Polynomial Time Algorithm for Area and Power Efficient Adder Synthesis in High-Performance Designs

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

***Abstract*—Adders are the most fundamental arithmetic units, and often on the timing critical paths of microprocessors. Among various adder configurations, parallel prefix adders provide the best performance vs. power/area trade-off, especially for higher bit-widths. With aggressive technology scaling, the performance of a parallel prefix adder, in addition to the dependence on the logic-level, is determined by wire-length and congestion which can be mitigated by adjusting fan-out. This paper proposes a polynomial-time algorithm to synthesize n bit parallel prefix adders targeting the minimization of the size of the prefix graph with $\log_2 n$ logic level and any arbitrary fan-out restriction. A structure aware prefix node cloning is then applied to the resultant prefix adder solutions to further optimize the size of the prefix graphs. The design space exploration by our approach provides a set of pareto-optimal solutions for delay vs. power trade-off, and these pareto-optimal solutions can be used in high-performance designs instead of picking from a fixed library (Kogge–Stone, Sklansky, etc.). Experimental results demonstrate that our approach: 1) excels highly competitive industry standard Synopsys design compiler adder, regular adders such as Sklansky adder and Kogge–Stone adder, and a highly run-time/memory intensive recent algorithm in 32 nm technology node and 2) improves performance/area over even 64 bit custom designed adders targeting 22 nm technology library and implemented in an industrial high-performance design.**

***Index Terms*—Fan-out, logic synthesis, parallel prefix adder, performance-power trade-off.**

I. Introduction

ADDERS are the primary building blocks in the datapath logic of a microprocessor, and thus adder design has been always a fundamental problem in VLSI industry. Several *ad-hoc* adder structures such as the carry-skip adder, the carry select adder and the carry-lookahead adder have been proposed in the past [2]. Parallel prefix adders represent a class of general adder structures that exhibit flexible performance-area trade-off, where logic-level and fan-out play a key role. Extreme corners have been realized through regular parallel prefix structures [2] like Kogge–Stone [3] (minimal logic level and fan-out), Sklansky [4] (minimal logic level and wire-tracks), and Brent–Kung [5] (minimal fan-out and wire-tracks). In addition to these structures, Ladner–Fischer [6], Han–Carlson [7], and Knowles [8] implemented the trade-off between each pair of these corners. Custom adders are typically designed by selecting a regular adder structure followed by further refinement in design parameters. So they are very effective in optimizing power and performance for a particular technology node [9], [10] but need a significant engineering effort and so not suitable for today's aggressive turn-around-time requirement.

On the contrary, an algorithmic synthesis approach is more flexible to engineering change orders, but generally does not achieve the performance of adders designed in a custom methodology. The traditional parallel prefix adder synthesis problem is to minimize the size of the prefix graph (s) under given bit-width (n) and logic-level (L) constraints. A lot of work [1], [11]–[13] have been done to target this problem. Most of them achieve the theoretical bound for s for $L \geq 2\log_2 n - 2$, given by Snir [14], but yield sub-optimal result when L is reduced to $\log_2 n$ pertaining to high-performance adders. Moreover, wire-length, load-distribution, and congestion play important roles in determining the performance of the adders in modern space-constrained designs after placement/routing. At the logic-synthesis level, congestion and load-distribution can be controlled by constraining fan-out. However, stringent fan-out restriction with logic-level $\log_2 n$ can lead to significant wire-length cost as in Kogge–Stone, and even Sklansky can give comparable timing to Kogge–Stone with appropriate buffer-insertion [15]. Therefore, more design space exploration is necessary to strike the right balance between congestion, load distribution and wire-length cost in order to achieve the best performance-area/power trade-off.

No existing algorithm considers the restriction in fan-out in synthesizing parallel prefix structures for $L = \log_2 n$ until a very recent work [16], [17], where a comprehensive pruning-based algorithm, exercised on exhaustive bottom-up enumeration, is presented to explore several parallel prefix

Manuscript received March 28, 2015; revised July 1, 2015; accepted August 14, 2015. Date of publication September 23, 2015; date of current version April 19, 2016. This paper was recommended by Associate Editor T. Kim.

S. Roy was with The University of Texas at Austin, Austin, TX 78712 USA. He is now with Cadence Design Systems, San Jose, CA 95134 USA (e-mail: subhendu@utexas.edu).

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

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

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.2015.2481794

structures at a time. However, there are certain limitations in this work.

1) Although this approach scales well to provide minimum size solutions without any fan-out restriction, it does not scale to higher bit adders with fan-out restriction. So it can not explore the wide design space of parallel prefix adders, especially for $n \geq 64$.
2) The algorithmic complexity is exponential in n, so in spite of several pruning techniques, the run time/memory overhead is very high.

This paper presents an $O(n^2 \log_2 n)$ algorithm to synthesize n-bit parallel prefix adders of logic level $\log_2 n$ with any maximum fan-out (mfo) restriction. This is performed by first constructing a graph computing outputs for odd bit-indices with fan-out restriction of $\lfloor (\text{mfo}/2) \rfloor$ and then constructing the prefix graph by computing outputs for even bit-indices with fan-out restriction of mfo. Although the main problem has been divided into two subproblems, our algorithm can still achieve the same solution quality (i.e., the same size of the prefix graph) with the highly runtime/memory intensive approach [16] for adders of lower bit-width ($n \leq 32$). For higher bit-widths, such as $n \geq 64$, Roy *et al.* [16] failed to provide solutions in most cases, whereas this algorithm generates solution for any n. In addition, we propose an algorithm for cloning the prefix nodes to achieve further optimization on the size of the prefix graph keeping the same fan-out and logic-level constraints. Also, the proposed rewiring of the cloned nodes is experimentally demonstrated to be placement/routing friendly. Our main contributions are summarized as follows.

1) A polynomial time algorithm is presented to synthesize prefix adders of bit-width n with logic level $\log_2 n$ under any arbitrary fan-out restriction.
2) A structure aware prefix node cloning algorithm is proposed to reduce the size of the prefix graphs, which enables our approach to achieve better performance/area/power metrics after placement and routing.
3) The design space exploration by our algorithm has provided adders which excel in timing, area/power over highly competitive design compiler (DC) adder and fast regular adders, such as Sklansky and Kogge–Stone.
4) Our approach even beats 64 bit custom designed adders implemented in an industrial high-performance design. It also improves in power/performance/area over a recent highly run-time/memory intensive algorithmic synthesis approach [16].

In the next section, we give the background of the binary addition problem. The problem formulation is illustrated in Section III. Section IV describes our algorithm to synthesize an n bit adder with $\log_2 n$ level and arbitrary fan-out restriction, succeeded by prefix node cloning algorithm. Finally, Section V presents the experimental results at both logic-synthesis level and after placement/routing followed by the conclusion in Section VI.

II. Preliminaries

Binary addition problem is defined as follows: given two n bit numbers $A = a_{n-1}..a_1a_0$ and $B = b_{n-1}..b_1b_0$, compute the sum $S = s_{n-1}..s_1s_0$ and carry out $C_{\text{out}} = c_{n-1}$, where $s_i = a_i \oplus b_i \oplus c_{i-1}$ and $c_i = a_ib_i + a_ic_{i-1} + b_ic_{i-1}$.

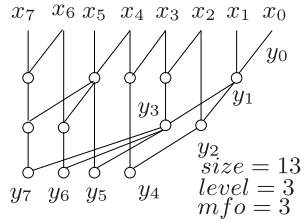

Fig. 1. Example 8 bit prefix graph.

With bitwise (group) generate function g (G) and propagate function p (P), n bit binary addition can be represented by the Weinberger's recurrence equations as follows [18].

1) *Pre-processing:* Bitwise generation of g, p

$$g_i = a_ib_i \text{ and } p_i = a_i \oplus b_i. \tag{1}$$

2) *Prefix Processing:* This part is the carry-propagation component where the concept of generate/propagate is extended to multiple bits and $G_{[i\,:\,j]}$, $P_{[i\,:\,j]}$ ($i \geq j$) are defined as

$$P_{[i\,:\,j]} = \begin{cases} p_i & \text{if } i = j \\ P_{[i\,:\,k]}P_{[k-1\,:\,j]} & \text{otherwise} \end{cases} \tag{2}$$

$$G_{[i\,:\,j]} = \begin{cases} g_i & \text{if } i = j \\ G_{[i\,:\,k]} + P_{[i\,:\,k]}G_{[k-1\,:\,j]} & \text{otherwise.} \end{cases} \tag{3}$$

The associative operation o is defined for (G, P) as

$$\begin{aligned}(G, P)_{[i:j]} &= (G, P)_{[i\,:\,k]} \; o \; (G, P)_{[k-1\,:\,j]} \\ &= \big(G_{[i\,:\,k]} + P_{[i\,:\,k]}G_{[k-1\,:\,j]}, P_{[i:k]}P_{[k-1\,:\,j]}\big).\end{aligned} \tag{4}$$

3) *Post-processing:* Sum/carry-out generation

$$s_i = p_i \oplus c_{i-1}, \quad c_i = G_{[i\,:\,0]} \text{ and } C_{\text{out}} = c_{n-1}. \tag{5}$$

The "Prefix processing" part can be mapped to a prefix graph problem with inputs $x_i = (p_i, g_i)$ and outputs $y_i = c_i$, such that y_i depends on all previous inputs x_j ($j \leq i$). Fig. 1 shows an example of such prefix graph of eight bit and we can see that $C_{\text{out}} = c_7 = y_7$ is given by

$$y_7 = ((x_7 \; o \; x_6) \; o \; (x_5 \; o \; x_4)) \; o \; ((x_3 \; o \; x_2) \; o \; (x_1 \; o \; x_0)). \tag{6}$$

In Eqn. (6) or Fig. 1, we have grouped two inputs at a time. This is called the radix-2 implementation of prefix network, since the number of fan-ins for each of the associative operation o is two. However, there exist radix 3, radix 4 or mixed-radix implementations. For instance, Ketter *et al.* [19] presented a mixed-radix Jackson adder implementation, but it is shown to be inefficient in terms of energy/area. On the contrary, radix-2 implementation has been demonstrated as the most energy-efficient option in [20]. In addition, Ling adders [18], [21] have been proposed by transforming the Weinberger's recurrence equations providing better performance. Since there is direct mapping between Weinberger's equations and Ling's equations [21], any optimized prefix network can be further explored for Ling implementation. As another design alternative, Mathew *et al.* [22]

proposed sparse tree-adders for specific applications, however, it needs conditional sum generators as additional design blocks.

III. Problem Formulation

The performance of a parallel prefix adder depends on how efficiently the prefix-processing unit is realized in terms of logic-level, fan-out, and size. Size (s) and mfo of any prefix graph are, respectively, defined as the number of prefix nodes and the mfo in that prefix graph. For instance, mfo $= 3$, $s = 13$, and $L = 3$ in Fig. 1.

Lower logic level helps in improving timing and size of the prefix graph gives a measure of area and wire-length at the logic-synthesis stage. Also, smaller size of prefix graph offers better flexibility during post-synthesis optimizations, such as gate sizing, buffer-insertion, etc., thus indirectly improving timing as well. Lower fan-out gives better timing by improving wire-congestion and load-distribution. So logic-level, size, and mfo of the prefix graph at the logic-synthesis stage altogether determine the area/performance of an adder after placement/routing.

To target high-performance designs, we fix $L = \lceil \log_2 n \rceil$, i.e., the minimum feasible logic level, and focus to explore the design space of adders by optimizing s under different fan-out restrictions. We formulate our problem as follows. Given mfo constraint of a parallel prefix adder of bit-width n with $L = \lceil \log_2 n \rceil$, minimize the size ($s$) of the prefix graph. However, this $\lceil \log_2 n \rceil$ logic level restriction can be realized in two ways: 1) the maximum level for each output bit-index m is $\lceil \log_2 n \rceil$, which can be termed as fixed level restriction and 2) the maximum level for each output bit-index m is $\lceil \log_2(m+1) \rceil$, which can be termed as bit-wise level restriction.

IV. Our Approach

A prefix graph of bit-width n computes output bits for bit-indices 0 to $n-1$. An n bit prefix graph will have $\lfloor (n/2) \rfloor$ odd bit-indices, i.e., 1, 3, $\ldots (2 \times \lfloor (n/2) \rfloor - 1)$, and $\lceil (n/2) \rceil$ even bit-indices, i.e., 0, 2, $\ldots (2 \times \lfloor (n/2) \rfloor)$. We divide the main problem into two sub-problems: 1) construct a graph (G_{odd}) which computes the outputs for odd bits with fan-out restriction of $\lfloor (\text{mfo}/2) \rfloor$ and 2) construct the prefix graph G from G_{odd} by computing the even bit outputs with fan-out restriction of mfo [23]. This division of the problem into two subproblems of computing odd and even bit outputs is motivated by the regular adder structures, such as Han–Carlson [7] or Brent–Kung [5], where the computation of odd bit outputs is followed by that of even bit outputs. Once a prefix graph (G) solution is obtained, we exercise a structure aware prefix node cloning mechanism to further improve the size of the prefix graph and rewire the cloned nodes which is favorable to placement and routing for any P&R tool.

A. Constructing Output for Odd Bit-Indices

We first generate a seed-structure for an n bit prefix graph ($G_{\text{seed}}(n)$) computing the odd bit outputs with a fan-out restriction of 2. This is followed by a heuristic which restructures $G_{\text{seed}}(n)$ to generate G_{odd} by relaxing the fan-out restriction to $\lfloor (\text{mfo}/2) \rfloor$ (where $\lfloor (\text{mfo}/2) \rfloor > 2$), thereby reducing several prefix nodes. Please note that, we do not add any prefix node of even indices at this stage. By prefix node of an odd/even index, we mean a prefix node whose most significant bit (MSB) is an odd/even index.

Algorithm 1 Generating Seed Structure $G_{\text{seed}}(n)$

```
1:  Step I:
2:  for lv = 1 to ⌈log2 n⌉ do
3:      if lv = 1 then
4:          loopIndex(lv) ← 3;
5:      else
6:          loopIndex(lv) ← 2^lv + 2^(lv−1) + 1;
7:      end if
8:      for i = 2 × ⌊n/2⌋ − 1 to loopIndex(lv) do
9:          msb(trNode) ← i;
10:         lsb(trNode) ← i − 2^(lv−1) + 1;
11:         msb(nonTrNode) ← lsb(trNode) − 1;
12:         lsb(nonTrNode) ← i − 2^lv + 1;
13:         node ← trNode + nonTrNode;
14:         bitSpan(index) ← lsb(node);
15:         i ← i − 2;
16:     end for
17: end for
18: Step II:
19: for i = 1 to 2 × ⌊n/2⌋ − 1 do
20:     msb(trNode) ← i;
21:     lsb(trNode) ← bitSpan(i);
22:     msb(nonTrNode) ← lsb(trNode) − 1;
23:     lsb(nonTrNode) ← 0;
24:     node ← trNode + nonTrNode;
25:     i ← i + 2;
26: end for
```

1) Generating Seed Structure: The generation of the seed structure is divided into two steps as shown in Algorithm 1. Fig. 2 shows the graph $G_{\text{seed}}(16)$, in which the prefix nodes generated in the first step are separated from that in the second by a dotted line. Note that 16 bit prefix adder is from bit-index 15 to 0. In the 1st step, two "for" loops are run, one within another. The outer loop runs for each level (lv), i.e., from level 1 to $\lceil \log_2 n \rceil$. For each lv, the inner for loop adds nodes at odd indices starting from $n-1$ or $n-2$ (whichever is odd) to loopIndex(lv) (Line 8). At the end of step I, for any prefix node $N_{x,l}$ of bit-index x at level l, MSB and least significant bit (LSB) are, respectively, given by $\text{msb}(N_{x,l}) = x$ and $\text{lsb}(N_{x,l}) = x - 2^l + 1$ and $N_{x,l}$ is obtained by combining $N_{x,l-1}$ (trivial fan-in node) and $N_{x-2^{l-1},l-1}$ (nontrivial fan-in node). Here, by trivial fan-in node (trNode) of a prefix node N, we mean the fan-in node sharing the same MSB as that of N. For instance, $N_{13,2}$ and $N_{9,2}$ are, respectively, the trivial and nontrivial fan-ins of $N_{13,3}$ in Fig. 2. It should be stressed that this $N_{x,l}$ notation has been used to specifically illustrate the construction of the prefix nodes in G_{odd}. However, we have mostly used the MSB:LSB notation to specify a prefix node. For instance, $N_{9,2}$ indicates the prefix node 9:6.

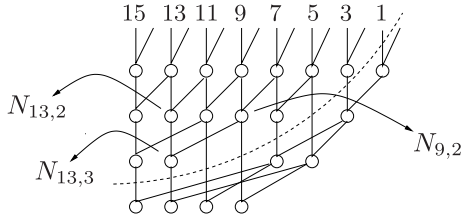

Fig. 2. Seed structure for 16 bit prefix graph.

In the second step, we add $\lfloor (n/2) \rfloor$ prefix nodes in the increasing order of odd-indices to generate the outputs for $\lfloor (n/2) \rfloor$ odd bit-indices. To do this, we keep a map (bitSpan) from the bit-index to the lsb of the highest-level prefix node of that bit-index in the existing structure. For instance, in Fig. 2, after step *I*, the highest level node of bit-index 7 is $N_{7,2}$, and its lsb is 4. So bitSpan(7) = 4 at the end of step I of Algorithm 1 and thus at step II, we add $N_{7,2}$ and $N_{3,2}$ to get the output node for bit-index 7.

It is worth-mentioning to draw an analogy of this seed structure with Han–Carlson adder. Suppose $G^{\mathrm{HC}}_{\mathrm{odd}}(n)$ be the graph structure for the computation of odd output bits in n bit Han–Carlson adder. In line 6 of Algorithm 1, if we modify the loopIndex(lv) from $2^{\mathrm{lv}} + 2^{\mathrm{lv}-1} + 1$ to $2^{\mathrm{lv}} + 1$, then the prefix nodes after step I would correspond to the black nodes in Han–Carlson adder, and the resultant $G_{\mathrm{seed}}(n)$ would be same as $G^{\mathrm{HC}}_{\mathrm{odd}}(n)$.

Lemma 1: Complexity of Algorithm 1 is $O(n \log_2 n)$.

Proof: Algorithm 1 runs in two steps. In the first step, outer loop runs for $\lceil \log_2 n \rceil$ times, the inner loop runs for $\lfloor (n/2) \rfloor$ times and $O(1)$ operations are executed in the inner loop. So the complexity of first step is $O(n \log_2 n)$. The second step runs in $O(n)$ time, so the overall complexity of Algorithm 1 is $O(n \log_2 n)$. ■

2) Fan-Out Relaxation Heuristic: Algorithm 2 shows the steps of this heuristic. We define the last fixed node for any bit-index i(lfn(i)) as the node of bit-index i with minimum level, such that any node of the same bit-index i with higher level has no nontrivial fan-out. This variable implies that any node of bit-index i with higher level than that of lfn(i), having no nontrivial fan-out, is more flexible to be removed in the graph-structure. If none of the node of bit-index i has nontrivial fan-out, then the node with level 1 is considered as the lfn(i). For instance in Fig. 2, lfn(13) = $N_{13,1}$ as $N_{13,2}$ and $N_{13,3}$ have no nontrivial fan-out. Algorithm 2 reconstructs the outputs of odd bit-indices in a decreasing order. For each odd bit-index i, it removes the nodes with higher logic level than that of lfn(i) and introduces minimum number of prefix nodes at that i keeping the fan-out restriction of $\lfloor (\mathrm{mfo}/2) \rfloor$ and level restriction (fixed or bit-wise). The condition checks for level/fan-out restriction are not shown in Algorithm 2. As we are not changing the nodes of bit-index i with lower levels than that of lfn(i), including itself, we need to find a list of bit-slices spanning from lsb(lfn(i)) − 1 to 0. This is found by calling a procedure "searchRecursive."

The procedure "searchRecursive" is a recursive subroutine with two arguments: 1) "sliceList," the existing list of bit-slices and 2) "node", the last node in the sliceList, except

Algorithm 2 Generating G_{odd} From $G_{\mathrm{seed}}(n)$ With $\lfloor (\mathrm{mfo}/2) \rfloor$

```
1: for i = 2 × ⌊n/2⌋ − 1 to 1 do
2:    for all node ∈ nodes(i) do
3:       if level(node) > level(lfn(i)) then
4:          delete node;
5:       end if
6:    end for
7:    sliceList ← createEmptyList;
8:    searchRecursive(lfn(i), sliceList);
9:    add nodes from finalSliceList to the prefix graph;
10:   i ← i − 2;
11: end for
12: Procedure searchRecursive(node, sliceList)
13: if lsb(node) = 0 and sliceList.size() < minSize then
14:   finalSliceList ← sliceList;
15:   minSize ← sliceList.size();
16: end if
17: nextIndex ← lsb(node) − 1;
18: for all nextNode ∈ nodes(nextIndex) in decreasing level
    do
19:   if level(nextNode) ≤ level(node) then
20:      break;
21:   end if
22:   sliceList.insert(nextNode);
23:   searchRecursive(nextNode, sliceList);
24:   sliceList.erase(nextNode);
25: end for
26: end Procedure
```

when "searchRecursive" is called from the main algorithm (Line 8), node is lfn(i). It also maintains a list of bit-slices finalList, which is the best bit-slice found at any instant. At any time, if the sliceList spans to bit 0, it compares the size of current sliceList and current finalList and if it finds that the former is less or equal to the latter, then finalList is changed to sliceList (Lines 13–16). However, there could be a number of choices for forming this bit-slice. We impose the restriction in the sliceList that if two nodes N_1, $N_2 \in$ sliceList and N_1 appears before N_2 in sliceList, then level(N_2) > level(N_1). Line 19 in Algorithm 2 imposes this restriction. This search-space restriction makes Algorithm 2 polynomially bounded in bit-width.

Let us illustrate this procedure with an example. Fig. 3 represents $G_{\mathrm{seed}}(20)$ and suppose we are interested in finding a prefix graph structure of bit-width 20 with mfo of 8. We can see that lfn(19) = $N_{19,1}$ and $\lfloor (\mathrm{mfo}/2) \rfloor = 4$. So the marked nodes in Fig. 3 are deleted and to find the bit-slices spanning from bit-index 17 to 0, "searchRecursive" explores the following set of bit-slices in order—[17:10 + 9:0], [17:14 + 13:0], [17:16 + 15:0], maintaining the restriction in logic level, fan-out and our imposed search-space restriction. Now there is a tie-breaking situation since these three options are of same size and Algorithm 2 prefers the last one ([17:16 + 15:0]). The intuition behind choosing this set of bit-slices is that this makes $N_{17,1}$ to be lfn(17). The other two choices ([17:10 + 9:0], [17:14 + 13:0]) make lfn(17) to be $N_{17,3}$ and $N_{17,2}$,

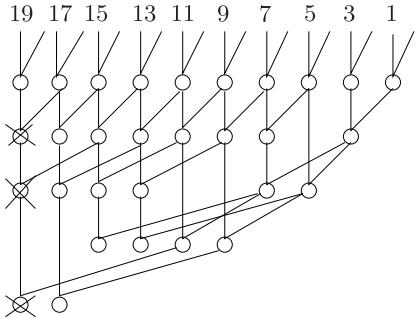

Fig. 3. [19:18 + 17:16 + 15:0] is the choice of bit-slices for bit-index 19 in Algorithm 2.

respectively. This preference offers more flexibility in reducing the number of prefix nodes for bit-index 17, as less is the level of lfn(17), more is the scope to reduce the number of prefix nodes at the later stage.

Lemma 2: "searchRecursive" procedure with a level restriction of p is an $O(p \cdot 2^p)$ operation.

Proof: "searchRecursive" procedure finds the bit-slices spanning from any bit-index to bit-index 0. For instance, we see in Fig. 3 that 19:18 is the last fixed node for bit-index 19, i.e., lfn(19) and "searchRecursive" finds the bit-slices 17:16 and 15:0, spanning from bit-index 17 to bit-index 0, thereby constructing the output node for bit 19.

Let x be the level of any bit-slice and $y = p - x$. Since the level of the bit-slices are in strictly increasing order, the level of the next bit-slice can be in the range $[x+1, p-1]$. So we can write the recursion relation in terms of y as $T(y+1) \leq T(y) + T(y-1) + T(y-2) + \cdots + T(1) + O(y)$, with $T(1) = O(1)$. Solving this recurrence relation we get, $T(y) = O(y \cdot 2^y)$. Since the maximum value of y can be p, "searchRecursive" procedure with level restriction p is $O(p \cdot 2^p)$. ■

Corollary 1: With $\log_2 n$ level restriction, "searchRecursive" procedure is an $O(n \cdot \log_2 n)$ operation.

Proof: This follows from Lemma 2 by putting $p = \log_2 n$. ■

Lemma 3: The complexity of Algorithm 2 is $O(n^2 \log_2 n)$.

Proof: The inner for loop (Lines 2–6) is executed in $O(\log_2 n)$ time and each "searchRecursive" procedure (in Line 8) is an at-most $O(n \log_2 n)$ operation (by Corollary 1). Also, the outer loop runs $\lfloor (n/2) \rfloor$ times. So the complexity of Algorithm 2 is $O(n^2 \log_2 n)$. ■

B. Constructing Output for Even Bit-Indices

The generation of output for even bit-indices consists of two stages as described in Algorithms 3 and 4. In Algorithm 3, the outputs of the even bit-indices are constructed by taking nodes from odd-bit indices using the same procedure "searchRecursive", mentioned in Algorithm 2. It is to be noted that, for outputs of odd bit-indices we modify a seed structure and then apply the procedure "searchRecursive", where the nodes of a particular bit-index are traversed in decreasing level (Line 18) to provide more flexibility in reducing the number of prefix-nodes for lower bit-indices. On the other-hand, the output for even bit-indices are generated without modifying the existing

Algorithm 3 Generating Prefix Graph G From G_{odd}

```
1: for i = 2 × ⌊n/2⌋ to 0 do
2:     node ← inNode(i);
3:     searchRecursive(node, sliceList);
4:     add nodes from finalSliceList to the prefix graph;
5:     i ← i − 2;
6: end for
```

Algorithm 4 Reducing Size of G by Local Transformations

```
1: for i = 2 × ⌊n/2⌋ to 0 do
2:     if numOfNodes(i) < 2 then
3:         continue;
4:     end if
5:     oddOutBitNode ← outNode(i − 1);
6:     if fo(oddOutBitNode) < mfo and
       level(oddOutBitNode) < maxLevel(i) then
7:         deleteNodes(i);
8:         outNode(i) ← oddOutBitNode + inNode(i);
9:         continue;
10:    end if
11:    evenOutBitNode ← outNode(i − 2);
12:    if level(evenOutBitNode) < maxLevel(i) then
13:        deleteNodes(i)
14:        Add node: node1 ← inNode(i) + inNode(i − 1);
15:        Add node: node2 ← node1 + evenOutBitNode;
16:    end if
17: end for
```

nodes in G_{odd}. So the traversal of nodes in "searchRecursive" is not mandatory to be in the order of decreasing level. At the end of Algorithm 3, a prefix graph of bit-width n is generated with the desired fan-out restriction. Note that in certain cases (for example, mfo = 2, 3) it is not possible to construct the output bit of an even index p with the fan-out restriction, and then Algorithm 1 is run with the variable i iterating from p to 2 in steps of 2. This does not increase the fan-out count of any prefix node of odd bit-index and bounds the fan-out of any prefix node of even bit-index to 2 as well.

In Algorithm 4, it is further restructured by either of the two transformations, specifically useful for fixed level restriction. The first one checks the condition (Line 6) whether it is possible to construct the output for even bit-index by connecting the output node of its previous odd-bit index [outNode($i-1$)] and the input node for i [inNode(i)] without violating the level/fan-out constraints. If it returns "true" value, this transformation is applied and continue with the next even bit-index in decreasing order. If unsuccessful at this transformation, the possibility of another local transformation is explored. It consists of adding two nodes: 1) node1 derived from inNode(i) and inNode($i-1$) and 2) node2 derived from node1 and outNode($i-2$). This transformation is also applied if it does not violate the level/fan-out constraint. The advantage of the first transformation is that it reduces the number of prefix-nodes, where as for the second one the benefit is two-fold. The first is that it can reduce the number of prefix nodes, if there were more than two prefix nodes at that bit-index before

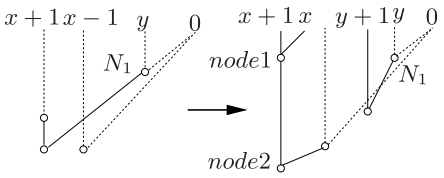

Fig. 4. Second transformation facilitating first transformation by reducing fan-out at N_1.

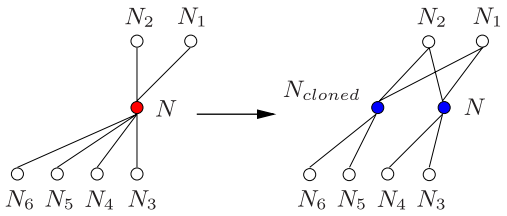

Fig. 5. Prefix node cloning.

the transformation, and the second is that this step reduces the fan-out count for output node of an odd-index, thereby facilitating the first transformation for lower bit-indices.

This situation is illustrated in Fig. 4, where the output of an even bit-index $x+1$ is constructed by adding node1 and node2 and this transformation reduces the fan-out count for the output node of odd bit-index y, i.e., N_1. Consequently, the output for bit-index $y+1$ can be now constructed by connecting N_1 and the input node of $y+1$ through first transformation, which might not have been feasible if the second transformation was not applied earlier reducing the fan-out count of N_1.

Lemma 4: The complexities of Algorithms 3 and 4 are $O(n^2 \log_2 n)$ and $O(n \log_2 n)$, respectively.

Proof: Algorithm 3 constructs a prefix graph of bit-width n from G_{odd}. The for loop runs for $O(n)$ times. Within the for loop Lines 2 and 5 are $O(1)$ operations, Line 4 is at most $O(\log_2 n)$ operation and Line 3 is an $O(n \log_2 n)$ operation. Therefore, complexity of Algorithm 3 is $O(n^2 \log_2 n)$.

For Algorithm 4, the for loop runs for $O(n)$ times and each of the operation inside the loop is either $O(1)$ or $O(\log_2 n)$ operation (Line 7). So the complexity of Algorithm 4 is $O(n \log_2 n)$. ■

C. Structure Aware Prefix Node Cloning

For any n-bit prefix graph with a given level restriction, the size of the prefix graph increases as the fan-out is restricted more and more, i.e., there is a trade-off between s and mfo. The average fan-out of the prefix nodes in the prefix graph $fo_{av} = (2s/(s+n)) \leq 2$, where s is the size of the prefix graph. This is so because there are s prefix nodes, each having two fan-ins, equals to $2s$ fan-ins which are derived from total $n+s$ nodes (n input nodes and s prefix nodes). So only a few prefix nodes in a prefix graph (having higher mfo) can have higher fan-out. For instance, 16 bit Sklansky prefix structure have mfo = 8, but only 7:0 node have fan-out of 8, and except 3 prefix nodes, all prefix nodes have less than 4 fan-out. In this context, we introduce the concept of prefix node cloning, which clones the prefix nodes with higher fan-out to reduce its fan-out count. This technique can be used to reduce the mfo of the prefix graph by adding several cloned nodes, and thereby increasing s. This approach also performs a trade-off between s and mfo, however, happens to be a better trade-off than that without prefix node cloning. Before explaining this in detail, we will first illustrate the operation of prefix node cloning by an example.

Fig. 5 shows such an example. The node N is derived from the prefix nodes N_1 and N_2, and driving four prefix nodes N_3, N_4, N_5 and N_6. So the fan-out count of N is 4. A cloned node of N is generated, called N_{cloned}, and each of N and N_{cloned} is derived from N_1 and N_2. In addition, N_{cloned} drives N_5 and N_6, and N drives only N_3 and N_4. So the fan-outs of N are distributed among N and N_{cloned} reducing the fan-out count from 4 to 2 by adding one prefix node. It should be noted that the prefix node cloning preserves the functionality.

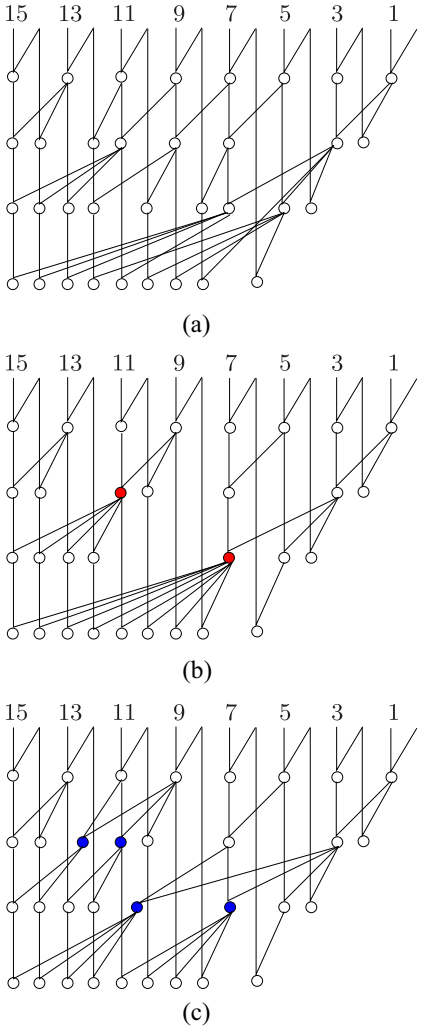

Fig. 6. Cloning reduces size by 1 with same mfo constraint for $n = 16$ and $L = 4$. (a) Size = 34 and mfo = 4. (b) Size = 31 and mfo = 8. (c) With prefix node cloning: Size = 33 and mfo = 4.

Fig. 6 illustrates the operation of prefix node cloning in $n = 16$ bit prefix graph with $L = 4$. The prefix graphs with mfo = 4 and mfo = 8 are shown in Fig. 6(a) and (b), respectively. The sizes of the prefix graphs are 34 and 31, respectively. Now in Fig. 6(b), there are only two nodes which have fan-out higher than 4 (11:8 and 7:0), which are marked. If we do clone

Algorithm 5 Structure Aware Prefix Node Cloning on G_{uncloned}

```
1: for node ∈ prefix nodes from higher index and level to
   lower index and level in G_uncloned do
2:   if fanout(node) > mfo then
3:     numPartition = ⌈fanout(node)/mfo⌉;
4:     (P_1, P_2, ....P_numPartition) = partition
       (fanoutNodes(node));
5:     numClonedNodes = numPartition − 1;
6:     (c_1, c_2, ... c_numClonedNodes) = clone(node);
7:     for i ∈ 1 to numClonedNodes do
8:       disconnect node to prefix nodes in P_i;
9:       connect c_i to prefix nodes in P_i;
10:    end for
11:  end if
12: end for
```

these two nodes and perform the reconnections as shown in Fig. 6(c), mfo is reduced from 8 to 4, the size becomes 33, which is still one less than the prefix graph with mfo = 4 without cloning. The benefit of this prefix node cloning is two-fold. First, the size of the prefix graph with cloning is less than that without cloning for same n, L, and mfo. It should be stressed that although the improvement in s is very small for the illustrated case, the prefix graph size reduction becomes higher as n increases. Second, it can help to reduce wire-length cost for a regular structured placement. For instance, 11:8 is driving 11:0, 13:8, 14:8, and 15:8 in Fig. 6(a) which are more distributed than the case when 11:8 is driving 14:8, 15:8 and its cloned node is driving 13:8, 12:8, and 11:0 in Fig. 6(c). As a result, the interconnect delay is expected to be smaller in case of cloning.

Algorithm 5 presents the steps of structure aware prefix node cloning on a prefix graph generated by Algorithms 1–4. We call this graph as G_{uncloned} hereafter on which Algorithm 5 is exercised. The prefix nodes in G_{uncloned} are traversed from higher to lower index, and for the same index more precedence is given to higher level. For instance, the traversal order in Fig. 6(b) is 15:0, 15:8, 15:12, 15:14, 14:0, 14:8. . . and so on. For each node in that order, the fan-out count is checked (Line 2). If it is greater than mfo, fan-out nodes of node are partitioned in order of decreasing bit-index. The number of partitions (numPartition) is $\lceil$(fanout(node)/mfo)$\rceil$, so that after partition, the number of nodes in the partition is $\leq$ mfo. So (numPartition-1) prefix nodes are cloned and the nodes in each partition, except the last one, are connected to each of the cloned nodes after disconnecting them from node (Lines 7–10). The connections for the prefix nodes in the last partition are untouched. Also, the partition with nodes which are closer to node are treated as the last partition. For instance, in Fig. 6(c), the fan-out nodes of 7:0 are partitioned into $P_1 =$ (15:0, 14:0, 13:0, 12:0) and $P_2 =$ (11:0, 10:0, 9:0, 8:0). The nodes in P_1 are connected to the cloned node, whereas the nodes in P_2 are kept connected to the actual node 7:0. Please note that a partition in random order of bit-index such as (15:0, 13:0, 10:0, 9:0) and (14:0, 12:0, 11:0, 8:0) might not be a good option, as that would increase the wire-length cost.

It should be emphasized that the closeness to a prefix node or distance between two prefix nodes is not well-defined as the prefix graph generation is much before actual placement, and even before the gate-level synthesis. However, the closeness here refers to the pictorial closeness which can map well with a structured regular placement. In Section V-B, we have experimentally demonstrated how the structure aware ordered partition can help in improving the solution quality in comparison to random partition which does not take into account the structure of the prefix graph.

When a node is cloned, the fan-out count of the fan-in nodes increases. For instance, in Fig. 6(c), when the node 7:0 is cloned, the fan-out counts of 3:0 and 7:4 increase from 3 to 4 and 1 to 2, respectively. So it may be possible that some prefix nodes have fan-out lesser than or equal to mfo initially before the cloning, but the fan-out count may go beyond mfo after cloning. However, traversal order of the prefix node ensures that once any prefix node is traversed and the fan-out count is reduced to mfo, if required, the fan-out count of that node can not increase. For instance, once we visit the node 7:0, after that the fan-out count of 7:0 can not increase since the nodes visited after 7:0 are the nodes either with indices lower than 7 or with index 7 but lower level. Note that at the end of this algorithm, the fan-out count of some input nodes may go beyond mfo, particularly when mfo = 2.

Lemma 5: Complexity of Algorithm 5 is $O(n^2 \log_2 n)$.

Proof: Size of a prefix graph or the total number of prefix nodes in a prefix graph with n bit and $\log_2 n$ logic level is upper bounded by $n \log_2 n$. This is because, maximum number of prefix nodes with MSB equal to a particular bit-index $= \log_2 n$, otherwise the level of the prefix graph would exceed $\log_2 n$. So the for loop in Algorithm 5 will run at most $n \log_2 n$ times. If fc be the fan-out count for any node, then the complexity of Lines 3-10 would be O(fc), as there will be O(fc) reconnections (Lines 7–10) and Lines 3–6 would be also O(fc). Since fc $\leq n$ always, the operations inside the for loop will be at most $O(n)$. Therefore, the complexity of Algorithm 5 is $O(n \times n \log_2 n) = O(n^2 \log_2 n)$. ■

Theorem 1: Our approach of generating an n bit parallel prefix graph with bounded fan-out mfo and logic level restriction $\log_2 n$ is a polynomial algorithm in n, viz. $O(n^2 \log_2 n)$.

Proof: It directly follows from Lemmas 1 and 3–5. ■

V. Experimental Results

We have implemented our approach in C++ and executed on a Linux machine with 72GB RAM and 2.8GHz CPU. We compare our approach at the logic synthesis stage with the most recent algorithmic adder synthesis approach [16], and after placement/routing with regular adders, [1], [16], DC adder and custom adders.

A. Comparison at Logic-Synthesis Level

Table I compares our approach with [16] for 32, 64, 96, and 128 bit adders in terms of the size (which is unit less) of the prefix graph under different mfo and bit-wise/fixed level restriction. Please note that this comparison does not include the cloning (Algorithm 5). For $n = 32$, Roy *et al.* [16] can

TABLE I
COMPARISON WITH [16] IN TERMS OF THE SIZE OF THE PREFIX GRAPHS

n	MFO	Our Approach		Approach in [16]	
		Bit-wise	Fixed	Bit-wise	Fixed
32	2	114	114	114	-
	4	92	90	92	-
	6	86	81	86	81
	8	83	78	83	78
	12	81	76	81	76
	16	79	74	79	74
64	2	290	290	290	-
	4	227	219	252	-
	6	214	197	238	-
	8	207	192	-	-
	10	202	184	-	-
	12	198	180	-	-
	16	194	178	192	-
	32	185	169	185	167
96	2	417	417	450	-
	4	337	295	-	-
	6	317	261	-	-
	8	307	258	-	-
	16	289	242	-	-
	32	278	235	278	-
128	2	706	706	706	-
	4	536	512	-	-
	6	507	462	-	-
	8	488	447	-	-
	16	455	413	-	-
	32	433	390	-	-
	64	416	373	416	364

generate solutions under different mfo (except for mfo $= 2, 4$ under fixed level restriction), and our approach can also provide the same solution quality. However, as n increases, Roy *et al.* [16] failed to give solutions in most of the cases. Apart from providing solutions in all cases, the most important advantage of our algorithm is its fast run-time (0.02 s for $n = 64$ and 0.08 s for $n = 128$) due to its polynomial-time complexity in n.

Roy *et al.* [16] proposes a comprehensive pruning based exhaustive and exponential time algorithm, which scales well without any fan-out restriction by setting the pruning parameter $\Delta = 3$ till 128 bit adders and the run-time for 128 bit adder is 25 s. But Δ needs to be increased for getting solutions with fan-out restriction as discussed in [16] and consequently it becomes intractable. Although it can generate solutions for bit-wise level restrictions in a few cases, it fails to provide solutions in other cases such as for mfo $= 8, 10, 12$ (for $n = 64$). The reason is setting Δ beyond 6/7 becomes infeasible even with 72 GB RAM due to the generation of millions of solutions at intermediate bit-widths. For fixed level restriction, this intractability is more severe, and it can not provide solutions as n goes beyond 32 (Note that mfo $= 32/64$ for $n = 64/128$ is equivalent to no fan-out restriction).

However, it is counter-intuitive that it could get the solutions for more stringent fan-out restriction, such as mfo $= 2$. This is because even with setting $\Delta = 30$, the number of intermediate solutions do not go beyond 100k. In [16], additional pruning, such as storing bounded number of solutions at each bit-width, helps to achieve solutions for mfo $= 4, 6$ ($n = 64$). But it costs in compromising the solution quality, in addition to its high memory-overhead (around 1.8 GB) and run-time (mentioned in Table V).

TABLE II
SIZE IMPROVEMENT BY PREFIX NODE CLONING ON 64 BIT PREFIX GRAPHS WITH DIFFERENT mfo

mfo	Size without cloning		mfo of $G_{uncloned}$	Size after cloning	
	Bit-wise	Fixed		Bit-wise	Fixed
2	290	290	4	301	281
			6	284	254
			8	290	260
			12	281	253
			16	281	250
			32	275	239
4	227	219	8	223	204
			12	220	195
			16	220	198
			32	215	191
6	214	197	12	207	186
			16	205	188
			32	199	181
8	207	192	16	202	183
			32	195	176

TABLE III
SIZE IMPROVEMENT BY PREFIX NODE CLONING ON 128 BIT PREFIX GRAPHS WITH DIFFERENT mfo

mfo	Size without cloning		mfo of $G_{uncloned}$	Size after cloning	
	Bit-wise	Fixed		Bit-wise	Fixed
2	706	706	4	703	645
			6	657	589
			8	663	583
			12	660	576
			16	642	559
			32	622	540
			64	616	529
4	536	512	8	526	472
			12	521	463
			16	514	457
			32	498	439
			64	485	423
6	507	462	12	490	440
			16	481	432
			32	467	417
			64	451	358
8	488	447	16	471	424
			32	455	406
			64	442	392

Next, we present the impact of cloning on the prefix graphs obtained by Algorithms 1–4 in Tables II and III, respectively, for 64 and 128 bit adders. In each table, Columns 2 and 3, respectively, show the size of the prefix graphs for bitwise and fixed level restriction with the fan-out restriction of mfo. Now, for cloning we apply Algorithm 5 on a prefix graph with higher mfo. For instance, we can apply Algorithm 5 on a prefix graph with mfo $= 8, 12, 16, 32$, etc., and finally generate the prefix graphs with mfo $= 4$. Column 4 presents the mfo of the prefix graph G_{uncloned} which acts as the input to Algorithm 5. Columns 5 and 6, respectively, show the size of the resultant prefix graph after cloning with the desired mfo (in Column 1). For instance, with bitwise level restriction, mfo $= 4$ and 64 bit adders, the size of the prefix graph without cloning is 227, whereas after cloning we can get prefix graphs with size 223, 220, 220, and 215, respectively, when the mfos of G_{uncloned}, input to Algorithm 5, are 8, 12, 16, and 32.

In general, the trend of prefix node cloning is that more is the mfo of G_{uncloned}, more is the total savings in size. This is because, the total size after cloning is the sum of size

TABLE IV
COMPARISON BETWEEN WITH AND WITHOUT PREFIX NODE CLONING FOR 128 BIT ADDERS

mfo	Delay (ps)			Area (μm^2)			Power (mW)		
	Without cloning	With Cloning	Imprv.	Without cloning	With Cloning	Imprv.	Without cloning	With cloning	Imprv.
2	403.7	400.7	0.7%	5467.8	4853.0	11.2%	17.1	14.6	14.6%
4	410.7	403.8	1.7%	4346.6	3777.0	13.1%	13.4	11.8	11.9%
6	411.3	405.4	1.4%	4056.6	3751.7	7.5%	12.6	11.8	6.3%
8	414.7	405.0	2.3%	3968.3	3616.8	8.9%	12.4	11.5	7.3%

of G_{uncloned} before cloning and the number of cloned nodes ($\text{cnt}_{\text{cloned}}$). For instance, consider the case mfo $= 4$, $n = 128$ and fixed level restriction. Algorithms 1–4 could get a size of 512. If the mfo of G_{uncloned} is 8, then size(G_{uncloned}) $= 447$ and $\text{cnt}_{\text{cloned}} = 25$, totaling $447 + 25 = 472$. But if the size of mfo of G_{uncloned} is changed from 8 to 16, size(G_{uncloned}) decreases to 413, and $\text{cnt}_{\text{cloned}}$ increases from 25 to 44 totaling $413 + 44 = 457$. This is intuitive that when mfo of G_{uncloned} increases, we need to clone more to achieve our desired mfo constraint. But since size(G_{uncloned}) typically decrease faster with relaxing (or increasing) mfo of G_{uncloned}, we get overall savings in size with increasing mfo of G_{uncloned}. However, there are some exceptions where $\text{cnt}_{\text{count}}$ increase more than the decrease in size(G_{uncloned}) causing lesser savings with increase in mfo of G_{uncloned}. For instance, for $n = 64$, mfo $= 6$ and fixed level restriction, size(G_{uncloned}) reduces from 180 to 178 when mfo of G_{uncloned} is changed from 12 to 16, but $\text{cnt}_{\text{cloned}}$ increase from 6 to 10 causing overall increase in size from 186 to 188.

B. Comparison After Placement/Routing

The adder architectures provided by our approach are synthesized in Synopsys DC (version H-2013.03-SP5), functionally verified by Verilog Compiler Simulator, and placed, routed and timed by IC Compiler (ICC) to compare with other approaches ([1], [16], Kogge–Stone, Sklansky, etc.) and behavioral adder implementation ($Y = A+B$) by DC. The behavioral adder implementation of DC generates modified Sklansky structure [19] providing delay almost close to Kogge–Stone at much lower area/power. For all reported DC/ICC results, "compile_ultra" command is used for adder synthesis. "tt1p05v125c" corner in 32 nm SAED cell-library [24] (available through Synopsys University Program) has been used for technology-mapping. FO4 delay in this corner is 36 ps and area of a unit-sized inverter is 1.27 μm^2. The target delay specified for 64 and 128 bit adders are, respectively, 100 and 200 ps, the operating frequency is 1 GHz and the activities at the primary input are 0.1.

First, we demonstrate the benefit of prefix node cloning in our approach. Table IV compares our solutions with and without prefix node cloning for 128 bit adders. Column 1 shows the mfo of the prefix graph. The multicolumns 2–4, 5–7, and 8–10 compare, respectively, the delay, area, and power between them. As illustrated in Table III, there can exist several prefix graph solution in our approach with prefix node cloning for same mfo. In general, we obtain similar or better delay in comparison with the uncloned version with improvements in area/power. For comparison in Table IV, we pick up the solution with the best delay. The mfos of G_{uncloned} in Table IV are not the same for all the entries, and are, respectively, 4, 64, 32, and 64. Note that the solution with the best delay does not typically have the best area/power numbers, so we have some solutions which are minutely worse compared to the reported delay, but they have better area/power than the reported ones.

We can see that the maximum improvement in power could be upto 14.6% with cloning and same mfo restrictions, with slightly better delay. In general, the power/area improvement is higher for lower mfo, because the percentage reduction in size is higher for lower mfo (Tables II and III). The improvement in delay is not significant, since we have run DC/ICC with an aggressive target delay, and those tools will try the best to get the fastest circuit implementation. However, it would need more aggressive post-synthesis optimizations, such as gate-sizing and buffering etc. which cost more area/power. So we have obtained significant improvement in area/power.

In order to demonstrate the benefit of the structure awareness during the cloning mechanism, we take an example solution of 128 bit adder with mfo $= 8$. In the first case, we apply our regular structure-aware cloning mechanism to reduce its mfo to 4. In the second case, instead of doing an ordered partition as in our cloning mechanism, we perform a random partition. Then we pass these two solutions through DC/ICC, and observe that the area/power numbers are very similar for the two solutions. But the first solution has better (2%) critical path delay than the second solution (with random partition during cloning). Similar observations have been made with 64 bit adders as well. So although the two solutions are similar in terms of logic-level, fan-out, and size, the prefix graph structure aware cloning solution with the ordered partition of the nodes achieves better delay. This can be illustrated as the placement/routing tools are capable of exploiting the structure-aware prefix node cloning, and placing the actual logic gates accordingly.

Next, we compare our approach with [16] in Table V for $n = 64$. Column 1 denotes the mfo, and the multicolumns 2–4, 5–7, 8–10, and 11–13, respectively, present the delay, area, power, and run-time for both approaches. In terms of run-time, our approach is much faster (800×–12 050×) than [16] due to its polynomial time complexity. In case of mfo $= 2$, our approach excels [16] in performance by 4.5%, area by 12.8% and power by 15.4%. For mfo $= 4$, our approach improves performance by 1.5% with, respectively, 17.2% and 15.3% improvement in area and power over [16]. The general trend is that as mfo increases more, the comparative performance/area/power benefit of our approach decreases. In case of mfo $= 16$, we got minor degradation (1.7%) in performance with 6%–7% improvement in area/power.

Typically, adders are synthesized with a target delay constraint. Since our algorithm works at the technology-independent stage, it is difficult for the algorithm

TABLE V
COMPARISON WITH [16] FOR 64 BIT ADDERS

mfo	Delay (ps)			Area (μm^2)			Power (mW)			Run-time (sec)		
	[16]	Our	Imprv.	[16]	Our	Imprv.	[16]	Our	Imprv.	[16]	Our	Imprv.
2	348.3	332.8	4.5%	2387.6	2081.9	12.8%	7.71	6.52	15.4%	16	0.02	800X
4	345.3	340.0	1.5%	2132.9	1765.6	17.2%	6.68	5.66	15.3%	241	0.02	12050X
6	355.0	350.2	1.4%	2058.2	1752.9	14.8%	6.49	5.68	12.5%	212	0.02	10600X
16	355.0	360.9	-1.7%	1835.0	1709.1	6.9%	5.89	5.53	6.1%	149	0.02	7450X

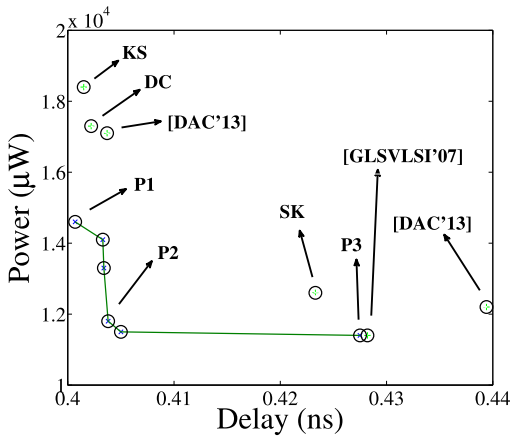

Fig. 7. Delay vs. power plot for 128 bit adders.

to adapt to the target timing constraint which is relevant after technology-mapping/placement/routing. However, under a certain timing constraint, the solutions provided by our approach can search for pareto-optimal points for delay vs. power. Fig. 7 shows such delay vs. power pareto-front for 128 bit adder solutions provided by our approach. For comparing with [16], we plot two solutions of [16] with, respectively, best performance and best power number. We can see that the solutions from other approaches are on the right and/or above this pareto curve. P_1 provides better solution than Kogge–Stone, behavioral DC adder and the best performance solution of [16], P_2 provides better solution than Sklansky and the best power solution of [16], and P_3 provides better solution than [1].

Table VI compares our approach with other approaches for 128 bit adders. Our approach (solution P_1) improves over Kogge–Stone adder in area by 16.4% and power by 20.7%, excels behavioral DC adder in area by 10.2% and power by 15.6%, and improves over [16] (best performance version) in area by 11.2% and power by 14.6% with slight improvement in performance over all the approaches. Our another solution (P_2) excels Sklansky adder in performance by 4.6%, area by 7.8% and power by 6.3%. It also improves over [16] (with best power number) in performance by 8.1%, area by 3.2% and power by 3.3%. Our best power solution (P_3) achieves same power in comparison to [1] with a slight improvement in area and performance. Note that the points P_1, P_2, and P_3 are the same solution points in the delay vs. power pareto-optimal curve in Fig. 7.

C. Comparison With Custom Adders

The designers come up with detailed gate-level verilog/VHDL netlist to build custom adders. This takes a lot

TABLE VI
COMPARISON WITH OTHER APPROACHES FOR 128 BIT ADDERS

Method	Delay (ps)	Area (μm^2)	Power (mW)
Kogge-Stone	401.5	5803.3	18.4
[16] (best perf.)	403.7	5467.8	17.1
Behav. DC	402.2	5405.4	17.3
Our (P_1)	**400.7**	**4853.0**	**14.6**
Sklansky	423.3	4094.8	12.6
[16] (best power)	439.4	3902.7	12.2
Our (P_2)	**403.8**	**3777.3**	**11.8**
[1]	428.2	3626.3	11.4
Our (P_3)	**427.5**	**3618.2**	**11.4**

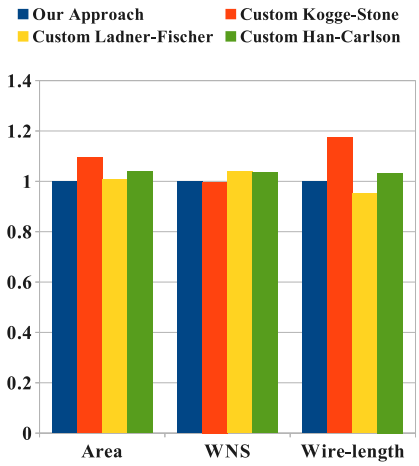

Fig. 8. Comparison with 64 bit custom adder blocks.

of engineering effort but achieves good performance/area trade-off for target technology node. In order to compare with such 64 bit custom adders implemented in an industrial high-performance design and targeting a cutting-edge technology node (CMOS SOI 22 nm), we have integrated our algorithm to an industrial placement driven synthesis [25] tool. Fig. 8 compares our approach with 64 bit custom adder blocks after placement in terms of area, worst negative slack and wire-length. Our approach improves area by 9.4% and wire-length by 17.5% over custom Kogge–Stone adder with same performance, improves area by 3.8%, performance by 2.1% and wire-length by 3.3% over custom Han–Carlson adder and improves area by 1%, performance by 2.5% over custom Ladner–Fischer adder with 4% overhead in wire-length. Note that the performance improvement has been calculated based on the actual critical path delay of the adders.

VI. CONCLUSION

In this paper, a novel polynomial-time algorithm is presented to synthesize n bit parallel prefix adder structures with

the objective of minimizing the size of the prefix graph for $\log_2 n$ level and any fan-out constraint. A post-processing cloning mechanism is also proposed which further optimizes the size maintaining the same constraints, and takes into account the structure of the prefix graph. This makes the prefix graph more friendly to placement/routing tools, which has been experimentally demonstrated. The design space exploration by our algorithm has provided high-performance adders which are more power/performance/area efficient than regular adders, industry-standard DC adder and adders generated by the state-of-the-art adder synthesis algorithms [1], [16]. It even beats 64 bit custom designed adders targeting 22 nm technology library and implemented in industrial designs. Furthermore, since our algorithm is highly scalable, it can be integrated into any commercial logic synthesis tool to synthesize designs containing thousands of adders and could provide the flexibility of performance-area/power trade-off in industrial designs. Currently, our algorithm focuses on adders of $\log_2 n$ logic level to target high-performance designs, but in future we plan to extend it for relaxed logic levels to achieve more area/power efficient solutions.

REFERENCES

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

[2] N. H. Weste and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*. Boston, MA, USA: Addison-Wesley, 2004.

[3] 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.

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

[5] 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.

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

[7] T. Han and D. Carlson, "Fast area-efficient VLSI adders," in *Proc. 8th Symp. Comput. Arithm.*, Como, Italy, May 1987, pp. 49–56.

[8] S. Knowles, "A family of adders," in *Proc. 15th IEEE Symp. Comput. Arithm.*, Vail, CO, USA, Jun. 2001, pp. 277–281.

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

[10] 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. Design Autom. Conf.*, Yokohama, Japan, Jan. 2007, pp. 609–615.

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

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

[13] 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. Design Autom. Conf.*, Orlando, FL, USA, Jun. 1990, pp. 361–364.

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

[15] Z. Huang and M. D. Ercegovac, "Effect of wire delay on the design of prefix adders in deep-submicron technology," in *Proc. ASILOMAR Conf. Signals Syst. Comput.*, Pacific Grove, CA, USA, Oct./Nov. 2000, pp. 1713–1717.

[16] 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. Design Autom. Conf.*, Austin, TX, USA, May/Jun. 2013, pp. 1–8.

[17] M. Choudhury, R. Puri, S. Roy, and S. C. Sundararajan, "Automated synthesis of high-performance two operand binary parallel prefix adder," U.S. Patent 8 683 398, Mar. 25, 2014.

[18] B. R. Zeydel, T. T. J. H. Kluter, and V. G. Oklobdzija, "Efficient mapping of addition recurrence algorithms in CMOS," in *Proc. IEEE Symp. Comput. Arithm.*, Cape Cod, MA, USA, Jun. 2005, pp. 107–113.

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

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

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

[22] 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.

[23] S. Roy, M. Choudhury, R. Puri, and D. Z. Pan, "Polynomial time algorithm for area and power efficient adder synthesis in high-performance designs," in *Proc. Asia South Pac. Design Autom. Conf.*, Chiba, Japan, Jan. 2015, pp. 249–254.

[24] *Synopsys 32/28nm Generic Library for Teaching IC Design*. [Online]. Available: http://www.synopsys.com/Community/UniversityProgram/Pages/32-28nm-generic-library.aspx, accessed Mar. 14, 2014.

[25] H. Ren, D. Z. Pan, and D. S. Kung, "Sensitivity guided net weighting for placement driven synthesis," in *Proc. Int. Conf. Phys. Design*, Phoenix, AZ, USA, Apr. 2004, pp. 10–17.

Subhendu Roy (S'13) received the B.E. degree in electronics and telecommunication engineering from Jadavpur University, Kolkata, India, in 2006, the M.Tech. degree in electronic systems from the Indian Institute of Technology Bombay, Mumbai, India, in 2009, and the Ph.D. degree in electrical and computer engineering from the University of Texas at Austin, Austin, TX, USA, in 2015.

He is currently a Principal Software Engineer with Cadence Design Systems, San Jose, CA, USA. He has three years of full-time industry experience in an Electronic Design Automation (EDA) Company, Atrenta, San Jose, CA, USA, where he was involved in developing tools in the architectural power domain and register transfer level domain. He did internships with IBM Thomas J. Watson Research Center, Yorktown Heights, NY, USA, in 2012 and Mentor Graphics, Wilsonville, OR, USA, in 2013 and 2014. He holds one patent and has first-authored papers in major EDA conferences/journals, such as Design Automation Conference (DAC), International Symposium on Physical Design (ISPD), Asia and South Pacific Design Automation Conference (ASP-DAC), and IEEE TRANSACTIONS ON COMPUTER AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS. His current research interests include design automation for logic synthesis, physical design, and cross-layer reliability.

Mr. Roy was a recipient of the Best Paper Award at ISPD 2014.

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, in 2005, and the M.S. and Ph.D. degrees in computer engineering from Rice University, Houston, TX, USA, in 2008 and 2011, respectively.

He is a Research Staff Member with the IBM Thomas 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 an IBM Fellow with IBM Thomas J. Watson Research Center, Yorktown Heights, NY, USA, where he leads software–hardware acceleration research for IBMs Analytics and Cognitive Solutions. Most recently, he led the design methodology innovations for IBMs latest Power7 and zEnterprise microprocessors and is currently leading Design Methodology Research Efforts on Future Processors. He has been an Adjunct Professor with the Department of Electrical Engineering, Columbia University, New York, NY, USA. He was the John Von-Neumann Chair with the Institute of Discrete Mathematics, Bonn University, Bonn, Germany. 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. He is an inventor of over 50 U.S. patents (both issued and pending) and have authored over 120 publications on the design and synthesis of low-power and high-performance circuits.

Dr. Puri was a recipient of the Best of IBM awards in 2011 and 2012, the IBM Corporate Award from IBMs CEO, the 2014 Asian American Engineer of the Year Award, the SRC Outstanding Mentor Award, several IBM Outstanding Technical Achievement awards, and numerous accolades including the highest technical position with IBM, the IBM Fellow, which was awarded for his transformational role in microprocessor design methodology. He was an Editor of the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS. He is a member of the IBM Academy of Technology, the IBM Master Inventor, the ACM Distinguished Speaker, and the IEEE Distinguished Lecturer.

David Z. Pan (S'97–M'00–SM'06–F'14) received the B.S. degree in Physics/Geophysics from Peking University, Beijing, China, in 1992, and the M.S. and Ph.D. degrees in computer science from the University of California, Los Angeles (ULCA), Los Angeles, CA, USA, in 1998 and 2000, respectively.

From 2000 to 2003, he was a Research Staff Member with IBM Thomas J. Watson Research Center, Yorktown Heights, NY, USA. He is currently the Engineering Foundation Professor with the Department of Electrical and Computer Engineering, University of Texas at Austin, Austin, TX, USA. He has published over 220 papers in refereed journals and conferences, and holds eight U.S. patents. His current research interests include cross-layer nanometer IC design for manufacturability/reliability, new frontiers of physical design, and Computer-Aided Design (CAD) for emerging technologies such as 3-D-IC, biochip, and nanophotonics.

Prof. Pan was a recipient of a number of awards for his research contributions and professional services, including the SRC 2013 Technical Excellence Award, the DAC Top 10 Author in Fifth Decade, the DAC Prolific Author Award, the ASP-DAC Frequently Cited Author Award, 11 Best Paper Awards in ISPD 2014, International Conference on Computer-Aided Design (ICCAD) 2013, ASP-DAC 2012, ISPD 2011, the IBM Research 2010 Pat Goldberg Memorial Best Paper Award, ASP-DAC 2010, DATE 2009, International Conference on IC Design and Technology 2009, and SRC Techcon in 1998, 2007, and 2012, the Communications of the ACM Research Highlights in 2014, the ACM/SIGDA Outstanding New Faculty Award in 2005, the National Science Foundation CAREER Award in 2007, the SRC Inventor Recognition Award thrice, the IBM Faculty Award four times, the UCLA Engineering Distinguished Young Alumnus Award in 2009, the UT Austin RAISE Faculty Excellence Award in 2014, the ISPD Routing Contest Awards in 2007, the eASIC Placement Contest Grand Prize in 2009, the ICCAD CAD Contest Awards in 2012 and 2013. He has served as the Senior Associate Editor for the *ACM Transactions on Design Automation of Electronic Systems*. He was 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: REGULAR PAPERS, the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS—PART II: EXPRESS BRIEFS, the SCIENCE CHINA INFORMATION SCIENCES, the *Journal of Computer Science and Technology*, the IEEE Society Newsletter. He has served in the executive/program committees of many major conferences, including DAC, ICCAD, ASP-DAC, and ISPD.