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

Subhendu Roy[‡], Mihir Choudhury[†], Ruchir Puri[†], David Z. Pan[‡]
[‡]Department of Electrical and Computer Engineering, University of Texas at Austin, USA
[†]IBM T. J. Watson Research Center, Yorktown Heights, USA
subhendu@utexas.edu, {choudhury,ruchir}@us.ibm.com, dpan@ece.utexas.edu

ABSTRACT

Adders are the most fundamental arithmetic units, and often on the timing critical paths of microprocessors. Among various adder configurations, parallel prefix structures provide the high performance adders 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_2n logic level and any arbitrary fan-out restriction. The design space exploration by our algorithm 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 (i) excels highly competitive industry standard Synopsys Design Compiler adder (128 bit) in performance (2%), area (25%) and power (13.3%) in 32nm technology node, and (ii) improves performance/area over even 64 bit custom designed adders targeting 22nm technology library and implemented in an industrial high-performance design.

1. 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 adhoc adder structures such as the carry-skip adder, the carry select adder and the carry-lookahead adder have been proposed in the past [1]. 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 [1] like Kogge-Stone [2] (minimal logic level and fan-out), Sklansky [3] (minimal logic level and wire-tracks) and Brent-Kung [4] (minimal fan-out and wire-tracks). In addition to these structures, Ladner-Fischer [5], Han-Carlson [6] and Knowles [7] 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 [8][9] but need a significant engineering effort and not suitable for today's aggressive turn-around-time requirement.

On the contrary, an algorithmic synthesis approach is more flexible to Engineering Change Orders (ECOs), 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 [10][11][12][13] have been done to target this problem. Most of them achieve the theoretical bound for s for $L \geq 2log_2n - 2$, given by Snir [14], but yield sub-optimal result when L is reduced to log_2n 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. 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_2n 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 $L = log_2n$ until a very recent work [16], where a comprehensive pruning based algorithm, exercised on exhaustive bottom-up enumeration, is presented to explore several parallel prefix structures at a time. However, there are certain limitations in this work, (i) 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 $n \geq 64$. (ii) 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_2n with any maximum fan-out restriction mfo. This is performed by first constructing a graph computing outputs for odd bit-indices with fan-out restriction of $\lfloor \frac{mfo}{2} \rfloor$ 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 sub-problems, our algorithm can still achieve the same solution quality with the highly runtime/memory intensive approach [16] for adders of lower bit-width ($n \leq 32$). For higher bit-width, such as $n \geq 64$, [16] fails to provide solutions in most cases, whereas our algorithm generates solution for any n. Our main contributions are summarized as follows:

- To the best of our knowledge, this is the first work to synthesize prefix adders of bit-width n with logic

978-1-4799-7792-5/15/$31.00 ©2015 IEEE

level $log_2 n$ under any arbitrary fan-out restriction in polynomial time.

- The design space exploration by our algorithm has provided adders which excel in timing, area/power over highly competitive Design Compiler adder and fast regular adders, such as Sklansky and Kogge-Stone.

- Our approach even beats 64 bit custom designed adders implemented in an industrial high-performance design. It also improves in area/power (and timing for higher bit adders) 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 3. Section 4 describes our algorithm to synthesize an n bit adder with $log_2 n$ level and arbitrary fan-out restriction. Finally, Section 5 presents the experimental results at both logic-synthesis level and after placement/routing followed by conclusion in Section 6.

2. 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 $C_{out} = c_{n-1}$, where $s_i = a_i \oplus b_i \oplus c_{i-1}$ and $c_i = a_ib_i + a_ic_{i-1} + b_ic_{i-1}$.

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 [17]:

- Pre-processing: Bitwise generation of g, p

$$g_i = a_i.b_i \text{ and } p_i = a_i \oplus b_i \quad (1)$$

- **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

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

$$G_{[i:j]} = \begin{cases} g_i & \text{if } i = j \\ G_{[i:k]} + P_{[i:k]}.G_{[k-1:j]} & \textit{otherwise} \end{cases} \quad (2)$$

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

$$\begin{aligned}(G,P)_{[i:j]} &= (G,P)_{[i:k]} \; o \; (G,P)_{[k-1:j]} \quad (3)\\ &= (G_{[i:k]} + P_{[i:k]}.G_{[k-1:j]}, P_{[i:k]}.P_{[k-1:j]})\end{aligned}$$

- Post-processing: Sum/Carry-out generation

$$s_i = p_i \oplus c_{i-1}, \quad c_i = G_{[i:0]} \text{ and } C_{out} = c_{n-1} \quad (4)$$

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 8 bit and we can see that $C_{out} = c_7 = y_7$ is given by

$$y_7 = ((x_7 \; o \; x_6) \; o \; (x_5 \; o \; x_4)) \; o \; ((x_3 \; o \; x_2) \; o \; (x_1 \; o \; x_0)) \quad (5)$$

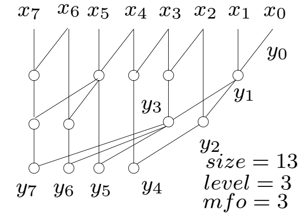

Figure 1: An example 8 bit prefix graph

3. 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 maximum fan-out 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 maximum fan-out 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 maximum fan-out (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 2 ways, - (a) the maximum level for each output bit-index m is $\lceil log_2 n \rceil$, which can be termed as *fixed level restriction* and (b) 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*.

4. OUR APPROACH

A prefix graph of bit-width n computes output bits for bit-indices 1 to $n-1$. An n bit prefix graph will have $\lfloor \frac{n}{2} \rfloor$ odd bit-indices, *i.e.*, 1, 3, ... $(2 \times \lfloor \frac{n}{2} \rfloor - 1)$ and $\lceil \frac{n}{2} \rceil$ even bit-indices, *i.e.*, 0, 2, ... $(2 \times \lfloor \frac{n}{2} \rfloor)$. We divide the main problem into 2 sub-problems, (a) Construct a graph (G_{odd}) which computes the outputs for odd bits with fan-out restriction of $\lfloor \frac{mfo}{2} \rfloor$ and (b) Construct the prefix graph G from G_{odd} by computing the even bit outputs with fan-out restriction of mfo. This division of the problem into 2 sub-problems of computing odd and even bit outputs is motivated by the regular adder structures, such as Han-Carlson[6] or Brent-Kung[4], where the computation of odd bit outputs is followed by that of even bit outputs.

4.1 Constructing Output for Odd Bit-indices

We first generate a seed-structure for an n bit prefix graph ($G_{seed}(n)$) computing the odd bit outputs with a fan-out restriction of 2. This is followed by a heuristic which restructures $G_{seed}(n)$ to generate G_{odd} by relaxing the fan-out restriction to $\lfloor \frac{mfo}{2} \rfloor$ (where $\lfloor \frac{mfo}{2} \rfloor > 2$), 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.

Generating Seed Structure: The generation of the seed structure is divided into 2 steps as shown in Algorithm 1. Fig. 2 shows the graph $G_{seed}(16)$, 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, 2 '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 $n-1/n-2$ (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 $msb(N_{x,l}) = x$ and $lsb(N_{x,l}) = x - 2^l + 1$ and $N_{x,l}$ is obtained by combining $N_{x,l-1}$ (trivial fan-in node) and $N_{x-2^{l-1},l-1}$ (non-trivial 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 non-trivial fan-ins of $N_{13,3}$ in Fig. 2.

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

```
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
```

In the second step, we add $\lfloor \frac{n}{2} \rfloor$ prefix nodes in the increasing order of odd-indices to generate the outputs for $\lfloor \frac{n}{2} \rfloor$ 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.

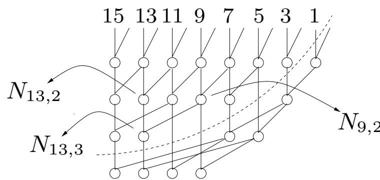

Figure 2: Seed Structure for 16 bit prefix graph

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 non-trivial fanout. This variable implies that any node of bit-index i with higher level than that of $lfn(i)$, having no non-trivial fan-out, is more flexible to be removed in the graph-structure. *If none of the node of bit-index i has non-trivial 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 non-trivial 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 $\lfloor \frac{mfo}{2} \rfloor$ 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".

Algorithm 2 Generating G_{odd} from $G_{seed}(n)$ with $\lfloor \frac{mfo}{2} \rfloor$

```
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
```

The procedure "searchRecursive" is a recursive subroutine with 2 arguments, (a) 'sliceList', the existing list of bit-slices and (b) 'node' the last node in the 'sliceList', except 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 2 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 $G_{seed}(20)$ and suppose we are interested in finding a prefix graph structure of bit-width 20 with fan-out

of 8. We can see that $lfn(19) = N_{19,1}$ and $\lfloor \frac{mfo}{2} \rfloor = 4$. 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 - [17:8 + 7:0], [17:14 + 13:0], [17:14 + 13:6 + 5:0], [17:16 + 15:0], maintaining the restriction in logic level, fan-out and our imposed search-space restriction. The option [17:14 + 13:6 + 5:0] is discarded as the size of it is higher than all other 3 choices. Rest 3 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 2 choices ([17:8 + 7:0], [17:14 + 13:0]) make $lfn(17)$ to be $N_{17,3}$ and $N_{17,2}$ 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.

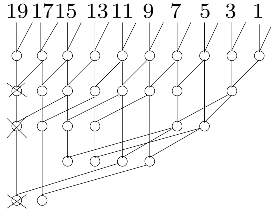

Figure 3: [19 : 18 + 17 : 16 + 15 : 0] **is the choice of bit-slices for bit-index** 19 **in Algorithm 2**

4.2 Constructing Output for Even Bit-indices

The generation of output for even bit-indices consists of 2 stages as described in Algorithm 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 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.

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
```

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 2 nodes, (a) $node1$ derived from $inNode(i)$ and $inNode(i-1)$, and (b) $node2$ 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 2-fold. The first is that it can reduce the number of prefix nodes, if there were more than 2 prefix nodes at that bit-index before 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.

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
```

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.

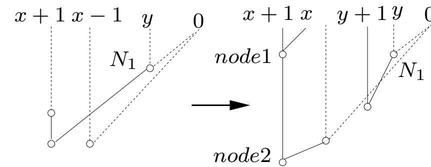

Figure 4: Second transformation facilitating first transformation by reducing fan-out at N_1

The algorithmic complexity of our method is $O(n^2 log_2 n)$ as shown in the following lemmas/theorems. The detailed proofs have been omitted for page limit. However, we have given the proof of Lemma 4.2 which provides the basis of polynomial time-complexity of our algorithm.

LEMMA 4.1. *Complexity of Algorithm 1 is* $O(nlog_2 n)$.

LEMMA 4.2. *'searchRecursive' procedure with a level restriction of p is an* $O(p.2^p)$ *operation.*

PROOF. 'searchRecursive' procedure finds the bit-slices spanning from any bit-index to bit-index 0. For instance, we see in the 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 $T(y+1) \leq T(y) + T(y-1) + T(y-2) + ... + T(1) + O(y)$, with $T(1) = O(1)$. Solving this recurrence relation we get, $T(y) = y.2^y$. Since the maximum value of y can be p, 'searchRecursive' procedure with level restriction p is $O(p.2^p)$. □

COROLLARY 4.3. *With $log_2 n$ level restriction, 'searchRecursive' procedure is an $O(n.log_2 n)$ operation.*

PROOF. This follows from Lemma 4.2 by putting $p = log_2 n$. □

LEMMA 4.4. *The complexity of Algorithm 2 is $O(n^2 log_2 n)$.*

LEMMA 4.5. *The complexities of Algorithm 3 and Algorithm 4 are $O(n^2 log_2 n)$ and $O(n log_2 n)$ respectively.*

THEOREM 4.6. *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. $O(n^2 log_2 n)$.*

5. 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, [16], Design Compiler (DC) adder and custom adders.

5.1 Comparison at logic-synthesis level

We have obtained the binary for [16] from the authors and then compared our approach with [16] in Table 1 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. For $n = 32$, [16] can 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, [16] fails 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 sec for $n = 64$ and 0.08 sec for $n = 128$) due to its polynomial-time complexity in n.

[16] is a comprehensive pruning based exhaustive and exponential time algorithm, but scales well without any fan-out restriction by setting the pruning parameter $\Delta = 3$ till 128 bit adders. But Δ needs to be increased for getting solutions with fan-out restriction as discussed in [16] and consequently it becomes intractable. The reason is setting Δ beyond 6 or 7 becomes infeasible even with 72GB 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 1.8GB) and run-time (mentioned in Table 2).

Table 1: 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

5.2 Comparison after placement/routing

The adder architectures provided by our approach are synthesized in Synopsys DC (version G-2012.06-SP4), functionally verified by VCS, and placed, routed and timed by IC Compiler (ICC) to compare with other approaches ([16], Kogge-Stone, Sklansky etc.) and behavioral adder implementation ($Y = A + B$) by DC. The behavioral adder implementation of DC generates modified Sklansky structure [18] 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** [19] (available through Synopsys University Program) has been used for technology-mapping. FO4 delay in this corner is 36ps and area of a unit-sized inverter is $1.27 \mu m^2$. The target delay specified for 64 and 128 bit adders are respectively 100ps and 200ps, the operating frequency is 1GHz and the activities at the primary input are 0.1. Table 2 compares our approach with [16] for 64 bit adders in terms of delay, power (leakage + dynamic), area and run-time. The solutions for other mfo values, such as 2, 8 etc., are not compared as [16] either generates same solution as ours or could not generate solutions.

Fig. 5 shows the delay vs. power pareto-front for 128 bit adders. P_1 provides better solution than Kogge-Stone and behavioral DC adder, and P_2 provides better solution than Sklansky and [16]. Table 3 compares our solution with best delay with other approaches for 128 bit adders. [16] improves slightly over our best delay solution in area/power, but with a significant delay overhead of 28.4ps. Behavioral

Table 2: 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.
4	356.4	358.8	-0.7%	2209.5	2071.4	6.2%	7.61	7.29	4.2%	241	0.02	12050X
6	357.4	357.4	0%	2073.9	1979.0	4.5%	7.09	7.07	0.3%	212	0.02	10600X
16	393.4	366.3	6.9%	1838.2	1818.7	1.1%	6.39	6.31	1.3%	149	0.02	7450X

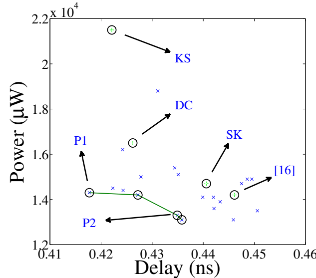

Figure 5: Delay vs. Power plot for 128 bit adders

DC adder achieves delay very close to that of Kogge-Stone at much lower area/power, and our approach improves over this highly competitive behavioral DC adder in performance (2%), area (25%) and power (13.3%).

Table 3: Comparison with other approaches for 128 bit adders

Method	Delay (ps)	Area (μm^2)	Power (mW)
Kogge-Stone	422.1	6279.3	21.5
Sklansky	440.6	4280.8	14.7
[16]	446.1	4091.1	14.2
Behav. DC	426.2	5475.4	16.5
Our	**417.7**	**4108.7**	**14.3**

Comparison with Custom Adders: The designers come up with detailed gate-level verilog/VHDL netlist to build custom adders. This takes a lot 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 22nm), we have integrated our algorithm to an industrial placement driven synthesis [20] tool. Fig. 6 compares our approach with 64 bit custom adder blocks after placement in terms of area, worst negative slack (WNS) 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.

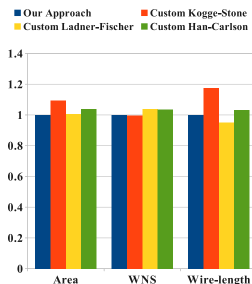

Figure 6: Comparison with 64 bit custom adder blocks

6. CONCLUSION

In this paper, a novel polynomial-time algorithm is presented to synthesize n bit parallel prefix structures with $log_2 n$ level and any fan-out restriction. The design space exploration by our algorithm has provided high-performance adders which are more area/power efficient than regular adders, industry-standard DC adder and adders generated by a highly run-time/memory intensive algorithm. It even beats 64 bit custom designed adders targeting 22nm technology library. 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.

7. REFERENCES

[1] N. H. Weste and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective.* Addison Wesley, 2004.
[2] P. M. Kogge and H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," *IEEE Trans. Computers*, pp. 786–793, 1973.
[3] J. Sklansky, "Conditional sum addition logic," *IRE Trans. on Electronic Computers*, pp. 226–231, 1960.
[4] R. P. Brent and H. T. Kung, "A regular layout for parallel adders," *IEEE Trans. Computers*, pp. 260–264, 1982.
[5] R. E. Ladner and M. J. Fischer, "Parallel prefix computation," *Journal of ACM*, pp. 831–838, 1980.
[6] T. Han and D. Carlson, "Fast area-efficient vlsi adders," *Proc. 8th Symp. Comp. Arithmetic*, pp. 49–56, 1987.
[7] S. Knowles, "A family of adders," *Proc. 15th IEEE Symp. Comp. Arithmetic*, pp. 277–281, 2001.
[8] C. Zhou *et al.*, "64-bit prefix adders: Power-efficient topologies and design solutions," *CICC*, pp. 179–182, 2009.
[9] J. Liu *et al.*, "Optimum prefix adders in a comprehensive area, timing and power design space," *ASPDAC*, 2007.
[10] T. Matsunaga and Y. Matsunaga, "Area minimization algorithm for parallel prefix adders under bitwise delay constraints," *GLSVLSI*, pp. 435–440, 2007.
[11] J. Liu *et al.*, "An algorithmic approach for generic parallel adders," *ICCAD*, pp. 734–740, 2003.
[12] R. Zimmermann, "Non-heuristic optimization and synthesis of parallel prefix adders," *IWLS*, pp. 123–132, 1996.
[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 antything in-between," *DAC*, pp. 361–364, 1990.
[14] M. Snir, "Depth-size trade-offs for parallel prefix computation," *Journal of Algorithms*, pp. 185–201, 1986.
[15] Z. Huang and M. D. Ercegovac, "Effect of wire delay on the design of prefix adders in deep-submicron technology," *ASILOMAR*, pp. 1713–17, 2000.
[16] S. Roy *et al.*, "Towards optimal performance-area trade-off in adders by synthesis of parallel prefix structures," *DAC*, 2013.
[17] B. R. Zeydel, T. T. J. H. Kluter, and V. G. Oklobdzija, "Efficient mapping of addition recurrence algorithms in CMOS," *IEEE Symp. on Computer Arithmetic*, pp. 107–113, 2005.
[18] M. Ketter, D. M. Harris, A. Macrae, R. Glick, M. Ong, and J. Schauer, "Implementation of 32-bit ling and jackson adders," *ASILOMAR*, pp. 170–175, 2011.
[19] `http://www.synopsys.com/Community/UniversityProgram/Pages/32-28nm-generic-library.aspx`. Accessed 14-March-2014.
[20] H. Ren *et al.*, "Sensitivity guided net weighting for placement driven synthesis," in *ISPD*, pp. 10–17, 2004.