Towards Optimal Performance-Area Trade-Off in Adders by Synthesis of Parallel Prefix Structures

Subhendu Roy, *Student Member, IEEE*, Mihir Choudhury, *Member, IEEE*, Ruchir Puri, *Fellow, IEEE*, and David Z. Pan, *Fellow, IEEE*

***Abstract*—This paper proposes an efficient algorithm to synthesize prefix graph structures that yield adders with the best performance-area trade-off. For designing a parallel prefix adder of a given bit-width, our approach generates prefix graph structures to optimize an objective function such as size of prefix graph subject to constraints like bit-wise output logic level. Given bit-width n and level (L) restriction, our algorithm excels the existing algorithms in minimizing the size of the prefix graph. We also prove its size-optimality when n is a power of two and $L = \log_2 n$. Besides prefix graph size optimization and having the best performance-area trade-off, our approach, unlike existing techniques, can 1) handle more complex constraints such as maximum node fanout or wire-length that impact the performance/area of a design and 2) generate several feasible solutions that minimize the objective function. Generating several size-optimal solutions provides the option to choose adder designs that mitigate constraints such as wire congestion or power consumption that are difficult to model as constraints during logic synthesis. Experimental results demonstrate that our approach improves performance by 3% and area by 9% over even a 64-bit full custom designed adder implemented in an industrial high-performance design.**

***Index Terms*—Bottom-up approach, logic synthesis, parallel prefix adder, performance-area trade-off.**

I. Introduction

DATAPATH logic constitutes a significant portion of a general purpose microprocessor and frequently occurs on the timing-critical paths in high-performance designs. Arithmetic components, such as adders, multipliers, shifters are the basic building blocks in datapath logic and hence, to a great extent dictate the performance of the entire chip. Binary addition is one of the most fundamental and widely used arithmetic operations in microprocessors. Today, adders are designed in two ways—either manually through full custom design or in an automated manner using synthesis tools. In a custom adder design methodology, a designer has to manually choose between regular adder structures such as Kogge–Stone [1], Sklansky [2], Brent–Kung [3], Han–Carlson [4], and tune physical design parameters such as placement, gate sizing, buffer optimization to maximize performance under power constraints for the target technology [5], [6]. Hence, custom adder design methodology is expensive, takes a long time to converge to a satisfactory design, and is inflexible to late design changes.

In contrast, automated synthesis approach is productive and flexible to late design changes but traditionally has lagged behind in performance as compared to custom designs. Therefore, the prevalent design approach for high-performance datapath logic continues to be custom design. In the past, several algorithms have been proposed to generate parallel prefix adders targeting minimization of the size of the prefix graph (s) under given bit-width (n) and logic level (L) constraints. A prefix graph is said to be zero deficiency if $s + L = 2n - 2$. Snir [7] has proved this theoretical bound for $L \geq 2\log_2 n - 2$ with uniform input profile. In [8], zero-deficiency prefix graphs $Z(L)$ are proposed, where $Z(L)$ has the provable maximum bit-width for a given depth L among all zero-deficiency prefix circuits. The bit-width of $Z(L)$ circuit is given by $N_Z(L) = F(L + 3) - 1$, (F denotes the fibonacci function) for $L > 1$. Compared to [7], [8] indeed gives a more general bound for size of the prefix graphs. For instance, $N_Z(6) = 33$, so for a prefix graph of bit-width 32 and level 6, the minimum achievable size $s_{\min} = 32 * 2 - 2 - 6 = 56$, which Snir fails to give as $6 < 2 * 5 - 2$.

Ladner and Fischer [9] present a recursive construction of parallel prefix graphs to obtain a trade-off between s and L, but it could not even achieve the bound provided by [7]. Other existing algorithms like a greedy depth-decreasing heuristic [10], dynamic programming based approaches [11], [12], or non-heuristic optimization [13] could achieve this bound for some cases but yield sub-optimal result as logic level constraints are reduced (for e.g., to $\log_2 n$)—which is more relevant for high performance adders. In [12], an algorithmic approach is proposed to achieve minimal delay at all output bits for uniform/non-uniform input profile, although this paper does not focus on minimizing the size of the prefix graph. Reference [13] presents an algorithm for the generation of parallel prefix structures for arbitrary level constraints to minimize the size, but it fails to get size-optimal solutions for levels closer to $\log_2 n$. Reference [14] proposes logarithmic adder structures with a fan-out of 2, and presents a model to analyze the area-delay product of those structures. However,

Manuscript received December 8, 2013; revised March 26, 2014 and June 11, 2014; accepted June 13, 2014. Date of current version September 16, 2014. This paper was recommended by Associate Editor J. Cortadella.

S. Roy and D. Z. Pan are with the Department of Electrical and Computer Engineering, University of Texas at Austin, Austin, TX 78712 USA (e-mail: subhendu@utexas.edu; dpan@ece.utexas.edu).

M. Choudhury and R. Puri are with the IBM T. J. Watson Research Center, Yorktown Heights, NY 10598 USA (e-mail: choudhury@us.ibm.com; ruchir@us.ibm.com).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2014.2341926

the key limitation of [14] is that these parallel prefix structures have more than $\log_2 n$ levels leading to a compromise in performance. Reference [15] attempts to generate a family of adder structures for $\log_2 n$ levels, but that does not give the size-optimal solutions. In [16], an exhaustive approach is attempted to explore the optimal arithmetic-circuit architectures through selective factorization, but it is very limited in terms of scalability.

The most recent approach [11], that uses dynamic programming (DP) on a restricted search space to generate a seed prefix graph followed by an area-heuristic to further reduce the size of the seed prefix graph, is the most effective in minimizing the size of the prefix graphs. However, the quality of the area-heuristic solution depends on the selection of seed solution from DP, which is not unique. Furthermore, this algorithm cannot handle fanout/wire-length constraints on nodes in the prefix graph or arrival/required time constraints on individual input/output bits that impact the performance, area, and power consumption of the adder after physical design.

To tackle these issues, this paper proposes an efficient algorithm to generate prefix graphs for synthesizing adders with the best performance-area trade-off. In this approach, prefix graph structures are constructed in bottom-up fashion by exhaustively generating all possible $n+1$ bit prefix graphs from n bit prefix graphs. For scalability to large adders up to 128 bits, our approach proposes a novel compact data structure for manipulating prefix graphs, efficient memory management techniques like lazy copy for storing several prefix graph solutions, and search space reduction strategies like level-restriction, dynamic size pruning, repeatability pruning for targeting prefix graph structures relevant for achieving the best performance-area trade-off. Furthermore, we have described a method to generate size-optimal solutions for any 2^m bit adder with level restriction of m. Compared to existing algorithms our approach has the following advantages.

1) It provides a way to generate size-optimum prefix graph structures for 2^m bit adder with level m and theoretically proves its optimality.
2) It is more effective than all existing algorithms in minimizing the size of the prefix graph for given bit-width n and arbitrary logic level, including bitwise input/output logic level constraints.
3) It provides greater opportunity for improving performance of the adder because the algorithm can handle fanout/wire-length constraints on nodes in the prefix graph and arrival/required time constraints on individual input/output bits.
4) It generates many candidate prefix graph structures for a given set of constraints, which can also be evaluated for placement and wiring congestion to yield efficient physical and routing implementation.

The rest of the paper is organized as follows. Section II describes binary addition as a prefix graph problem. Section III presents our algorithm for generating prefix graph structures. Section IV presents the results of this approach with a conclusion in Section V.

II. Preliminaries

Given an ordered n inputs $x_0, x_1, \ldots, x_{n-1}$ (where x_{n-1} is the most significant bit or MSB and x_0 is the least significant

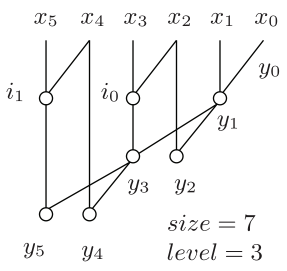

Fig. 1. Prefix graph representation.

bit LSB) and an associative operation o, prefix computation of n outputs is defined as follows:

$$y_i = x_i \ o \ x_{i-1} \ o \ldots o \ x_0 \quad \forall i \in [0, n-1] \tag{1}$$

where i-th output depends on all previous inputs x_j ($j \le i$). A prefix graph of width n is a directed acyclic graph (with n inputs/outputs) whose nodes correspond to the associative operation "o" in the prefix computation and there exists an edge from node v_i to node v_j if v_i is an operand of v_j. Fig. 1 represents a prefix graph for 6 bit. In this example, we can write y_5 as

$$\begin{aligned} y_5 = i_1 \ o \ y_3 &= (x_5 \ o \ x_4) \ o \ (i_0 \ o \ y_1) \\ &= (x_5 \ o \ x_4) \ o \ ((x_3 \ o \ x_2) \ o \ (x_1 \ o \ x_0)). \end{aligned} \tag{2}$$

Next, we will explain this prefix graph in the context of binary addition.

Binary addition problem is defined as follows [17]: given n bit augend $A = a_{n-1} \ldots a_1 a_0$ and n bit addend $B = b_{n-1} \ldots b_1 b_0$, compute the sum $S = s_{n-1} \ldots s_1 s_0$ and carry out $C_{\text{out}} = c_{n-1}$, where $s_i = a_i \oplus b_i \oplus c_{i-1}$ and $c_i = a_i b_i + a_i c_{i-1} + b_i c_{i-1}$.

With bitwise (group) generate function g (G) and propagate function p (P), n bit binary addition can be mapped to a prefix computation problem with three components as follows [18].

1) *Preprocessing:* Bitwise g, p generation

$$g_i = a_i . b_i \text{ and } p_i = a_i \oplus b_i. \tag{3}$$

2) *Prefix-Processing:* The concept of generate/propagate is extended to multiple bits and $G_{[i:j]}$, $P_{[i:j]}$ ($i \ge j$) are defined as

$$\begin{aligned} P_{[i:j]} &= \begin{cases} p_i & \text{if } i = j \\ P_{[i:k]} . P_{[k-1:j]} & \text{otherwise} \end{cases} \\ G_{[i:j]} &= \begin{cases} g_i & \text{if } i = j \\ G_{[i:k]} + P_{[i:k]} . G_{[k-1:j]} & \text{otherwise.} \end{cases} \end{aligned} \tag{4}$$

The computation for (G, P) is expressed in terms of associative operation o as

$$\begin{aligned} (G, P)_{[i:j]} &= (G, P)_{[i:k]} \ o \ (G, P)_{[k-1:j]} \\ &= (G_{[i:k]} + P_{[i:k]} . G_{[k-1:j]}, P_{[i:k]} . P_{[k-1:j]}). \end{aligned} \tag{5}$$

3) *Post-Processing:* Sum generation

$$s_i = p_i \oplus c_{i-1} \text{ and } c_i = G_{[i:0]}. \tag{6}$$

Among the three components of binary addition problem, both preprocessing and postprocessing parts are fixed structures. However, o being an associative operator, provides the flexibility of grouping the sequence of operations in prefix processing part and executing them in parallel. So the structure of the prefix graph determines the extent of parallelism.

At the technology independent level, size of the prefix graphs (# of prefix nodes) gives the area measure and the logic levels of the nodes estimate roughly the timing. It is important to note that the actual timing depends on other parameters as well like fan-out distribution and size of the prefix graph. Smaller sizes of prefix graph offer better flexibility during post-synthesis optimizations such as gate sizing, buffer insertion etc.

Equations (3)–(6) represent the Weinberger recurrance equation [19] for carry-propagation. Ling adders [19], [20] have been proposed as an alternative in the past by transforming these equations which have provided better performance. Since there is direct mapping between Weinberger's equations and Ling's equations [20], one can explore the Ling implementation of any prefix network, such as Sklansky, Kogge–Stone, etc. As another design alternative, sparse tree-adders have also been used in [21] for specific applications, however, it needs conditional sum generators as additional design blocks. In (2) or Fig. 1, we can see that the number of fan-ins for each of the associative operation o is two and thus it is often termed as radix-2 implementation of prefix network. However, there exist other choices such as radix-3 or radix-4 implementation, but the complexity is very high and not beneficial in static CMOS circuits [22]. In [23] and [24], fast domino adders are implemented using radix-4 Ling network, but domino logic has been phased out due to the high power consumption. Reference [25] demonstrates that radix-2 implementation is indeed the most energy-efficient. An implementation of mixed-radix Jackson adder has also been shown to be inefficient in terms of energy/area [22].

III. Our Approach

This section describes a compact data structure for storing and manipulating a prefix graph, efficient memory management strategies for storing several prefix graph solutions, and pruning strategies to scale our approach up to 128 bit adders. We also prove the size-optimality of any 2^m bit prefix graph with level m, generated by our approach, by incorporating several additional pruning strategies and ensuring that any pruning does not degrade the optimality of the solution. Any prefix graph solution is said to be size-optimum under certain restrictions, if the size of the prefix graph is minimum with those restrictions.

Due to the associative nature of the prefix operation o, each output for bit-index i can be constructed by combining the previous input bits $0, 1 \ldots i$ in any way keeping their relative orders intact and the number of possible ways is $catalan(i)$, where $catalan(i) = 1/(i+1)\binom{2i}{i}$. Let G_n denotes the set of all possible prefix graphs with bit-width n. Then size of G_n grows exponentially with n and is given by $catalan(n-1) * catalan(n-2) * \ldots\ catalan(0)$. For example, $|G_8| = 332972640$, $|G_{12}| = 2.29 * 10^{24}$. However, we will be exploring the set of prefix graphs with the following restrictions.

1) One of the fan-in node of any prefix node is the most recent node sharing the same MSB with that of the prefix node. For instance, in Fig. 2, x can not be a fan-in node of z. Alternatively y and c can be combined to form z. So each prefix node (p) in a prefix graph has 2 fan-in nodes. One node is vertically above p having the same MSB as that of p, we define it as trivial fan-in (tf) and the other node is termed as non-trivial fan-in node (ntf). For instance, a and c are respectively trivial and non-trivial fan-in node of b.
2) The prefix-graph is non-overlapping, i.e., for any prefix node, $LSB(tf) - MSB(ntf) = 1$. However, idempotency property can be used to generate correct and overlapping prefix trees [15].

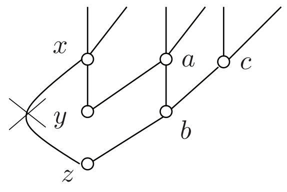

Fig. 2. Prefix graph restriction.

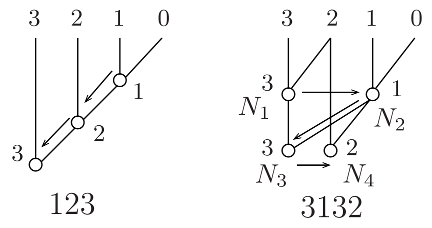

Fig. 3. Compact notation for a prefix graph.

But we impose these restrictions to reduce the search space and at the same time attempt to generate the potential candidate prefix trees which could give best performance/area trade-off after placement/routing. We denote this set of non-overlapping prefix graphs as *PG*.

However, the search space is still huge and we require compact data structure, efficient memory management, and search space reduction techniques to scale this approach.

A. Compact Notation and Data Structure

We represent the prefix graph by a sequence of indices (*seq*), where each index represents a prefix node and it is the MSB of that node. Fig. 3 illustrates the compact notation, where the sequence is determined in topological order, and in addition, precedence is given to higher significant bits in the sequence of indices. Let *SEQ* be the set of all sequences representing any prefix graph. Suppose *VS* is the set of valid sequences in our approach, where the restriction of left-to-right precedence is imposed in addition to topological ordering, inherent in *SEQ*. For instance, in Fig. 3 (right side), indices {3,1} and {3,2} occur at first and second topological levels respectively. With only topological ordering, 4 sequences are possible—"3132" ($N_1N_2N_3N_4$), "3123" ($N_1N_2N_4N_3$), "1332" ($N_2N_1N_3N_4$), "1323" ($N_2N_1N_4N_3$). Thus all 4 sequences belong to *SEQ*. But since "3" is given precedence over "1" and "2" at the first and second topological levels respectively, the only valid sequence or the only element of *VS* here is "3132". So although the mapping from *SEQ* to *PG* is many-to-one, the mapping from *VS*, a subset of *SEQ*, to *PG* is $1-1$ and bijective as well (Fig. 4). Later, we will formally prove this bijective relationship.

Algorithm 1 presents a procedure "checkValidSequence (seq, n)," which returns "true" if $seq \in VS$ representing an n bit

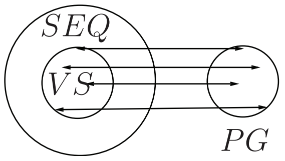

Fig. 4. Bijective mapping between *VS* and *PG*.

Algorithm 1 Procedure to Check if $seq \in VS$

```
1:  **Procedure** checkValidSequence($s$, $n$);
2:  **for** $i = 0$ to $n - 1$ **do**
3:      $bitSpan(i) = i$;
4:  **end for**
5:  **for all** $index \in seq$ from left to right **do**
6:      **if** $index > lastIndex$ **and** $bitSpan(index) - 1 \neq lastIndex$ **then**
7:          **return** false;
8:      **end if**
9:      $bitSpan(index) = bitSpan(bitSpan(index) - 1)$;
10: **end for**
11: **for** $i = 1$ to $n - 1$ **do**
12:     **if** $bitSpan(i) \neq 0$ **then**
13:         **return** false;
14:     **end if**
15: **end for**
16: **return** true;
17: **end Procedure**
```

prefix graph. Here *bitSpan*(i) at any instant of traversing the sequence represents the LSB of the node with index i, having maximum logic level at that instant. So when we start traversing *seq*, *bitSpan*(i) is equal to i and *bitSpan*(i) should be equal to 0 when the entire sequence is traversed. Lines 2-4 initialize *bitSpan*(i) with i representing the input nodes. Lines 11-14 check whether *seq* represents a prefix graph by ensuring that the *LSB* of each output node is 0, where as Lines 6-8 check the topological left to right ordering. For instance, for the sequence "3123," when the second "3" is visited, then $index = 3 > 2 = lastIndex$ indicating right-to-left ordering. So the node represented by this "3" should topologically depend on the node represented by "2" and $bitSpan(3) - 1$ should be equal to "2" to maintain the topological left-to-right ordering, but $bitSpan(3) - 1 = 2 - 1 \neq 2$. So "3123" is not a valid sequence. On the contrary, for the example sequence "3132," when the second 3 is visited, $index = 3 > 2 = lastIndex$, but $bitSpan(3) - 1 = 2 - 1 = 1 = lastIndex$. For other indices in the same sequence, $index < lastIndex$. So the condition for Line 6 in Algorithm 1 is not satisfied for any of the indices and "3132" is determined as a valid sequence.

On the other hand, we can construct a prefix graph by traversing the sequence of indices from left to right in the following way: for each index i in the sequence, we add a node p which is derived from 2 nodes—the most recent node r with index i (or input bit i) and the node just before p in the sequence (or the input bit $LSB(r) - 1$). For instance, in the sequence "3132" in Fig. 3, the node for first "3" is constructed from input bits 3 and 2, where as that for second "3" is constructed from the node for first "3" and the node (with index 1) just before it.

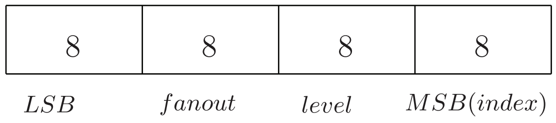

Fig. 5. Bit slicing.

Lemma 1: The mapping from *VS* to *PG* is $1 - 1$, i.e., if s_1, $s_2 \in VS$ represent the same prefix graph in *PG*, then $s_1 = s_2$

Proof: First, we will show that if we enumerate the prefix nodes of a prefix graph in a topological order from left to right, then the order of the list of the prefix nodes is fixed. For instance, in Fig. 3 (right side), this fixed order is $N_1N_2N_3N_4$. Once we prove this, the sequence representation is guaranteed to be unique as each index in the sequence corresponds to the MSB of each prefix node. We will prove this by induction. We consider that the order of the prefix nodes is fixed till some node x_n in the list. At this point, we will have a set of topologically dependent prefix-nodes (S_t) for which both trivial and non-trivial fan-in are either any node in the list till x_n or any input node. So the next node in the list will be any one node in S_t and this node will be shown unique. Since the trivial fan-in of any prefix node is the most recent node with the same MSB as that of the prefix node, for any two nodes x_i, $x_j \in S_t$, MSB(x_i) $\neq$ MSB(x_j), otherwise either x_i would topologically depend on x_j or x_j on x_i which is not possible. This implies that there exists a unique prefix node $x_u \in S_t$, such that MSB(x_u) is maximum and the next node in the list is x_u. Note that for the base case of induction, i.e., when the list is empty, the first node corresponding to the first element in the sequence is the node in the sequence having highest MSB with logic level 1 and thus unique as well. ■

Corollary 1: There exists a bijective mapping between *VS* and *PG*.

Proof: For any prefix graph in *PG* there exists a sequence representation following topological ordering from left to right. So the mapping is surjective. Also, it follows from Lemma 1 that the mapping from *VS* to *PG* is injective. Hence the Corollary is proved. ■

Apart from storing the *index*, we also need to track the *LSB*, *level*, *fanout* for each node in the prefix graph. We store all this information using a single integer for each node, and represent a prefix graph by a list/sequence of integers. Since we want to explore adders up to 128 bits and provision a carry-in as the 129th bit, we reserve 8 bits ($\lceil \log_2(129) \rceil$) for *index*, *level*, *fanout*, and *LSB*. Thus, all information for a node can be stored in a single integer as shown in Fig. 5.

This compact data structure helps in reducing memory usage and runtime (due to faster copy/delete operation for a prefix node) as compared to using a structure to store *index*, *LSB*, *level*, and *fanout* as individual integers.

B. Exhaustive Bottom-Up Enumeration

We start from a prefix graph of 2 bits (represented by a single index sequence "1") and construct the prefix graph structures for higher bits in an inductive way, i.e., given all possible prefix graphs (G_n) for n bit, we construct all possible prefix graphs (G_{n+1}) of $n + 1$ bit. The process of generating such graphs of $n+1$ bit from an element of G_n by inserting n at

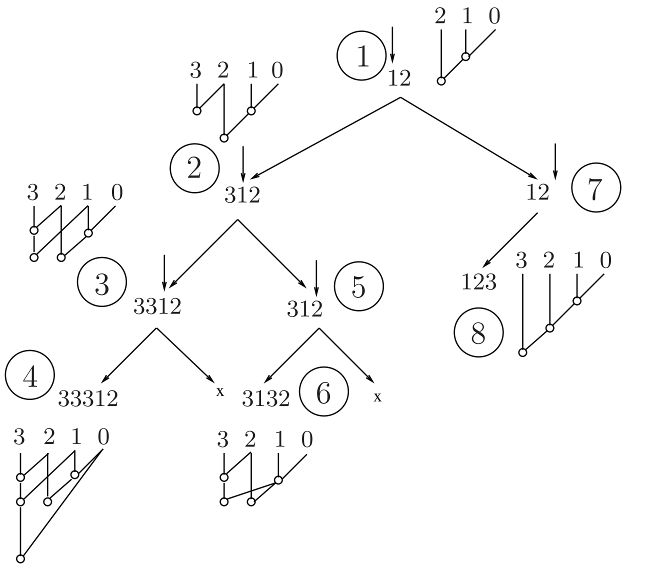

Fig. 6. Illustrative example.

appropriate positions is a recursive procedure. Fig. 6 explains this for an element "12" of G_3 with the help of a recursion tree.

At the beginning of this recursive procedure (*RP*), we have a sequence "12" (node 1) with an arrow on "1." The arrow points to the index before which "3" can be inserted. At any stage, there are two options, either insert "3" and call *RP*, or move the arrow to a suitable position and then call *RP*. This position is found by iterating the list/sequence in forward direction until *searchIndex* ($= LSB(recentNode(3)) - 1$) is found, where *recentNode*(i) signifies the most recent node with index i in the sequence. The left subtree denotes the first option and the right subtree indicates the second option. So the procedure either inserts "3" at the beginning of "12" and goes to node 2 or it goes to node 7 by moving the arrow to the appropriate position. We can see that, $searchIndex = LSB(recentNode(3)) - 1 = 3 - 1 = 2$ for this case. Similarly, for node 2, the *searchIndex* has become $2 - 1 = 1$, and so this procedure either inserts "3" (in node 3) or shifts the pointer after "1" (in node 5). The traversal is done in preorder and this recursion is continued until *LSB*(*recentNode*(3)) becomes "0" or alternatively, a 4 bit prefix graph is constructed. The right subtree of a node is not traversed if a prefix graph for 4 bits has been constructed at the left child of the node. For example, we do not traverse the right subtree of node 3 and node 5.

Algorithm 2 illustrates the steps of this exhaustive enumeration technique. The algorithm preserves the uniqueness of the solutions by inserting the indices at appropriate positions. In the "buildRecursive" procedure, *nodeList* is an STL list (*insert* and *erase* operations are thus $O(1)$ operations), *recentNode* is passed as a parameter which is used to find *searchIndex* and to track if a solution has been generated. *currIter* is the iterator corresponding to $\downarrow$ in Fig. 6. The return value of the procedure is true, when *nodeList* is a solution of G_{n+1}, thereby indicating that the right subtree of parent of *nodeList* does not require traversal.

Theorem 1: The bottom-up enumeration in Algorithm 2 is exhaustive and non-repetitive.

Proof: We construct all possible prefix graphs of bit-width $n + 1$ from any element of G_n, by inserting n at appropriate positions. At any instant, say the arrow is pointed to a node

Algorithm 2 Exhaustive Bottom-Up Enumeration

```
// Given $G_n$ construct $G_{n+1}\ldots$
Procedure buildBottomUp($G_n$)
for all $g \in G_n$ do
    buildRecursive($g$, $null$, $g.begin$, $n$);
end for
end Procedure
Procedure buildRecursive($nodeList$, $recentNode$, $currIter$, $index$)
if $recentNode \neq null$ and $LSB(recentNode) = 0$ then
    save solution $nodeList$ in $G_{n+1}$;
    return true;
end if
$searchIndex \leftarrow LSB(recentNode) - 1$;
$newIter \leftarrow nodeList.insert(currIter, index)$;
$newNode \leftarrow$ value at $newIter$;
$flag \leftarrow$ buildRecursive($nodeList$, $newNode$, $currIter$, $index$);
if $flag = true$ then
    return false;
end if
$nodeList.erase(newIter)$;
repeat
    $node \leftarrow$ value at $currIter$;
    $currIter \leftarrow currIter + 1$;
until $MSB(node) \neq searchIndex$ and $currIter \neq nodeList.end$
buildRecursive($nodeList$, $recentNode$, $currIter$, $index$);
end Procedure
```

x_i and either we insert n before x_i or we forward the pointer in the sequence for next possible insertion point, and suppose the next insertion position be after x_p, i.e., x_p is the first node in the sequence after x_i, such that $searchIndex = MSB(x_p)$. If we can prove the proposition that inserting n at any other intermediate position does not follow the topological left-to-right ordering, then we are generating all sequences following the topological left-to-right ordering (*VS*), and since the mapping from *VS* to any prefix graph of our consideration (*PG*) is bijective (by Corollary. 1), it would be sufficient to infer that Algorithm 2 is exhaustive. Also, this bijective mapping from *VS* to *PG* ensures that we are generating non-repetitive prefix graph solutions of G_{n+1}.

Suppose, for contradiction, we insert n after x_q which is an intermediate node between x_i and x_p, and the inserted node be x_n. But $MSB(x_n) = n > MSB(x_q)$, so x_q would be at right to x_n. Since x_n comes after x_q, x_n should be topologically dependent on x_q, which means the non-trivial fan-in node of x_n should depend on x_q. But x_n is just the next node to x_q, which means x_q is the non-trivial fan-in node of x_n. So $MSB(x_q) = LSB(recentNode(n)) - 1$, which is the *searchIndex*. As x_p is the first node in the sequence after x_i, for which $MSB(x_p) = searchIndex$, $x_q = x_p$. Hence the bottom-up enumeration in Algorithm 2 is exhaustive and non-repetitive. ∎

C. Efficient Recursion Implementation

The key step of Algorithm 2 is the recursive procedure as explained in Fig. 6. In a preorder traversal of typical recursion tree implementation, when we move from root node to its left subtree, a copy of the root node is stored to traverse the right subtree at later stage. In our approach, we copy the sequence only when we get a valid prefix graph, otherwise keep on modifying the sequence. As for example, we do not store the sequences ("312," "3312") in Fig. 6, i.e., when we move to the left subtree of a node in the recursion tree, we insert the index and delete it while coming back to the node in the preorder traversal, and store only the leaf nodes. This notion of late copy is motivated by a concept in object-oriented-programming, known as lazy copy or copy-on-write [26] which is a combination of deep copy and shallow copy. In lazy-copy, when an object is copied initially, a shallow copy (fast) is used and then deep copy (slow) is performed when it is absolutely necessary (for example, modifying a shared object). Lazy copy helps to significantly reduce run time by replacing list copy and delete operations with list entry insertion and deletion operations at a given position (iterator) which are $O(1)$ operations and thus improves the runtime. For the simple example shown in Fig. 6, an implementation without lazy copy needs five list copy and two list delete operations whereas an implementation with lazy copy only needs three list copy operations and no list delete operations. The benefits of lazy copy increase exponentially with bit-width.

D. Search Space Reduction

As the size of the solution space of all prefix graphs is huge, it is not feasible to generate all possible prefix graphs. Many prefix graphs are also not relevant because they do not have a good performance-area trade-off. We are interested only in generating candidate solutions to optimize performance (prefix graphs with minimum logic levels) and area (prefix graphs with minimum number of prefix nodes). Hence, the following search space reduction techniques are employed to scale this approach, however, the details of these techniques are not shown in Algorithm 2.

1) Level Pruning: The performance of an adder depends directly on the number of logic levels of the prefix graph. Our approach intends to minimize the number of prefix nodes with given bit-width and logic level (L) constraints. In Algorithm 2, we keep track of the levels of each prefix node and solutions are discarded if the level of the inserted node (or index) becomes greater than L.

2) Dynamic Size Pruning: As discussed in Section III-B, we construct the set G_{n+1} from G_n. While doing this, we prune the solution space based on size (# of prefix nodes) of elements in G_n. Let $s_{\min}$ be the size of the minimum sized prefix graph(s) of G_n. Then we prune the solutions (g) for which $size(g) > s_{\min} + \Delta$. For example, suppose the sizes of the solutions in $G_n = [9 \quad 10 \quad 11]$ and $\Delta = 2$. To construct G_{n+1}, we select the graphs of G_n in increasing order of sizes and build the elements of G_{n+1}. Let the graphs with sizes $X_1 = [12 \quad 13 \quad 14 \quad 15]$, $X_2 = [11 \quad 14]$ and $X_3 = [13 \quad 16]$ be respectively constructed from the graphs of sizes 9, 10, 11 in G_n. In this case, the minimum size solution is the solution with size 11 and so the sizes of the solutions stored in $G_{n+1} = [[12 \quad 13], [11], [13]]$. This pruning is done to choose the potential elements of G_{n+1}, which can give minimum size solution for the higher bits. The selection of Δ is critical to reduce the search space and we found empirically that $\Delta = 3$ is sufficient to get minimum size solutions for $\log_2 n$ level till

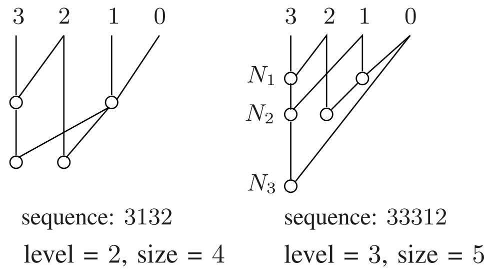

Fig. 7. 3132 is better prefix structure than 33312.

128 bit. But any kind of restriction (like fanout) on the graph structure requires higher Δ to achieve feasible solutions. In that case, we store a fixed number of solutions of G_n for each size s ($s_{\min} \leq s \leq s_{\min} + \Delta$), which allows higher Δ without increasing memory usage too much.

However, pruning the superfluous solutions after constructing the whole set G_{n+1} can cause peak memory overshoot. So we employ the strategy "Delete as early as possible," i.e., we generate solutions on the basis of current minimum size $s_{\min}^{\text{current}}$. Let us take the same example to illustrate this. In X_1, $s_{\min}^{\text{current}} = 12$ and so we do not construct the graph with size 15, as $15 > 12 + 2$. Similarly, when we get the solution with size 11 in X_2, we delete the graph with size 14 from X_1 and do not construct the graph with size 14 in X_2 and 16 in X_3. Indeed, whenever the size of the list/sequence in Algorithm 2 exceeds $s_{\min}^{\text{current}}$ by $\Delta + 1$, the flow is returned from *RP*. Apart from reducing the peak memory usage, this dynamic pruning of solutions helps in improving run time by reducing copy/delete operations.

3) Repeatability Pruning: The sequence (in our notation) denoting a prefix graph can have consecutive indices. We denote the maximum number of consecutive indices in a sequence by R. For instance, "33312" in Fig. 6 has 3 consecutive 3's in the sequence so $R = 3$. We have observed that $R = 1$ does not degrade the solution quality, but significantly reduces the search space at an early stage. For instance, in Fig. 7, "3132" is a better solution than "33312" both in terms of logic level and size. Algorithm 2 is modified to track repeatability and prune solutions with $R > 1$.

Lemma 2: If $R > 1$, the non-trivial fan-in node of the prefix node represented by the repetitive index is an input node. For instance, N_1, N_2, and N_3 in Fig. 7 are represented by the index 3 consecutively. Among them, N_2 and N_3 are the nodes where repetition of the index 3 occurs. By this lemma, the non-trivial fan-in nodes of N_2 and N_3 would be input nodes. Please note that, the non-trivial fan-in node of N_1 (represented by first occurring index) is also an input node in this example, but it is not necessarily true always.

Proof: Let p and x be 2 consecutive prefix nodes in a sequence and they have the same MSB as shown in Fig. 8. Then the trivial fan-in node of x is p and suppose the non-trivial fan-in node of x be y. We need to prove that y is an input node. We shall prove this by contradiction. Let us consider that y is a prefix node, then the relative order of the prefix nodes must be $p \rightarrow x \rightarrow y$ or $y \rightarrow p \rightarrow x$, since p and x are consecutive. $p \rightarrow x \rightarrow y$ is not possible as it violates the topological ordering and $y \rightarrow p \rightarrow x$ violates the left-to-right ordering (since y must be right to p). So y must be an input node. ■

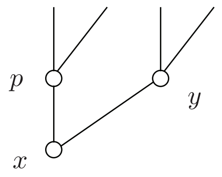

Fig. 8. Proof of Lemma 2.

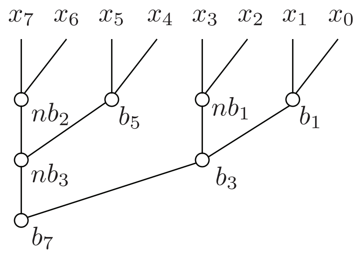

Fig. 9. Prefix structure restriction.

4) Prefix Structure Restriction: This is a special restriction in prefix graph structure for 2^m bit adders with m logic levels. For instance, if we need to construct an 8 bit adder with logic level 3, the only possible way to realize output bit 7 using the same notation as (2) is given by

$$y_7 = ((x_7 \ o \ x_6) \ o \ (x_5 \ o \ x_4)) \ o((x_3 \ o \ x_2) \ o \ (x_1 \ o \ x_0)). \quad (7)$$

So $2^m - 1$ prefix nodes are fixed and must be present in any 2^m bit adder with m level. These fixed prefix nodes form a binary-tree structure as illustrated for 8 bit in Fig. 9. Among these fixed nodes, we define the bottom-most node (or the node with highest topological level) in each bit-column of this binary-tree prefix structure to be the base node for that bit. For instance, b_3 is the base node for bit-index x_3. Please note that, we have used the terms bit-width and bit-index interchangeably. As bit-index starts from 0, the prefix graph of bit-width n is same as that for bit-index $n - 1$.

Lemma 3: Let $lv(b_i)$ denotes the level of base-node of bit-index i and $j = i - 2^{lv(b_i)}$. Then $\forall j$, s.t. $j > 0$, $lv(b_j) > lv(b_i)$.

Proof: Let a bit-index i be represented as $i + 1 = 2^{a_0} + 2^{a_1} + \cdots + 2^{a_{k-1}} + 2^{a_k}$, where $a_0 > a_1 > \ldots a_{k-1} > a_k$. Then $lv(b_i) = a_k$ (bit-index starting from 0). For example, $lv(b_5) = 1$, since $5 + 1 = 6 = 2^2 + 2^1$. Therefore, $j + 1 = 2^{a_0} + 2^{a_1} + \cdots + 2^{a_{k-1}}$, which implies that $lv(b_j) = a_{k-1} > a_k = lv(b_i)$. ■

Next, we will prove several lemmas/theorems which will hold good under this prefix structure restriction and provide a basis of generating size-optimum solutions for 2^m bit prefix graph with level m. Please note that, we are not claiming that our approach with the restrictions imposed by these lemmas/theorems will provide all size-optimum solutions. Instead, we will prove theoretically that our approach with each of these restrictions does not hamper the optimality and we will be able to obtain at least one optimum solution. In practice, our approach provides more than one optimum solution (to be discussed in Section III-E).

Any node N_1 is said to be above (or below) another node N_2 if MSB (N_1) = MSB (N_2) and level (N_1) < (or >) level (N_2). For example, node nb_2 is above the node b_7 in Fig. 9.

Lemma 4: There exists an optimum solution even when a restriction is imposed in search space by not allowing non-trivial fan-in from the nodes which are above the base nodes.

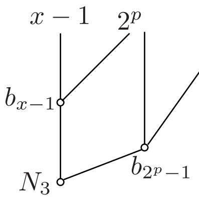

Fig. 10. Proof of Theorem 2 (2).

For example, if we do not allow non-trivial fan-in from nb_1, nb_2, nb_3 (Fig. 9) for constructing any prefix graph of bit-width 2^m with level m, we will still get a size-optimum solution.

Please refer to Appendix for proof.

Corollary 2: $\forall m$, there exists an optimum solution when all non-trivial fan-ins from bit-index $(2^m - 1)$ are taken from its base-node, b_{2^m-1}.

Proof: Since the base-node for any bit index $(2^m - 1)$ is the output node for that bit-index as well, the proof directly follows from Lemma 4. ■

Theorem 2: Let $G^{\text{opt}}_{2^m}$ be an optimum prefix graph of bit-width 2^m and level m with the imposed restriction mentioned in Lemma 4. Suppose G_x be the prefix graph of bit-width x, embedded in $G^{\text{opt}}_{2^m}$. Then G_x is an optimum prefix graph of bit-width x and level m under prefix structure restriction, if either of the following conditions are satisfied for x.

1) $x = 2^p$.
2) $x = 2^p + 2^q$.

$p, q \in Z^+$ and $p, q < m$.

Proof: Suppose, G_{2^p} is not an optimum prefix graph of bit-width 2^p and level restriction m. By Corollary 2, all non-trivial fan-ins from bit-index $2^p - 1$ are from its base-node b_{2^p-1} (this is the output node for bit-index $2^p - 1$ as well), which implies that any prefix node, which is at the right-side of the bit-index $2^p - 1$ (or alternatively bit-indices lesser than $2^p - 1$), will not be used for constructing higher output bits ($i > 2^p - 1$). So if G_{2^p} is not optimum, then we should be able to reduce the size of G_{2^p} keeping the rest of the prefix-structure, which is at the left side of bit-index $(2^p - 1)$, intact. But that reduces the size of $G^{\text{opt}}_{2^m}$, leading to contradiction.

Without any loss of generalization, we can assume $p > q$ ($p = q$ leads to condition 1) and suppose G_x is not optimum, where $x = 2^p + 2^q$. Therefore, $lv(b_{x-1}) = q$ and q prefix nodes, in the column corresponding to bit-index $x - 1$, are fixed under prefix structure restriction. The optimal way to generate the output for bit-index $x - 1$ is by combining the base nodes b_{2^p-1} $(2^p - 1{:}0)$ and b_{x-1} $(x - 1{:}2^p)$ as shown in Fig. 10, because it adds only 1 node N_3 and increases its level to its minimum possible value $p + 1$ (output bit for bit-index $x-1$ can not be realized in less than $p+1$ levels as $x-1 > 2^p$). By Lemma 4, the non-trivial fan-in from bit-index $x - 1$ can only come from b_{2^p-1} or N_3, which signifies that for any prefix node of bit-index $i > x-1$, there is no non-trivial fan-in from the bits Y for optimality, where $y \in Y$ if $x - 1 < y < 2^p - 1$. Moreover, G_{2^p} is optimum. Now, if G_x is not optimum, then we should be able to reduce the size of G_x restoring the prefix structure between the bit-ranges $2^m - 1{:}x$ and $2^p - 1{:}0$, but that reduces the size of $G^{\text{opt}}_{2^m}$, leading to contradiction. ■

Let us denote the bit-indices $0, 2, \ldots$ be even indices (E) and $1, 3, \ldots$ be odd indices (O). In our approach, we construct the prefix-graphs of higher bits in a bottom-up fashion.

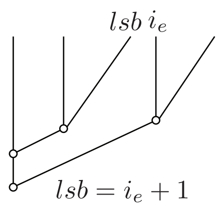

Fig. 11. Proof of Lemma 5.

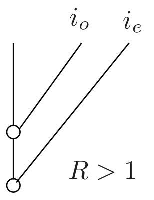

Fig. 12. Proof of Theorem 4.

Lemma 5: Under prefix structure restriction there exists an optimum solution without allowing any non-trivial fan-in from a prefix node corresponding to bit-index $i_e \in E$.

Proof: $\forall i_o \in O$, $lv(b_{i_o}) \geq 1$, which means there exists an optimum solution where any input node corresponding to odd indices is not a non-trivial fan-in node (by Lemma 4) implying that it is not essential to have any prefix node with LSB $lsb \in O$ to get an optimum solution. But to have a non-trivial fan-in from a prefix node of bit-index $i_e \in E$ we need to have at least one prefix node whose LSB $lsb = i_e + 1 \in O$ (Fig. 11). Hence the Lemma is proved. ■

Theorem 3: There exists an optimum solution under prefix structure restriction when prefix-graph of bit-index $i_o + 1$ is constructed from a prefix graph (g_{i_o}) of bit-index i_o, by adding minimum number of prefix nodes, where $i_o \in O$.

Proof: It follows from Lemma 5 that there exists an optimum solution where no non-trivial fan-in is taken from any prefix node of bit $i_e \in E$. So addition of minimum number of prefix nodes to construct a prefix-graph of bit-index $i_o + 1$ from g_{i_o} restores the optimality. ■

Theorem 4: There exists an optimum solution when search space is restricted by setting $R = 1$.

Proof: Let $R > 1$. By Lemma 2, the non-trivial fan-in node for the corresponding node is an input node. Since $\forall i_o \in O$, $lv(b_{i_o}) \geq 1$, there exists an optimum solution where any input node corresponding to odd indices is not a non-trivial fan-in node (by Lemma 4). Now it remains to prove that we do not need such non-trivial fan-in input node to be of bit-index $i_e \in E$ either. For contradiction, let us consider that input node corresponds to i_e. But, this will require a non-trivial fan-in from an input node of $i_o = i_e + 1$ (Fig. 12), which is not essential to get an optimum solution. So $R = 1$ will provide an optimum solution. ■

E. Method to Generate Size Optimum Solution for 2^m Bit Adder With Level m

Procedure "buildBottomUp" in Algorithm 2 generates G_{n+1} from G_n exhaustively and we call this procedure for bit-indices 2 to $2^m - 1$ to generate the solutions for G_{2^m}. We apply certain pruning strategies to this approach, and each pruning strategy is proven not to degrade the optimality of the solution. These strategies are as follows.

1) Enabling prefix-structure restriction, which is a constraint for generating any 2^m bit adder with level m.
2) Not allowing any non-trivial fan-in from any node above base-nodes. (Lemma 4 ensures the optimality in this case).
3) Set $\Delta = 0$ for any bit-index $x-1$, such that $x = 2^p + 2^q$ ($p, q \in Z$). We have proved in Theorem 2 that prefix graphs of bit-width x embedded in an optimum prefix graph (with the restriction imposed by Lemma 4) for 2^m bit adder with level m is also optimum for x bit adder with level m under prefix structure restriction. So keeping only the minimum size solutions at each bit-index $x - 1$ is not going to hamper the optimality of the solution.
4) Greedy construction of prefix graph of even bit-index by adding the minimum prefix node to the prefix graph of its immediate next lower bit-index (Theorem 3).
5) Set $R = 1$. (Theorem 4 ensures optimality).

With this approach, we are able to generate the size-optimum solutions for 32, 64, and 128 bits (optimum sizes are 74, 167, and 364 respectively). The total number of size-optimum solutions for them are respectively 2, 8, and 768. It is interesting to note that we also get exactly these many size-optimum solutions without using the restriction imposed by Lemma 4, Theorems 2 and 3, rather by setting $\Delta = 0, 1, 2$ for $n = 32, 64, 128$ respectively and enabling prefix structure restriction (note that without this prefix structure restriction, Δ needs to be 3 to achieve the optimum size for $n = 128$). This is intuitive as we need higher Δ (i.e., more exploration of search space) to get optimum solutions for higher bits. Increasing Δ beyond that does not reduce the size further, and this reinforces our claim of theoretical size-optimality for 2^m bit adders with level m. The run-time for generating the size-optimum solutions for 128 bit is 5.8 s, where as the same for 64 bit adder is 0.04 s.

We denote the pruning strategies 1 to 4 as the set of special pruning strategies ($S^{\text{bin}}_{\text{pruning}}$) which is effective under binary prefix structure restriction and without any other restrictions, such as fan-out. However, we will be using $S^{\text{bin}}_{\text{pruning}}$ for more general cases to be illustrated later. Please note that, we have kept the restriction $R = 1$ outside this set, as we will be using this pruning strategy more extensively and in all situations.

F. Generating Solutions for More General Case

In the earlier section, we have described a method to generate size-optimum solutions for $n = 2^m$ bit adder with level m. We have extended our approach for bit-width $n \neq 2^m$ and levels other than $\log_2 n$. We impose the pruning strategies $S^{\text{bin}}_{\text{pruning}}$ till $2^{\lceil \log_2(n) \rceil - 1}$ and then remove that restriction. For example, while we run our algorithm to generate 64 bit prefix graphs with *level* > 6, we remove the prefix structure restriction after 32 bit. The notion behind this heuristic is that keeping the balanced structure till some point would help in getting minimum-size solutions for higher bits. In addition to this, we set $\Delta = 3$ and $R = 1$ to scale the approach in general case.

TABLE I
PREFIX GRAPH SIZE FOR $\log_2 n$ LEVEL

n	Our Approach	Area Heuristic [11]	Sklansky
16	31	31	32
24	45	46	-
32	74	74	80
48	102	106	-
64	167	169	192
96	222	241	-
128	364	375	448

IV. EXPERIMENTAL RESULTS

We have implemented our approach in C++ and integrated our approach to a placement driven synthesis (PDS) [27] tool in IBM. It has been executed on a linux machine with 72GB RAM and 2.8GHz CPU. First, we present our results at the logic synthesis (technology independent) level. As the dynamic programming based area-heuristic approach presented in [11] has achieved better results compared to the other existing techniques [12], [13], we have implemented this approach as well to compare with our experimental results. Table I presents the comparison of minimum number of prefix nodes for adders with different bit-width (n) with $\log_2 n$ logic level constraint for all output bits. The number of prefix nodes for Sklansky adders are also mentioned in Table I for adders of bit-widths which are power of 2. For 128 bit adder, our approach improves Sklansky adder by 18.8% in terms of the size of the prefix graph. Table II compares the result of our algorithm with [11] for levels greater than $\log_2 n$. We can see that we have achieved theoretically possible minimum size solutions for most of the cases, where the bound is known. Prefix graph solutions for 32 bit adders with level 5 and 6 generated by our approach are shown in Fig. 13.

Next, we run our algorithm to generate the zero-deficiency prefix graphs. For example, we can build a zero-deficiency prefix graph with $L = 7$ till 54 bit and the minimum achievable size is 99. So we ran our algorithm for 54 bit graph with level restriction of 7, and got the minimum size ($s_{\min}$) as 99 which is the theoretical minimum indeed. With same constraints, the minimum size solutions for [11] is 109 and for [13] it is 104 [8]. Table III presents the result for $L = [3, 8]$ and our approach is able to achieve the theoretically possible minimum prefix graph sizes.

In Tables I–III, the input profile is uniform, i.e., the arrival times of all input bits are assumed to be the same. In Table IV, we have compared the result for non-uniform input profile. The required time of arrival for all output bits are set to 9 and the input arrival levels have been randomly generated between 0–4. Finally, we run our algorithm for 32 bit adders with non-uniform input/output profiles appeared in [13]. In these examples, the input arrival times are correlated, for example late higher words or monotonically increasing inputs, which are more common in practical situations like multiplications etc. Table V compares the result with [11] and [13] for those profiles. We can see that we have obtained comparable/better results than [11] and [13] in all cases.

As mentioned earlier, the existing automated synthesis approaches ([11], [12], [13], etc.) are not flexible in restricting parameters like fan-out, which is a critical parameter to optimize post-synthesis design performance. Usually, electrical violations at high-fanout points are mitigated by buffer-insertion and gate-sizing, but at the cost of performance. We study the impact of the parameter maximum fan-out (MFO) by plotting the worst negative slack (WNS) against the size of the prefix graph for 16 bit adders (Fig. 14). We observe that the prefix graphs of higher node count and smaller MFO are better for timing. For high-performance designs, Kogge–Stone [1] is the most effective adder structure due to the special property

TABLE II
PREFIX GRAPH SIZE FOR OTHER THAN $\log_2 n$ LEVEL

n	L	Our Approach	Area Heuristic [11]	Bound
16	5	25	25	25
	6	24	24	24
	7	23	23	23
	8	22	22	22
32	6	56	58	56
	7	55	55	55
	8	54	54	54
	9	53	53	53
64	7	126	138	-
	8	118	120	118
	9	117	117	117
	10	116	116	116
128	8	276	304	-
	9	250	284	245
	10	245	257	244

TABLE III
PREFIX GRAPH SIZE FOR ZERO-DEFICIENCY PREFIX GRAPHS

L	$N_Z(L)$	s_{min}	Our Approach	Area Heuristic [11]
3	7	9	9	9
4	12	18	18	18
5	20	33	33	34
6	33	58	58	61
7	54	99	99	104
8	88	166	166	190

TABLE IV
PREFIX GRAPH SIZE FOR NON-UNIFORM INPUT PROFILE IN A 32 BIT ADDER

Profile	Our Approach	Area Heuristic [11]
A	55	56
B	55	58
C	56	60
D	54	59
E	53	59
F	55	59
G	53	57

TABLE V
COMPARISON ON ZIMMERMANN'S EXAMPLES

DATA	Our Approach	Area Heuristic [11]	ZIM [13]
A	49	49	50
B	59	61	61
C	56	56	56
D	63	64	63
E	50	55	55
F	73	73	73
G	56	58	59
H	78	79	78
I	68	68	68

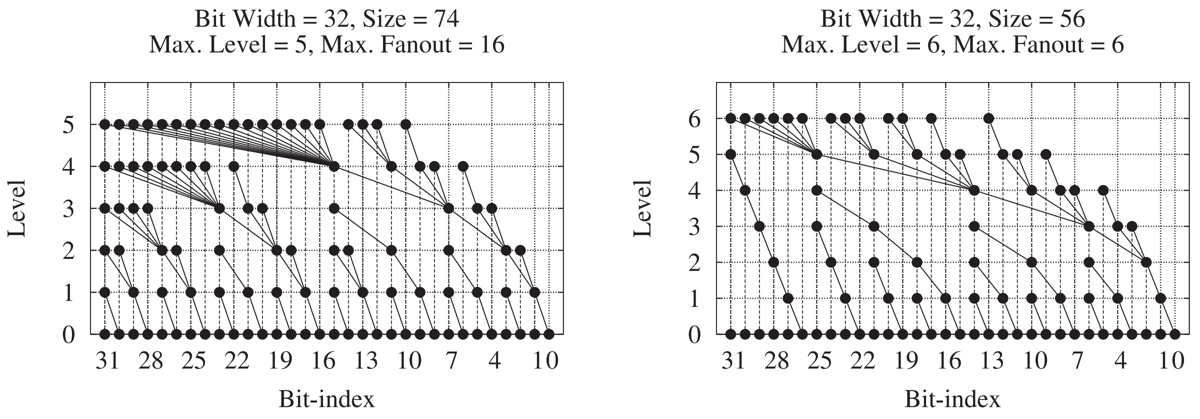
Bit Width = 32, Size = 74
Max. Level = 5, Max. Fanout = 16
Bit Width = 32, Size = 56
Max. Level = 6, Max. Fanout = 6
Level
Bit-index

Fig. 13. 32 bit prefix graphs generated by our approach with level 5 and 6.

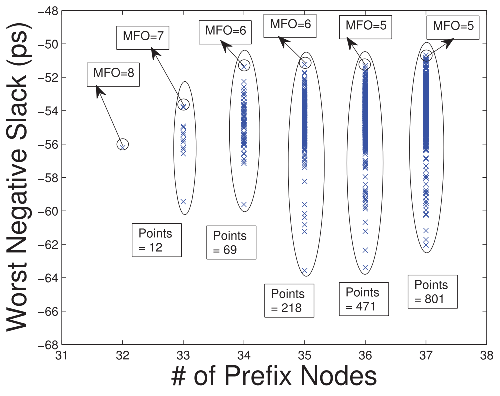

Fig. 14. # of prefix nodes versus. WNS for 16 bit adder.

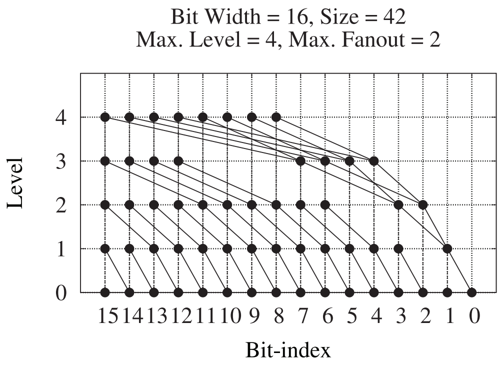

Fig. 15. Size of a 16 bit prefix graph with level 4 and fanout 2 generated by our approach is less than that of Kogge Stone by 7.

TABLE VI
COMPARISON WITH KOGGE–STONE ADDER

n	Our Approach (MFO = 2)	Our Approach (MFO = $\log_2 n$)	Kogge-Stone
8	14	13	17
16	42	35	49
32	114	89	129
64	290	238	321
128	706	631	769

TABLE VII
POST PLACEMENT COMPARISON

n	Method	Area	Worst Slack (ps)	Wire Length	TNS (ps)
8	Brent-Kung	828	-71.7	3996	-527
	Kogge-Stone	1146	-48.9	5889	-391
	Dyn. Prog.	853	-47.4	3761	-371
	Our Approach	871	-43.4	3804	-351
16	Brent-Kung	2147	-75.7	12712	-1156
	Kogge-Stone	2101	-55.5	13604	-878
	Dyn. Prog.	1980	-56.2	9776	-852
	Our Approach	2152	-50.7	11102	-812
32	Brent-Kung	4292	-107.5	26397	-3072
	Kogge-Stone	5495	-65.5	39474	-2082
	Dyn. Prog.	4538	-71.3	25784	-2096
	Our Approach	4692	-64.9	24683	-2074
64	Brent-Kung	9832	-120.3	59402	-6931
	Kogge-Stone	13389	-84.5	120600	-5181
	Dyn. Prog.	10718	-88.9	66249	-5334
	Custom	10905	-89.1	71054	-5709
	Our Approach	10048	-83.8	60450	-5230

that maximum fan-out (MFO) of a n bit adder is less than $\log_2 n$ (without any buffer insertion) and the fan-out for prefix nodes at logic level $\log_2 n - 1$ is 2. Table VI shows that, even with a fan-out restriction of 2 for all prefix nodes, the prefix graph generated by our approach has fewer prefix nodes than the prefix graph for a Kogge–Stone adder. Fig. 15 shows such an example for 16 bit. As mentioned in Section III-D2, Δ needs to be set to a higher value in this case. For instance, the parameters used to generate the 64 bit adder solution with a fan-out restriction of 2 is $\Delta = 20$, $R = 1$, and MFO = 2. However, it should be noted that although our approach scales with fan-out restriction and logic level $log_2 n$, it does not scale well with fan-out restriction and levels higher than $log_2 n$ for adders of higher bit-width ($n > 32$).

We run our approach, integrated in PDS tool, on the minimum size solutions of 8, 16, 32, 64 bit adders under tight timing constraints. A cutting-edge technology node (CMOS SOI 22nm) is used for technology mapping. In addition to this, other optimization techniques such as buffer-insertion, gate-sizing etc., which are inherent in the tool are applied followed by placement. However, we have prevented V_{th}-swapping in the placement tool so that the leakage power becomes proportional to area. We present the various metrics like area, WNS, wire-length, total-negative-slack (TNS) after placement in Table VII for the solution having best WNS. The target delay specified for 8, 16, 32, and 64 bit adders are respectively 35ps, 45ps, 65ps, and 75ps. So we can calculate the critical

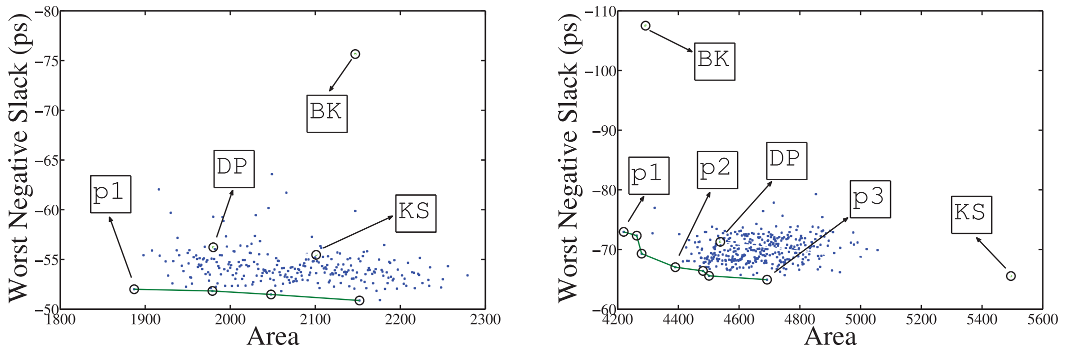

Fig. 16. Area versus worst negative slack plot for 16 and 32 bit adders.

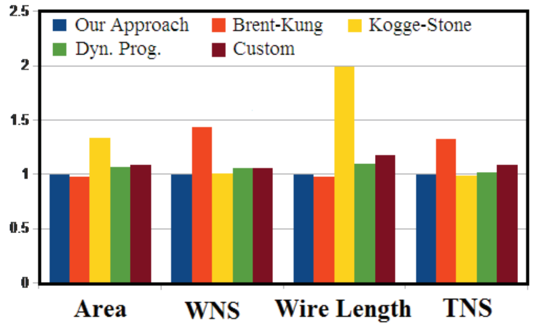

Fig. 17. 64 bit adder after placement.

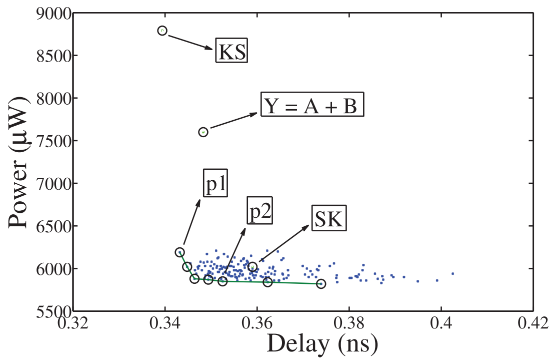

Fig. 18. Delay versus power plot for 64 bit adder.

path delay by adding the target delay and the absolute value of the WNS. For instance, the critical path delay for 64 bit Kogge–Stone adder is $75 + 84.5 = 159.5$ps. Both wirelength and area are unitless. Area is reported as the number of icells and wirelength as the number of tracks. An icell has a constant area based on pitch. Our approach is compared against regular adders like Brent–Kung (BK), Kogge–Stone (KS) adders, adders generated by dynamic programming (DP) [11], and 64 bit full custom adder (CT).

Fig. 16 represents the plot of area versus WNS for the solutions provided by our approach along with those provided by other methods. We can draw a pareto curve with the solution points obtained using our approach, which gives the option to select the individual points on the pareto curve based on area/power budget. We see that the solution points of the other methods are above and/or to the right of this curve, which indicates that we can always get some solution on the pareto-front, which is better in terms of performance and/or area than each of the other methods. For a 16 bit adder, the total number of pareto-optimal points is 4 and the single point $p1$ provides better solution than DP, KS, and BK. For a 32 bit adder, the points $p1$, $p2$, $p3$ are better solutions than BK, DP, KS respectively.

Fig. 17 compares these metrics for single solution (with best WNS) of 64 bit adder with other approaches. Our approach improves performance by 19% with 2% higher area over a Brent-Kung adder, improves performance and area by 0.4% and 33%, respectively, over a Kogge–Stone adder, improves performance and area by 3% and 6.7%, respectively over Dynamic Programming [11], and improves performance and area by 3.2% and 8.5% over a full custom adder design. Note that the performance improvement was computed based on the actual critical path delay value and not the worst negative slack. Our approach also improves wire-length and TNS over both Kogge–Stone and full custom adder design.

Since most adders today are synthesized in Design Compiler (DC) using Synopsys DesignWare, the adder architectures provided by our approach are also synthesized in DC (Version G-2012.06-SP4) and placed, routed and timed by IC Compiler (ICC) to compare with the behavioral adder implementation ($Y = A + B$) by DC. To generate high-performance adders, DC produces modified Sklansky adders consisting of alternating AOI21 and OAI21 gates, and employing gate-sizing or buffer insertion to handle the high-fanout nodes. This generally gives delay almost close to Kogge–Stone at much lower area/power and competitive power/performance/area with even custom adders. 32 nm SAED LVT cell-library [28] (available through Synopsys University Program) has been used for technology-mapping. All experimental results for DC/ICC are in "tt1p05v125c" corner, in which the supply voltage is 1.05 V and temperature is 125°C. The FO4 delay of a unit-sized inverter in this corner is 36 ps and the area of the unit-sized inverter is 1.27 μm^2.

Fig. 18 shows the delay versus power (total power i.e., leakage + switching + internal power) plot for minimum size solutions of 64 bit adder architectures provided by our approach after synthesis by DC and placed, routed by ICC. For all these runs (including those for Sklansky, Kogge–Stone and behavioral adder synthesis by DC), the target delay is

TABLE VIII
COMPARISON FOR 64 BIT ADDERS, SYNTHESIZED BY DC AND PLACED/ROUTED BY ICC

Method	Delay (ps)	Area (μm^2)	Power (mW)
Our Approach	343.2	1813.83	6.19
Y = A + B	348.3	2456.96	7.60
Sklansky	359.0	1794.86	6.02
Kogge-Stone	339.4	2611.62	8.79

set to 200 ps, the operating frequency is 1 GHz, activities at the primary inputs are 0.1, and the adders are synthesized by the command "compile_ultra." Please note that, the option "-area_high_effort_script" is on by default. We also perform some experiments by: 1) switching on the option "-timing_high_effort_script" which can further optimize at the expense of run time and 2) altering the target delay (180 ps or 220 ps), but observe that the change in delay value remains within a range of 5–10 ps. We can draw the pareto-optimal curve of delay versus power with those solutions and see that the solution provided by Sklansky adder, Kogge–Stone adder and that by behavioral adder implementation of DC are above and/or to the right side of the pareto-front. For instance, the solution p_2 in Fig. 18 improves Sklansky adder in all metrics, i.e., delay (1.8%), area (2.4%), and power (2.8%) or solution p_1 in Fig. 18 improves Kogge–Stone adder in area by 30.6% and power by 29.6% with 3.8 ps or 1.1% overhead in delay. Compared to DC behavioral adder implementation, our approach (point p_1) provides competitive delay (5 ps better) with significant area (26%) and power (18%) reduction. Table VIII compares our approach with other approaches in terms of delay, power, and area. Note that the solution with best delay is considered for this comparison.

It should be stressed that our approach generates several candidate prefix graphs for performance/area trade-off and prefix networks, which would give best performance, are not the same across different technology node and libraries. For instance, we have run our approach in PDS (IBM) with CMOS SOI 22 nm and in Synopsys DesignWare (DC + ICC) with 32 nm SAED library, and the prefix trees which have given the best performance in the two cases differ one from another. Ling transformations [20] can also be applied to the prefix graphs generated in our approach to further optimize the performance. Also, since the solutions for regular adders are located above and/or to the right side of the pareto-front, we believe that the solutions on the pareto-front can be used as alternatives for regular adders for use in custom designs.

V. CONCLUSION

In this paper, a highly efficient parallel prefix graph generation driven high performance adder synthesis technique is presented. The complexity of parallel prefix graph generation problem for adders is exponential in the number of bits. We present efficient pruning strategies and implementation techniques to scale this approach up to 128 bit adders. We have demonstrated a way to generate size-optimum prefix graphs for 2^m bit adders with level m and proved its optimality. The results, both at the technology-independent level and after physical synthesis (post placement) show that this approach significantly improves over existing techniques by yielding better quality of results in terms of both timing and wire length for high performance adders in state of the art microprocessor designs. The proposed approach improves over even the manually designed custom adders yielding, up to 3% better delay and 9% better area. As our approach can generate multiple prefix graph structures for given constraints, it provides a framework for further exploration to identify structures that can account for practical design issues like wire congestion and power consumption.

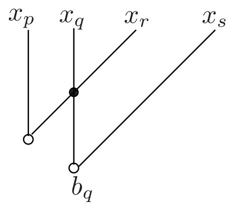

Fig. 19. $x_p : x_r \notin RBR \Longrightarrow$ non-trivial fan-in from node above b_q.

APPENDIX

Proof (Lemma 4): Let us denote any node by a triplet, viz. bit-range of the node (MSB and LSB) and level. We consider a node M_1 (msb_1, lsb_1, $level_1$) to be no worse than another node M_2 (msb_2, lsb_2, $level_2$) iff $msb_1 = msb_2$, $lsb_1 = lsb_2$ (i.e., bit-ranges of M_1 and M_2 are equal) and $level_1 \leq level_2$. We define a restricted set of bit-range (RBR) as any bit-range $msb{:}lsb \in RBR$, if $\forall i$, such that $msb > i \geq lsb$, $LSB(b_i) \geq lsb$. For instance, $7{:}4 \in RBR$, since $LSB(b_6) = 6 \geq 4$, $LSB(b_5) = LSB(b_4) = 4 \geq 4$, where as $4{:}2 \notin RBR$, since $LSB(b_3) = 0 < 2$. It is easy to notice that if there is no non-trivial fan-in from nodes above base-nodes, then there does not exist any node in the prefix graph, for which the bit-range is not in RBR, because for any bit-range $msb{:}lsb \notin RBR$, $\exists q$, such that $msb > q \geq lsb$ and $LSB(b_q) < lsb$, which is not possible unless there is a non-trivial fan-in from any node above b_q (black node marked in Fig. 19).

The structure of the proof is as follows. We will first prove the proposition (by induction) that by not allowing any non-trivial fan-in from the nodes above base-nodes, we can still realize any bit-range $br \in RBR$ with same (or less) level restriction and size, compared to allowing non-trivial fan-in from nodes above base-nodes. Once we prove this for any such bit-range, it directly follows that we can get the size-optimum solutions of 2^m bit prefix graph with level m by not allowing any non-trivial fan-in from the nodes above base-nodes, because the bit-ranges of all output bit nodes $\in RBR$.

Let b_x (x, $z+1$, r) be a base-node for bit-index x and N_1 (x, $y+1$, l_1) be any node above b_x, where $l_1 < r$ (Fig. 20). We assume that this proposition holds for bit-ranges with $MSB \leq x$ and then prove its validity for any bit-range with $MSB = x+1$ (by induction). Please note that, the proposition holds for $x = 1$ (Bit-range 1:0 can be constructed only by adding input bits for bit-index 0 and 1). The node N_1 may be used for constructing any bit-range with MSB $x+1$ by taking a non-trivial fan-in from N_1. But if we can show that there is always an alternative way by taking non-trivial fan-in from or below b_x (which is no worser than allowing the non-trivial fan-in from N_1) to construct the bit-range with MSB $x+1$, then we are done. Let we combine the node N_1 with the input node for bit-index $x+1$ to get N_5 ($x+1$, $y+1$, l_1+1). Let N_2 (z, u, l_2) be the node for bit z, which is used for realizing any arbitrary bit-range

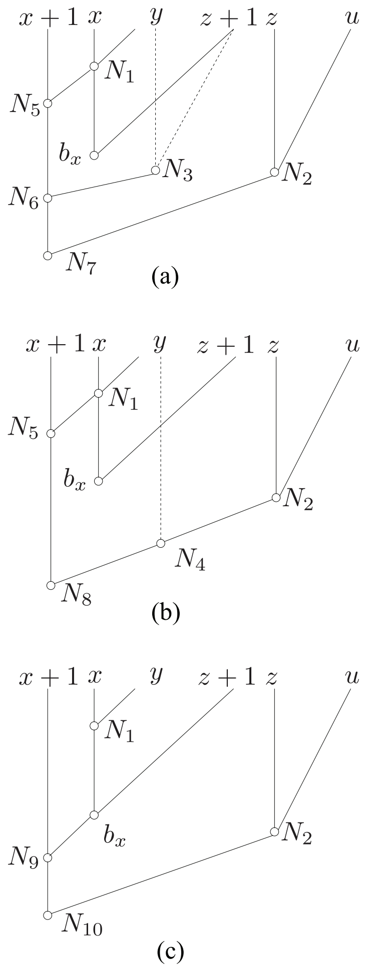

Fig. 20. Proof of lemma 4. (a) Option 1. (b) Option 2. (c) Alternative option.

$x{+}1{:}u \in RBR$ with MSB $x{+}1$. By our assumption of induction, $l_2 \geq lv(b_z)$ and $lv(b_z) > lv(b_x) = r$ (by Lemma 3). Therefore, $l_2 > r$.

Now, there are 2 options to get $x+1 : u$ by using nodes N_5 and N_2. Firstly, we can combine N_5 and some node N_3 $(y, z+1, l_3)$ to generate N_6 $(x+1, z+1, l_6)$ and then combine with N_2 to generate N_7 $(x+1, u, l_7)$ [Fig. 20(a)]. $l_6 = \max(l_1+2, r+1)$ (since $x+1-z > 2^r$). Therefore, $l_7 = \max(l_1+3, r+2, l_2+1) = \max(l_1+3, l_2+1)$ (since $l_2 > r$). In the second case [Fig. 20(b)], we combine N_4 and N_5 to generate N_8 $(x{+}1, u, l_8)$, where $l_8 \geq \max(l_1{+}2, l_2{+}2)$. But we can always have an alternative choice to construct the bit-range $x{+}1 : u$ by combining b_x and the input node for bit-index $x{+}1$ and then combine with N_2 [Fig. 20(c)] to generate N_{10} $(x+1, u, l_{10})$ where $l_{10} = \max(r+2, l_2+1) = l_2+1$. Compared to both option 1 and option 2, the alternative choice adds less or equal number of nodes and still realize the same bit-range with less or same level restriction ($l_{10} < l_8$ and $l_{10} \leq l_7$).

Hence the proposition holds for any bit-range $\in RBR$ with MSB $= x+1$, given it holds for any bit-range $\in RBR$ with MSB $\leq x$. This proves the lemma. ■

Acknowledgment

The authors would like to thank R. Chhabra, currently with Broadcom, for his help in setting up DC/ICC run.

References

[1] P. M. Kogge and H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," *IEEE Trans. Comput.*, vol. C-22, no. 8, pp. 786–793, Aug. 1973.

[2] J. Sklansky, "Conditional sum addition logic," *IRE Trans. Electron. Comput.*, vol. EC-9, no. 2, pp. 226–231, Jun. 1960.

[3] R. P. Brent and H. T. Kung, "A regular layout for parallel adders," *IEEE Trans. Comput.*, vol. C-31, no. 3, pp. 260–264, Mar. 1982.

[4] T. Han and D. Carlson, "Fast area-efficient VLSI adders," in *Proc. IEEE 8th Symp. Comput. Arith. (ARITH)*, Como, Italy, May 1987, pp. 49–56.

[5] C. Zhou, B. M. Fleischer, M. Gschwind, and R. Puri, "64-bit prefix adders: Power-efficient topologies and design solutions," in *Proc. IEEE Custom Integr. Circuit Conf.*, San Jose, CA, USA, Sep. 2009, pp. 179–182.

[6] J. Liu, Y. Zhu, H. Zhu, C. K. Cheng, and J. Lillis, "Optimum prefix adders in a comprehensive area, timing and power design space," in *Proc. Asia South Pac. Des. Autom. Conf.*, Yokohama, Japan, Jan. 2007, pp. 609–615.

[7] M. Snir, "Depth-size trade-offs for parallel prefix computation," *J. Algorithms*, vol. 7, no. 2, pp. 185–201, Jun. 1986.

[8] C. K. Cheng, H. Zhu, and R. Graham, "Constructing zero-deficiency parallel prefix adder of minimum depth," in *Proc. Asia South Pacific Des. Autom. Conf.*, Jan. 2005, pp. 883–88.

[9] R. E. Ladner and M. J. Fischer, "Parallel prefix computation," *J. ACM*, vol. 27, no. 4, pp. 831–838, Oct. 1980.

[10] J. P. Fishburn, "A depth decreasing heuristic for combinational logic; or how to convert a ripple-carry adder into a carry-lookahead adder or anything in-between," in *Proc. Des. Autom. Conf.*, Orlando, FL, USA, Jun. 1990, pp. 361–364.

[11] T. Matsunaga and Y. Matsunaga, "Area minimization algorithm for parallel prefix adders under bitwise delay constraints," in *Proc. Great Lakes Symp. VLSI*, 2007, pp. 435–440.

[12] J. Liu, S. Zhou, H. Zhu, and C. K. Cheng, "An algorithmic approach for generic parallel adders," in *Proc. Int. Conf. Comput. Aided Des.*, San Jose, CA, USA, Nov. 2003, pp. 734–740.

[13] R. Zimmermann, "Non-heuristic optimization and synthesis of parallel prefix adders," in *Proc. Int. Workshop Logic Archit. Synth.*, 1996, pp. 123–132.

[14] M. Ziegler and M. Stan, "Optimal logarithmic adder structures with a fanout of two for minimizing the area-delay product," in *Proc. Int. Symp. Circuit. Syst.*, Sydney, NSW, Australia, May 2001, pp. 657–660.

[15] S. Knowles, "A family of adders," in *Proc. 15th IEEE Symp. Comput. Arithmetic*, Vail, CO, USA, 2001, pp. 277–284.

[16] A. K. Verma and P. Lenne, "Towards the automatic exploration of arithmetic-circuit architectures," in *Proc. Des. Autom. Conf.*, San Francisco, CA, USA, 2006, pp. 445–450.

[17] S. Roy, M. Choudhury, R. Puri, and D. Z. Pan, "Towards optimal performance-area trade-off in adders by synthesis of parallel prefix structures," in *Proc. 50th ACM/EDAC/IEEE Des. Autom. Conf.*, Austin, TX, USA, May/Jun. 2013, pp. 1–8.

[18] D. Harris, "A taxonomy of parallel prefix networks," in *Proc. 37th Asilomar Conf. Signals Syst. Comput.*, Nov. 2003, pp. 2213–2217.

[19] B. R. Zeydel, T. T. J. H. Kluter, and V. G. Oklobdzija, "Efficient mapping of addition recurrence algorithms in CMOS," in *Proc. 17th IEEE Symp. Comput. Arithmetic*, Jun. 2005, pp. 107–113.

[20] G. Dimitrakopoulos and D. Nikolos, "High-speed parallel-prefix VLSI ling adders," *IEEE Trans. Comput.*, vol. 54, no. 2, pp. 225–231, Feb. 2005.

[21] S. Mathew, M. Anders, R. K. Krishnamurthy, and S. Borkar, "A 4-GHz 130 nm address generation unit with 32-bit sparse-tree adder core," *IEEE J. Solid-State Circuits*, vol. 38, no. 5, pp. 689–695, May. 2003.

[22] M. Ketter *et al.*, "Implementation of 32-bit Ling and Jackson adders," in *Proc. 45th Asilomar Conf. Signals Syst. Comput. (ASILOMAR)*, Pacific Grove, CA, USA, Nov. 2011, pp. 170–175.

[23] S. Kao, R. Zlatanovici, and B. Nikolic, "A 240ps 64b carry-lookahead adder in 90nm CMOS," in *Proc. Int. Solid-State Circuits Conf.*, San Francisco, CA, USA, Feb. 2006, pp. 1735–1744.

[24] S. Naffziger, "A subnanosecond 0.5 um 64b adder design," in *Proc. IEEE Int. Solid-State Circuits Conf.*, San Francisco, CA, USA, Feb. 1996, pp. 362–363.

[25] D. Patil, M. Horowitz, R. Ho, and R. Ananthraman, "Robust energy-efficient adder topologies," in *Proc. IEEE Symp. Comput. Arithmetic*, Montepellier, France, Jun. 2007, pp. 16–28.

[26] H. Sutter, *More Exceptional C++*. Addison Wesley, 2002 [Online]. Available: http://www.gotw.ca/publications/mxc++.htm

[27] H. Ren, D. Z. Pan, and D. S. Kung, "Sensitivity guided net weighting for placement driven synthesis," in *Proc. Int. Symp. Phys. Des.*, Apr. 2004, pp. 10–17.

[28] (2014, Mar. 14). [Online]. Available: http://www.synopsys.com/Community/UniversityProgram/Pages/32-28nm-generic-library.aspx

Subhendu Roy (S'13) received the B.E. degree in electronics and telecommunication engineering from Jadavpur University, Kolkata, India, in 2006, and the M.Tech. degree in electronic systems from the Indian Institute of Technology, Bombay, Mumbai, India, in 2009. He is currently pursuing the Ph.D. degree from the Department of Electrical and Computer Engineering, University of Texas at Austin, Austin, TX, USA.

His current research interests include design automation for logic synthesis, physical design, and cross-layer reliability. He has 3 years of full-time industry experience at EDA company, Atrenta, where he was involved in developing tools in the architectural power domain and RTL domain. He also did internships at IBM T. J. Watson Research Center in 2012 and Mentor Graphics in 2013 and 2014.

Mr. Roy received the Best Paper Award from ISPD'14.

Mihir Choudhury (S'05–M'12) received the B.Tech. degree in computer science and engineering from the Indian Institute of Technology, Bombay, Mumbai, India, and the M.S. and Ph.D. degrees in computer engineering from Rice University, Houston, TX, USA.

He is a Research Staff Member at the IBM T. J. Watson Research Center, Yorktown Heights, NY, USA. His current research interests include advanced logic synthesis algorithms and high-level synthesis.

Ruchir Puri (F'07) received the bachelor's degree in electronics and communication engineering from the National Institute of Technology, Kurukshetra, India, in 1988, the master's degree in electrical engineering from the Indian Institute of Technology, Kanpur, Kanpur, India, in 1990, and the Ph.D. degree in electrical and computer Engineering from the University of Calgary, Calgary, AB, Canada, in 1994.

He is currently an IBM Fellow at IBM T. J. Watson Research Center, Yorktown Heights, NY, USA, where he leads high performance design and methodology solutions for all of IBM's enterprise server and system chip designs. He is an Inventor of over 50 U.S. patents (both issued and pending) and has authored over 120 publications on the automated design of low-power and high-performance circuits with several Best Paper awards. He is very passionate about technology among school children and has been evangelizing fun with electronics and FIRST LEGO LEAGUE Robotics in community schools.

Dr. Puri is a member of the IBM Academy of Technology and is an IBM Master Inventor. In addition, he has received the Best of IBMİ awards in both 2011 and 2012. He is a recipient of Semiconductor Research Corporation Mehboob Khan outstanding Mentor Award and has been an Adjunct Professor at the Department of Electrical Engineering, Columbia University, New York, NY, USA. In 2011, he was honored with the John Von-Neumann Chair at the Institute of Discrete Mathematics at Bonn University, Bonn, Germany, for his scientific contributions and their impact on broader society. He has received numerous accolades including the highest technical position at IBM, the IBM Fellow, which was awarded for his transformational role in microprocessor design methodology. He is also an ACM Distinguished Speaker and has been an IEEE Distinguished Lecturer. He also received the 2014 Asian American Engineer of the Year Award. He has delivered numerous keynotes and invited talks at major VLSI Design and Automation conferences, National Science Foundation and U.S. Department of Defense Research panels and has been an Editor of the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS.

David Z. Pan (S'97–M'00–SM'06–F'14) received the B.S. degree from Peking University, Beijing, China, and the M.S. and Ph.D. degrees from the University of California, Los Angeles (UCLA), Los Angeles, CA, USA.

From 2000 to 2003, he was a Research Staff Member at IBM T. J. Watson Research Center. He is currently a Full Professor and Brasfield Endowed Faculty Fellow at the Department of Electrical and Computer Engineering, University of Texas at Austin, Austin, TX, USA. He has published over 200 papers in refereed journals and conferences, and is the holder of eight U.S. patents. His current research interests include nanoscale design for manufacturability and reliability, physical design, vertical integration design and technology, and design/CAD for emerging technologies.

Prof. Pan has served as a Senior Associate Editor for *ACM Transactions on Design Automation of Electronic Systems*, an Associate Editor for the IEEE TRANSACTIONS ON COMPUTER AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION SYSTEMS, the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS—PART I, the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS—PART II, *Science China Information Sciences*, *Journal of Computer Science and Technology*, and the IEEE CAS Society Newsletter. He has served as the Chair of the IEEE CANDE Committee and the ACM/SIGDA Physical Design Technical Committee, Program/General Chair of ISPD, TPC Subcommittee Chair for DAC, ICCAD, ASPDAC, ISLPED, ICCD, ISCAS, VLSI-DAT, ISQED, and Tutorial Chair for DAC 2014, among others. He received a number of awards for his research contributions and professional services, including the SRC 2013 Technical Excellence Award, DAC Top 10 Author in Fifth Decade, DAC Prolific Author Award, 11 Best Paper Awards at premier venues (ISPD 2014, ICCAD 2013, ASPDAC 2012, ISPD 2011, IBM Research 2010 Pat Goldberg Memorial Best Paper Award in CS/EE/Math, ASPDAC 2010, DATE 2009, ICICDT 2009, SRC Techcon in 1998, 2007, and 2012), Communications of the ACM Research Highlights in 2014, ACM/SIGDA Outstanding New Faculty Award in 2005, NSF CAREER Award in 2007, SRC Inventor Recognition Award three times, IBM Faculty Award four times, UCLA Engineering Distinguished Young Alumnus Award in 2009, UT Austin RAISE Faculty Excellence Award in 2014, ISPD Routing Contest Awards in 2007, eASIC Placement Contest Grand Prize in 2009, ICCAD'12 and ICCAD'13 CAD Contest Awards, IBM Research Bravo Award in 2003, Dimitris Chorafas Foundation Research Award in 2000, and ACM Recognition of Service Award in 2007 and 2008. From 2008 to 2009, he was an IEEE CAS Society Distinguished Lecturer.