In contrast, our timing-driven optimization technique can be used to derive *all* these fast adders from a ripple carry adder. Let a and b be two 2-bit binary numbers and c_{in} be the carry-in bit. Let y denote the two bit sum and c_{out} denote the carry. Let $g_i = a_i b_i$ denote the generate bit and $p_i = a_i + b_i$ denote the propagate bit.

The simplest implementation of an n -bit adder is a ripple carry adder that can be realized by linearly cascading n full adders. Although the ripple carry-adder has a small area, the critical path delay of the ripple carry adder is $O(n)$. The carry-propagation logic is the most delay-intensive operation in a ripple carry adder. In a 2-bit ripple carry adder, $c_{\text{out}} = g_2 + p_2(g_1 + p_1 c_{\text{in}})$ with 5 levels of logic. We will now explain how our timing-driven decomposition can transform a ripple carry adder into all these fast adders.

CLA (4 levels, disjoint): Based on the discussion in Sec. 3, two levels of timing-driven decomposition, i.e., (Σ_2, y_2) and (Σ_1, y_1) can be used to convert a ripple carry adder into a CLA. The window functions at the two levels are disjoint.

$$\Sigma_1 = (a_1 \oplus b_1) \text{ and } \Sigma_2 = (a_2 \oplus b_2)$$

$$y_0 = c_{\text{in}}, y_1 = a_1, \text{ and } y_2 = a_2$$

$$c_{\text{out}} = \bar{\Sigma}_2 y_2 + \Sigma_2 \bar{\Sigma}_1 y_1 + \Sigma_2 \Sigma_1 y_0$$

Carry select and carry bypass adders (4 levels, overlapping):

For the carry select and carry bypass adders, a single-level of decomposition is sufficient to realize the final implementation. However, it is important to note that 2-bit carry select and carry bypass adders have 4 levels of logic if a multiplexer is considered as a single level of logic. Both decompositions are overlapping because y_1 and y_0 have common inputs in their support. For the carry select

adder, we have:

$$\Sigma_1 = c_{\text{in}}, y_0 = g_2 + p_2 p_1, \text{ and } y_1 = g_2 + p_1 g_1$$

$$c_{\text{out}} = \bar{\Sigma}_1 y_1 + \Sigma_1 y_0$$

For the carry bypass adder, we have:

$$\Sigma_1 = p_2 p_1, y_0 = c_{\text{in}}, \text{ and } y_1 = g_2 + p_2 g_1$$

$$c_{\text{out}} = \bar{\Sigma}_1 y_1 + \Sigma_1 y_0$$

New decomposition (4 levels, overlapping): The proposed technique also reveals another decomposition of the 2-bit adder with 4 logic levels. This decomposition also falls under the category of a single-level overlapping decomposition.

$$\Sigma_1 = c_{\text{in}} + g_2 + p_2 g_1, y_0 = g_2 + p_2 p_1, \text{ and } y_1 = 0$$

$$c_{\text{out}} = \bar{\Sigma}_1 y_1 + \Sigma_1 y_0$$

From these examples, it is clear that even a simple circuit like a 2-bit adder has four different decompositions with the optimal number of logic levels. This illustrates the expressive power of overlapping timing-driven decomposition techniques to extract equivalent descriptions with area-delay tradeoffs.

For a 2-bit adder, it is easy to identify many different fast implementations. In general, for an n -bit adder ($n \geq 4$), identifying the adder implementation with the optimal number of logic levels is non-trivial. To illustrate this, we present the best results from SIS, ABC, an industry-standard synthesizer, and our technique to optimize an n -bit ($n = 2, 4, 8, 16, 32$) ripple carry adder (details of the scripts used are given in the next section). We compare the results of synthesis to the theoretical number of logic levels required to generate the carry in a tree-structured CLA for each value of n in table 1. Note that in the optimum tree-structured CLA, the critical path terminates in the output computing the most significant bit (MSB) of the sum. Hence, the optimum number of logic levels for a 2-bit tree-structured CLA is 5, even though c_{out} has 4 logic levels. The number of logic levels obtained using existing techniques is higher than the theoretical optimum for the tree-structured CLA. In contrast, our technique provides the optimum solution for $n = 2$ and returns a circuit with one level of logic less than the optimum for $n \geq 4$. This is because our approach is able to identify a Boolean factoring for the MSB of the sum and c_{out} simultaneously.

Table 1: Comparison of best AIG levels after timing optimization of an n -bit adder, $n = 2, 4, 8, 16, 32$.

n	Tree-structured CLA	SIS [21]	ABC [17]	Industry-standard synthesizer	Lookahead logic circuits
2	5	6	6	5	5
4	7	11	9	8	6
8	9	17	18	11	8
16	11	28	34	15	10
32	13	51	66	18	12

5. Results

Our timing-driven optimization technique for synthesis of lookahead logic circuits is implemented within ABC [17]. All experiments were run on a 64-bit 2.4 GHz Optron-based system with 6 GB memory. The performance of lookahead logic circuits is compared to state-of-the-art academic tools SIS and ABC, and an industry-standard synthesizer. Fifteen circuits from the MCNC and ISCAS benchmark suites and the OpenSPARC T1 processor are used to compare our technique to the best results obtained using

Table 2: Comparison of the proposed technique with the best algorithms in SIS, ABC, and the industry-standard synthesizer

Name	PI/POs	SIS [21]				ABC [17]				Industry-standard synthesizer				Lookahead logic circuits			
		Gates	Levels	Delay	Power	Gates	Levels	Delay	Power	Gates	Levels	Delay	Power	Gates	Levels	Delay	Power
rot	135/107	621	14	167.5	4.4	453	21	216	3.6	591	15	174.7	5.5	624	11	172.2	5.8
dalu	75/16	1604	20	227.4	2.8	1046	31	355.6	5	703	14	138.5	3.5	966	11	123.4	5.4
i10	257/224	2454	29	436.2	12.6	1784	32	407.1	10.1	1935	26	284.4	13.9	1931	22	262	13.5
C432	36/7	273	22	260.4	2	136	23	248.7	1.2	197	19	205.4	2.4	250	15	174.4	3
C880	60/26	376	16	177.8	3	310	21	198.9	2.4	402	17	171	3.7	280	13	141.5	2.6
C2670	233/140	765	18	224.3	6.2	555	17	188.6	4.8	599	15	151.6	6.5	973	14	144.4	9.8
C5315	178/123	1784	21	245.4	13.3	1295	32	315.9	10.7	1464	26	261.8	14.1	1655	19	222	17.7
sparc_exu_ecl_flat	572/634	2409	13	184.8	14.2	2108	13	161.4	14.5	2422	12	144.6	19.5	2191	11	150.1	20.1
lsu_stb_ctl_flat	182/169	838	16	191.3	4.8	712	20	213.3	4.3	896	16	160.2	7.3	909	12	134.4	7.3
sparc_ifu_dcl_flat	136/94	487	13	146.1	3.1	414	19	192.5	2.6	474	15	151.7	3.5	517	12	153.2	4
sparc_ifu_dec_flat	131/146	881	14	153.3	4.4	797	14	158	4.4	923	13	186.4	6.2	828	13	151.8	6.6
lsu_excpctl_flat	251/179	670	12	130.9	4.9	567	13	137.6	4.3	685	12	133.7	5.8	743	12	127.7	6.6
sparc_ttu_intctl_flat	82/80	227	8	96.5	1.2	174	11	115.4	1	304	7	77.7	2.6	266	6	76.9	2.5
sparc_ifu_fcl_flat	465/522	2254	14	247.4	15.8	2043	17	256.1	13.8	2387	14	176.6	19.1	2459	11	184.4	22.4
ttu_hyperv_flat	449/464	2397	17	198.9	14.5	2278	11	309.7	17	2573	11	128.6	20.3	2424	10	102.3	17
Relative average	–	1.15	1.09	1.22	0.80	0.88	1.28	1.40	0.70	1	1	1	1	0.98	0.78	0.90	1.10

[†] SIS: delay, rugged, algebraic and speed_up; ABC: resyn2rs;
 Industry-standard synthesizer: -map-effort high -area-effort high

these tools. Each benchmark circuit is optimized with each tool and mapped to a library of gates for the 65nm CMOS technology. For each circuit, an equivalence check is performed after optimization to ensure that the original and optimized circuits are equivalent. Our approach is computationally efficient, with a runtime of 100 seconds on the largest circuit considered in this paper.

The first two columns in table 2 give the circuit information. Subsequent columns report the number of gates in the AIG, logic levels in the AIG, technology-mapped delay, and the power consumption at 1GHz for the best results obtained with each optimization tool. Within SIS, the scripts `delay`, `rugged`, `algebraic`, and `speed_up` were used. For each benchmark circuit, the *best* results with the lowest technology-mapped delay are reported in the table. Within ABC, script `resyn2rs` was used. Within the industry-standard synthesizer, each design was compiled with the options `-map-effort high` and `-area-effort high`. The last row in the table compares the tools, on average and normalized to the industry-standard tool. On average, our technique shows a 40%, 56%, and 22% reduction in the number of logic levels in the optimized circuit over SIS, ABC, and the industry-standard synthesizer, respectively. Note that, on average, the size of the decomposed circuit obtained using our technique and the industry-standard tool are comparable. When mapped delays are evaluated, our technique achieves an average reduction of 21%, 56% and 10% over the best results of SIS, ABC, and the industry-standard synthesizer, respectively. For our technique, the trade-off for a 10% improvement in mapped delay over the industry-standard synthesizer is a 10% increase in the total power consumption.

6. Conclusions

This paper described a timing-driven optimization technique based on lookahead logic circuits. Lookahead logic circuits are synthesized by simplifying the technology-independent network of the original circuit using path sensitization information. The original logic circuit is then reconstructed from the lookahead logic circuits using Shannon’s decomposition and its implication-based simplifications. The use of a technology-independent network for simplifications provides a computationally efficient means for searching a rich space of circuit decompositions to enhance the performance of the original circuit.

References

- [1] K. Singh *et al.*, “Timing optimization of combinational logic,” in *Proc. Intl. Conference Computer-aided Design*, pp. 282–285, 1988.
- [2] C. Berman *et al.*, “Efficient techniques for timing correction,” in *Proc. Intl. Symposium on Circuits and Systems*, pp. 415–419, 1990.
- [3] P. McGeer *et al.*, “Performance enhancement through the generalized bypass transform,” in *Proc. Intl. Conference Computer-aided Design*, pp. 184–187, 1991.
- [4] K. Keutzer *et al.*, “Is redundancy necessary to reduce delay?,” *IEEE Trans. Computer-aided Design*, vol. 10, no. 4, pp. 427–435, 1991.
- [5] H. Touati *et al.*, “Delay optimization of combinational logic circuits by clustering and partial collapsing,” in *Proc. Intl. Conference Computer-aided Design*, pp. 188–191, 1991.
- [6] K. Chen *et al.*, “Timing optimization for multi-level combinational networks,” in *Proc. Design Automation Conference*, pp. 339–344, 1991.
- [7] A. Saldanha *et al.*, “Performance optimization using exact sensitization,” in *Proc. Design Automation Conference*, pp. 425–429, 1994.
- [8] G. de Micheli, *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.
- [9] M. Fujita *et al.*, “Multi-level logic optimization,” in *Logic synthesis and verification* (S. Hassoun, T. Sasao, and R. K. Brayton, eds.), ch. 2, Kluwer Academic Publishers, Boston, MA, 2002.
- [10] F. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann Publishers, 1991.
- [11] S. Lakshminarayanan and S. K. Dhall, *Parallel Computing Using the Prefix Problem*. Oxford University Press, 1994.
- [12] Y.-T. Lai *et al.*, “OBDD-based function decomposition: Algorithms and implementation,” *IEEE Trans. Computer-aided Design*, vol. 15, no. 8, pp. 977–990, 1996.
- [13] B. Becker *et al.*, “On the expressive power of OKFDDs,” *Formal Methods in System Design*, vol. 11, no. 1, pp. 5–21, 1997.
- [14] C. Yang *et al.*, “BDS: A BDD-based logic optimization system,” *IEEE Trans. Computer-aided Design*, vol. 21, no. 7, pp. 866–876, 2000.
- [15] D. Wu *et al.*, “FBDD: A folded logic synthesis system,” in *Intl. Conference on ASIC*, pp. 746–751, 2005.
- [16] R. Ladner and M. Fischer, “Parallel prefix computation,” *Journal of the ACM*, vol. 27, no. 4, pp. 831–838, 1980.
- [17] “ABC Logic synthesis tool!” Please visit the URL <http://www.eecs.berkeley.edu/~alanmi/abc/> for further details.
- [18] L. Benini *et al.*, “Telescopic units: A new paradigm for performance optimization of vlsi designs,” *IEEE Trans. Computer-aided Design*, vol. 17, no. 3, pp. 220–232, 1998.
- [19] L. Benini *et al.*, “Automatic synthesis of large telescopic units based on near-minimum timed supersetting,” *IEEE Trans. Computers*, vol. 48, no. 8, pp. 769–779, 1999.
- [20] Y.-S. Su *et al.*, “An efficient mechanism for performance optimization of variable-latency designs,” in *Proc. Design Automation Conference*, pp. 976–981, 2007.
- [21] E. Sentovich *et al.*, “SIS: A system for sequential circuit synthesis,” Tech. Rep. UCB/ERL M92/41, EECS Department, University of California, Berkeley, 1992.