# Towards Optimal Performance-area Trade-off in Adders by Synthesis of Parallel Prefix Structures

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

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

## Categories and Subject Descriptors

B.2.m [**Hardware, Arithmetic and Logic Structure**]: Miscellaneous;

## General Terms

Algorithms, Design, Performance

## Keywords

Logic synthesis, Parallel prefix adder, Bottom-up approach

## 1. INTRODUCTION

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

In contrast, automated synthesis approach is productive and flexible to late design changes but traditionally has lagged behind in performance as compared to custom designs. Therefore, the prevalent design approach for high-performance datapath logic continues to be custom design. In the past, several algorithms have been proposed to generate parallel prefix adders targeting minimization of the size of the prefix graph ($s$) under given bit-width ($N$) and logic level ($L$) constraints. Snir [6] has given a theoretical bound of $s$ for $L \geq 2\log_2 N - 2$ with uniform input profile. [7] presents a recursive construction of parallel prefix graphs to obtain a trade-off between size and level, but it could not achieve the bound provided by [6]. Other existing algorithms like a greedy depth-decreasing heuristic [8], dynamic programming based approaches ([9], [10]) or non-heuristic optimization [11] could achieve this bound for some cases but yield a non-optimal result as logic level constraints are reduced (for e.g. to $\log_2 N$) – which is more relevant for high performance adders. The most recent approach [9], that uses dynamic programming (DP) on a restricted search space to generate a seed prefix graph followed by an area-heuristic to further reduce the size of the seed prefix graph, is also the most effective in minimizing the size of the prefix graphs. However, the quality of the area-heuristic solution depends on the selection of seed solution from DP, which is not unique. Furthermore, this algorithm cannot handle fanout/wire-length constraints on nodes in the prefix graph or arrival/required time constraints on individual input/output bits that impact the performance, area, and power consumption of the adder after physical design. In [12], an exhaustive approach is attempted to explore the optimal arithmetic-circuit architectures through selective factorization, but it is very limited in terms of scalability.

To tackle these issues, this paper proposes an efficient al-

*DAC*'13 May 29 - June 07 2013, Austin, Tx, USA
Copyright 2013 ACM 978-1-4503-2071-9/13/05 ...$15.00.

gorithm to generate prefix graphs for synthesizing adders with the best performance-area trade-off. In this approach, prefix graph structures are constructed in bottom-up fashion by exhaustively generating all possible $n+1$ bit prefix graphs from $n$ bit prefix graphs. For scalability to large adders up to 128 bits, our approach proposes a novel compact data structure for manipulating prefix graphs, efficient memory management techniques like lazy copy for storing several prefix graph solutions, and search space reduction strategies like level-restriction, dynamic size pruning, repeatability pruning for targeting prefix graph structures relevant for achieving the best performance-area trade-off. Compared to existing algorithms our approach has the following advantages:

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

The rest of the paper is organized as follows. Section 2 describes binary addition as a prefix graph problem. Section 3 presents our algorithm for generating prefix graph structures. Section 4 presents the results of this approach with a conclusion in section 5.

## 2. PRELIMINARIES

Given an ordered $n$ inputs $x_0$, $x_1$, ..., $x_{n-1}$ (where $x_{n-1}$ is the MSB and $x_0$ is the LSB) and an associative operation $o$, prefix computation of $n$ outputs is defined as follows:

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

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

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

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

Binary addition problem is defined as follows: given $n$ bit augend $A = a_{n-1}....a_1a_0$ and $n$ bit addend $B = b_{n-1}....b_1b_0$, 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 mapped to a prefix computation problem as follows:

- Pre-processing: Bitwise $g$, $p$ generation

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

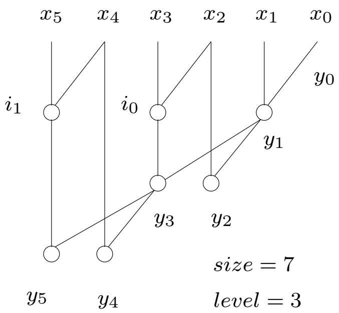


**Figure 1: Prefix Graph Representation**

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

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

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

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

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

- Post-processing: Sum generation

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

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

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

## 3. OUR APPROACH

This section describes a compact data structure for storing and manipulating a prefix graph, efficient memory management strategies for storing several prefix graph solutions, and pruning strategies to scale our approach up to 128 bit adders. Due to the associative nature of the prefix operation $o$, each output bit ($m$) can be constructed by combining the previous input bits 0, 1 ... $m$ in any way keeping their relative orders intact and the number of possible ways is $catalan(m)$, where $catalan(m) = \frac{1}{m+1}\binom{2m}{m}$. Let $G_n$ denotes the set of all possible prefix graphs with bit-width $n$. Then size of $G_n$ grows exponentially with $n$ and is given by $catalan(n-1) * catalan(n-2) * .... catalan(0)$. For example, $|G_8| = 332972640$, $|G_{12}| = 2.29 * 10^{24}$. As the search space is huge, we require compact data structure, efficient memory management and search space reduction techniques to scale this approach.

## 3.1 Compact Notation and Data Structure

We represent the prefix graph by a sequence of indices. Each prefix node is represented by an index, which is the most significant bit (MSB) of the node. Fig.2 illustrates the compact notation, where the sequence is determined in topological order, and in addition, precedence is given to higher significant bits in the sequence of indices. For instance, in Fig.2 (right side), indices {3,1} and {3,2} occur at first and second topological levels respectively. With only topological ordering, 4 possible sequences are possible - 3132, 3123, 1332, 1323. Since 3 is given precedence over 1 and 2 at the first and second topological levels respectively, the only possible sequence here is 3132. On the other hand, we can construct a prefix graph by traversing the sequence of indices from left to right in the following way: for each index $i$ in the sequence, we add a node $p$ which is derived from 2 nodes – the most recent node $r$ with index $i$ (or input bit $i$) and the node just before $p$ in the sequence (or the input bit $LSB(r) - 1$). For example, in the sequence '3132' in Fig.2, the node for first 3 is constructed from input bits 3 and 2, where as that for second 3 is constructed from the node for first 3 and the node (with index 1) just before it.

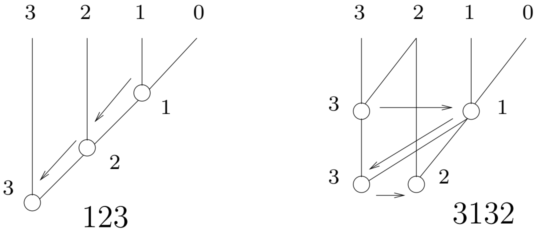


**Figure 2: Compact Notation for a prefix graph**

Apart from storing the *index*, we also need to track the *LSB*, *level*, *fanout* for each node in the prefix graph. We store all this information using a single integer for each node, and represent a prefix graph by a list/sequence of integers. Since we want to explore adders up to 128 bits and provision a carry-in as the $129^{th}$ bit, we reserve 8 bits ($\lceil \log_2(129) \rceil$) for *index*, *level*, *fanout* and *LSB*. Thus, all information for a node can be stored in a single integer as shown in Fig.3.

| | | | |
|---|---|---|---|
| 8 | 8 | 8 | 8 |
| $LSB$ | $fanout$ | $level$ | $MSB(index)$ |

**Figure 3: Bit Slicing**

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

## 3.2 Exhaustive Bottom-up Enumeration

We start from a prefix graph of 2 bits (represented by a single index sequence '1') and construct the prefix graph structures for higher bits in an inductive way, i.e. given all possible prefix graphs ($G_n$) for $n$ bit, we construct all possible prefix graphs ($G_{n+1}$) of $n + 1$ bit. The process of generating such graphs of $n+1$ bit from an element of $G_n$ by inserting $n$ at appropriate positions is a recursive procedure. Fig.4 explains this for an element ('12') of $G_3$ with the help of a recursion tree.

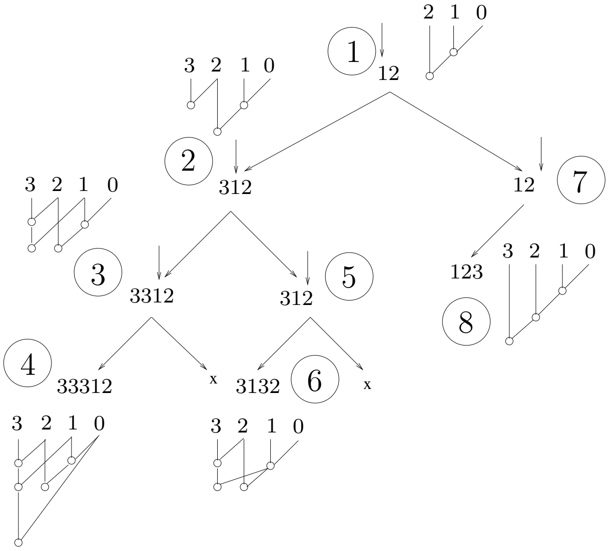


**Figure 4: An illustrative example**

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

Algorithm 1 illustrates the steps of this exhaustive enumeration technique. The algorithm preserves the uniqueness of the solutions by inserting the indices at appropriate positions. In the 'buildRecursive' procedure, $nodeList$ is an STL list ($insert$ and $erase$ operations are thus $O(1)$ operations), $recentNode$ is passed as a parameter which is used to find $searchIndex$ and to track if a solution has been generated. $currIter$ is the iterator corresponding to ↓ in Fig.4. The return value of the procedure is true, when $nodeList$ is a solution of $G_{n+1}$, thereby indicating that the right subtree of parent of $nodeList$ does not require traversal.

## 3.3 Efficient Recursion Implementation

The key step of Algorithm 1 is the recursive procedure as explained in Fig.4. In a pre-order traversal of typical recursion tree implementation, when we move from root node to

**Algorithm 1** Exhaustive Bottom-up Enumeration

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

its left subtree, a copy of the root node is stored to traverse the right subtree at later stage. In our approach, we copy the sequence only when we get a valid prefix graph, otherwise keep on modifying the sequence. As for example, we do not store the sequences ('312', '3312') in Fig.4, i.e. when we move to the left subtree of a node in the recursion tree, we insert the index and delete it while coming back to the node in the pre-order traversal, and store only the leaf nodes. This notion of late copy is motivated by a concept in object-oriented-programming, known as lazy copy or copy-on-write [13] which is a combination of deep copy and shallow copy. In lazy-copy, when an object is copied initially, a shallow copy (fast) is used and then deep copy (slow) is performed when it is absolutely necessary (for example, modifying a shared object). Lazy copy helps to significantly reduce run time by replacing list copy and delete operations with list entry insertion and deletion operations at a given position (iterator) which, being an $O(1)$ operation, does not impact the runtime. For the simple example shown in Fig. 4, an implementation without lazy copy needs 5 list copy and 2 list delete operations whereas an implementation with lazy copy only needs 3 list copy operations and no list delete operations. The benefits of lazy copy increase exponentially with bit-width.

## 3.4 Search Space Reduction

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

**Level Pruning:** The performance of an adder depends directly on the number of logic levels of the prefix graph. Our approach intends to minimize the number of prefix nodes with given bit-width and logic level ($L$) constraints. In Algorithm 1, we keep track of the levels of each prefix node and solutions are discarded if the level of the inserted node (or index) becomes greater than $L$. This work focusses on synthesizing adders with maximum performance and hence, constrains the level at each output bit to the smallest possible value, i.e., bit $m$ is constrained to be at level $\lceil \log_2 m \rceil$.

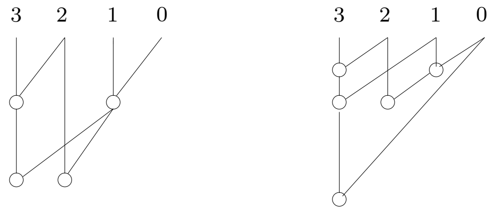
**Dynamic Size Pruning:** As discussed in section 3.2, we construct the set $G_{n+1}$ from $G_n$. While doing this, we prune the solution space based on size (# of prefix nodes) of elements in $G_n$. Let $s_{min}$ be the size of the minimum sized prefix graph(s) of $G_n$. Then we prune the solutions ($g$) for which $size(g) > s_{min} + \Delta$. For example, suppose the sizes of the solutions in $G_n = [9 \quad 10 \quad 11]$ and $\Delta = 2$. To construct $G_{n+1}$, we select the graphs of $G_n$ in increasing order of sizes and build the elements of $G_{n+1}$. Let the graphs with sizes $X_1 = [12 \quad 13 \quad 14 \quad 15]$, $X_2 = [11 \quad 14]$ and $X_3 = [13 \quad 16]$ be respectively constructed from the graphs of sizes 9, 10, 11 in $G_n$. In this case, the minimum size solution is the solution with size 11 and so the sizes of the solutions stored in $G_{n+1} = [[12 \quad 13], [11], [13]]$. This pruning is done to choose the potential elements of $G_{n+1}$, which can give minimum size solution for the higher bits. The selection of $\Delta$ is critical to reduce the search space and we found empirically that $\Delta = 3$ is sufficient to get minimum size solutions for $\log_2 N$ level till 128 bit. But any kind of restriction (like fanout) on the graph structure requires higher $\Delta$ to achieve feasible solutions. In that case, we store a fixed number of solutions of $G_n$ for each size $s$ ($s_{min} \leq s \leq s_{min} + \Delta$), which allows higher $\Delta$ without increasing memory usage too much.

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

**Repeatability Pruning:** The sequence (in our notation) denoting a prefix graph can have consecutive indices. We denote the maximum number of consecutive indices in a sequence by $R$. For example, '33312' in Fig.4 has 3 consecutive 3's in the sequence so $R = 3$. We have observed that $R = 1$ does not degrade the solution quality, but significantly reduces the search space at an early stage. For example, in Fig.5, '3132' is a better solution than '33312' both in terms of logic level and size. Algorithm 1 is modified to track repeatability and prune solutions with $R > 1$.

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

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



sequence: 3132
level = 2, size = 4

sequence: 33312
level = 3, size = 5

**Figure 5: 3132 is better prefix structure than 33312**

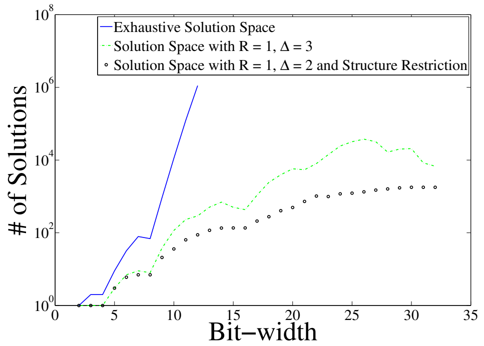


**Figure 6: Search Space Reduction for each output bit $m$ at level $\lceil log_2 m \rceil$**

So 7 nodes or alternatively $(2^n - 1)$ prefix nodes are fixed for the $2^n$ bit adder with $n$ level. We impose this restriction in our implementation for generating the sequence of indices, which helps in improving the run time significantly.

Fig.6 plots the number of solutions (each output bit $m$ being at level $\lceil log_2 m \rceil$) with bit-width for 3 cases, first the exhaustive solution space which grows exponentially with bit-width, next the solution space with $\Delta = 3$, $R = 1$ and without any structure restriction and finally that with structure restriction and $\Delta = 2$, $R = 1$. We have observed that the third case is able to generate the same first 1786 minimum size solutions for 32 bit as that of second case which reinforces that the prefix structure restriction can help in achieving same solution quality with less search space exploration, thereby reducing runtime.

## 4. RESULTS

We have implemented our approach in C++ and executed on a linux machine with 12 GB RAM and 2.8 GHz CPU. First, we present our results at the logic synthesis (technology independent) level. As the dynamic programming based area-heuristic approach presented in [9] has achieved better results compared to the other existing techniques, we have implemented this approach as well to compare with our experimental results. Table 1 presents the comparison of number of prefix nodes for adders with different bit-width ($N$) with $\log_2 N$ logic level constraint for all output bits. In this case, the input profile is uniform, i.e. the arrival times of all input bits are assumed to be the same. Results for non-uniform profile for 32 bit adder are shown in table 2. We can see that our approach outperforms [9] in both cases. The runtime of our approach for generating 128 bit prefix graphs with level constraint of 7 is 25 seconds, which is acceptable for any logic synthesis tool.

**Table 1: Prefix Graph size for $\log_2 N$ level**

| Bit-width | Our Approach | Area Heuristic [9] |
|---|---|---|
| 16 | 31 | 31 |
| 24 | 45 | 46 |
| 32 | 74 | 74 |
| 48 | 105 | 106 |
| 64 | 167 | 169 |
| 128 | 364 | 375 |

**Table 2: Prefix Graph size for non-uniform input profile in a 32 bit adder**

| Profile | Our Approach | Area Heuristic [9] |
|---|---|---|
| A | 55 | 56 |
| B | 55 | 58 |
| C | 56 | 60 |
| D | 54 | 59 |
| E | 53 | 59 |
| F | 55 | 59 |
| G | 53 | 57 |

As mentioned earlier, the existing approaches ([9], [10], [11] etc) are not flexible in restricting parameters like fanout, which is a critical parameter to optimize post-synthesis design performance. Usually, electrical violations at high-fanout points are mitigated by buffer-insertion and gate-sizing, but at the cost of performance. Hence, for high-performance designs, Kogge-Stone [1] is the most effective adder structure. An important property of this structure is that maximum fan-out (MFO) of a $n$ bit adder is less than $\log_2 n$ (without any buffer insertion) and the fan-out for prefix nodes at logic level $\log_2 n - 1$ is 2. Table 3 shows that, even with a fan-out restriction of 2 for *all* prefix nodes, the prefix graph generated by our approach has fewer prefix nodes than the prefix graph for a Kogge-Stone adder.

We have integrated our approach to a placement driven synthesis [14] tool and run the tool on the minimum size solutions of 8,16,32,64 bit adders. A cutting-edge technology node is used for technology mapping. We present the various metrics like area, worst negative slack (WNS), wire-length, figure of merit (FOM) after placement in Table 4 for the solution having best WNS. FOM signifies the sum of the total negative slacks at the timing end-points. Both wirelength and area are unitless. Area is reported as the number of icells and wirelength as the number of tracks. An icell has a constant area based on pitch. Our approach is compared against regular adders like Brent-Kung (BK), Kogge-Stone (KS) adders, adders generated by Dynamic Programming (DP) [9], and 64 bit full custom adder (CT). It is to be noted that we have prevented $V_{th}$-swapping in the placement tool, so leakage power would be proportional to area.

Fig.8 represents the plot of area versus WNS for the solutions provided by our approach along with those provided by other methods. We can draw a pareto curve with the solution points obtained using our approach, which gives the option to select the individual points on the pareto curve based on area/power budget. We see that the solution points of the other methods are above and/or to the right of this curve, which indicates that we can always get some solution on the pareto front, which is better in terms of performance and/or area than each of the other methods. For a 16 bit adder, the total number of pareto-optimal points is 4 and the single point $p1$ provides better solution than DP, KS

**Table 3: Comparison with Kogge-Stone Adder**

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

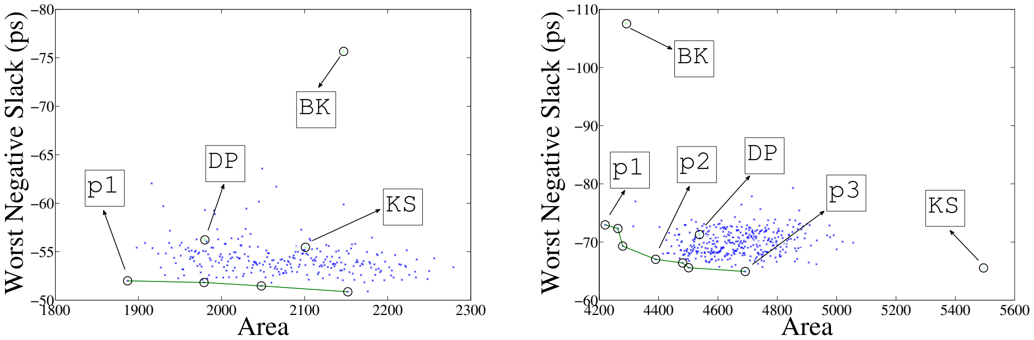


**Figure 8: Area vs. Worst Negative Slack plot for 16 and 32 bit adders**

**Table 4: Post Placement Comparison**

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

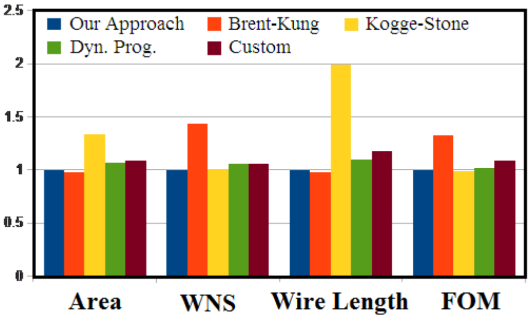


**Figure 7: 64 bit adder after placement.**

and BK. For a 32 bit adder, the points $p1$, $p2$, $p3$ are better solutions than BK, DP, KS respectively.

Fig.7 compares these metrics for single solution (with best WNS) of 64 bit adder with other approaches. Our approach improves performance by 19% with 2% higher area over a Brent-Kung adder, improves performance and area by 0.4% and 33%, respectively over a Kogge-Stone adder, improves performance and area by 3% and 6.7%, respectively over Dynamic Programming [9], and improves performance and area by 3.2% and 8.5% over a full custom adder design. Note that the performance improvement was computed based on the actual critical path delay value and not the worst negative slack. Our approach also improves wire-length and FOM over both Kogge-Stone and full custom adder design.

## 5. CONCLUSION AND FUTURE WORK

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

## 6. REFERENCES

[1] P. M. Kogge and H. S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. *IEEE Trans. Computers*, pages 786–793, 1973.

[2] J. Sklansky. Conditional sum addition logic. *IRE Trans. on Electronic Computers*, pages 226–231, 1960.

[3] R. P. Brent and H. T. Kung. A regular layout for parallel adders. *IEEE Trans. Computers*, pages 260–264, 1982.

[4] C. Zhou et al. 64-bit prefix adders: Power-efficient topologies and design solutions. *Custom Integrated Circuit Conference*, pages 179–182, 2009.

[5] J. Liu et al. Optimum prefix adders in a comprehensive area, timing and power design space. *ASPDAC*, 2007.

[6] M. Snir. Depth-size trade-offs for parallel prefix computation. *Journal of Algorithms*, pages 185–201, 1986.

[7] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *Journal of ACM*, pages 831–838, 1980.

[8] 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*, pages 361–364, 1990.

[9] T. Matsunaga and Y. Matsunaga. Area minimization algorithm for parallel prefix adders under bitwise delay constraints. *Great Lakes Symposium on VLSI*, pages 435–440, 2007.

[10] J. Liu et al. An algorithmic approach for generic parallel adders. *International Conference on Computer Aided Design*, pages 734–740, 2003.

[11] R. Zimmermann. Non-heuristic optimization and synthesis of parallel prefix adders. *International Workshop on Logic and Architecture Synthesis*, pages 123–132, 1996.

[12] A. K. Verma and P. Lenne. Towards the automatic exploration of arithmetic-circuit architectures. *DAC*, pages 445–450, 2006.

[13] H. Sutter. *More Exceptional C++*. Addison Wesley, 2002.

[14] H. Ren et al. Sensitivity guided net weighting for placement driven synthesis. In *International Symposium on Physical Design*, pages 10–17, 2004.

# APPENDIX

## A. NON-UNIFORM INPUT PROFILE

In Table 2, we have compared the result for non-uniform input profile. The required time of arrival for all output bits are set to 9 and the input arrival levels have been randomly generated between 0-4. Table 5 presents those arrival levels of each input bit for all profiles.

**Table 5: Input Arrival Times for Table 2**

| Bit | A | B | C | D | E | F | G |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 1 | 2 | 1 | 2 | 2 |
| 1 | 2 | 1 | 3 | 3 | 2 | 3 | 1 |
| 2 | 1 | 3 | 2 | 1 | 1 | 3 | 1 |
| 3 | 3 | 2 | 3 | 1 | 1 | 1 | 2 |
| 4 | 4 | 1 | 2 | 2 | 1 | 2 | 1 |
| 5 | 2 | 0 | 1 | 3 | 3 | 1 | 2 |
| 6 | 1 | 4 | 3 | 2 | 2 | 1 | 1 |
| 7 | 3 | 3 | 2 | 1 | 3 | 1 | 2 |
| 8 | 1 | 2 | 1 | 4 | 4 | 1 | 1 |
| 9 | 2 | 1 | 3 | 3 | 2 | 2 | 3 |
| 10 | 1 | 3 | 4 | 2 | 3 | 2 | 2 |
| 11 | 0 | 2 | 2 | 1 | 1 | 2 | 2 |
| 12 | 3 | 2 | 1 | 3 | 2 | 3 | 2 |
| 13 | 2 | 1 | 2 | 2 | 3 | 2 | 2 |
| 14 | 1 | 4 | 4 | 1 | 2 | 3 | 2 |
| 15 | 4 | 2 | 1 | 1 | 1 | 2 | 1 |
| 16 | 2 | 1 | 3 | 2 | 2 | 1 | 3 |
| 17 | 2 | 2 | 1 | 3 | 2 | 1 | 1 |
| 18 | 1 | 3 | 2 | 2 | 1 | 4 | 1 |
| 19 | 0 | 1 | 1 | 1 | 2 | 1 | 2 |
| 20 | 1 | 4 | 2 | 3 | 1 | 2 | 3 |
| 21 | 3 | 0 | 1 | 1 | 1 | 1 | 4 |
| 22 | 4 | 2 | 1 | 4 | 2 | 1 | 4 |
| 23 | 1 | 1 | 2 | 1 | 2 | 2 | 4 |
| 24 | 2 | 2 | 3 | 2 | 2 | 1 | 2 |
| 25 | 2 | 1 | 2 | 3 | 2 | 2 | 1 |
| 26 | 1 | 3 | 4 | 2 | 1 | 1 | 4 |
| 27 | 3 | 2 | 1 | 3 | 3 | 2 | 2 |
| 28 | 1 | 1 | 3 | 1 | 2 | 1 | 4 |
| 29 | 2 | 3 | 2 | 2 | 4 | 1 | 1 |
| 30 | 2 | 1 | 2 | 1 | 1 | 2 | 2 |
| 31 | 3 | 2 | 1 | 2 | 2 | 1 | 3 |

Table 6 compares our approach with [9] and [11] for correlated input profile, like late higher words or monotonically increasing inputs, appeared in [11].

**Table 6: Comparison on Zimmermann's examples**

| DATA | Our Approach | Area [9] Heuristic | Zimmermann [11] |
|---|---|---|---|
| A | 49 | 49 | 50 |
| B | 59 | 61 | 61 |
| C | 56 | 56 | 56 |
| D | 63 | 64 | 63 |
| E | 50 | 55 | 55 |
| F | 73 | 73 | 73 |
| G | 56 | 58 | 59 |
| H | 78 | 79 | 78 |
| I | 68 | 68 | 68 |

## B. IMPACT OF MFO ON POST PLACEMENT RESULT

In Fig.9, the worst negative slack ($WNS$) is plotted against the size of the prefix graph for 16 bit adders. We can see that the prefix graphs of higher node count and smaller maximum fan-out ($MFO$) are better for timing.

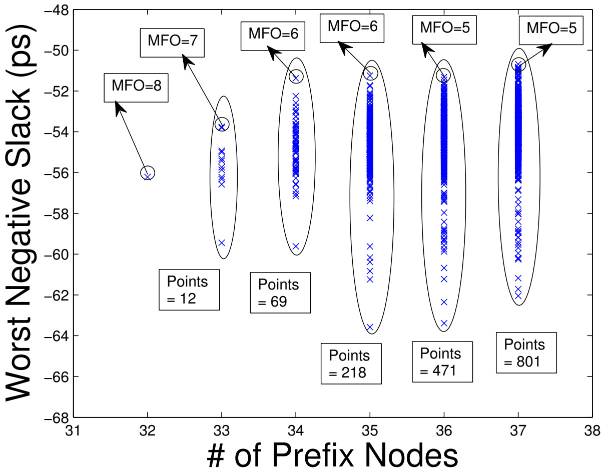


**Figure 9: # of Prefix Nodes vs. WNS for 16 bit adder**

## C. PREFIX GRAPHS

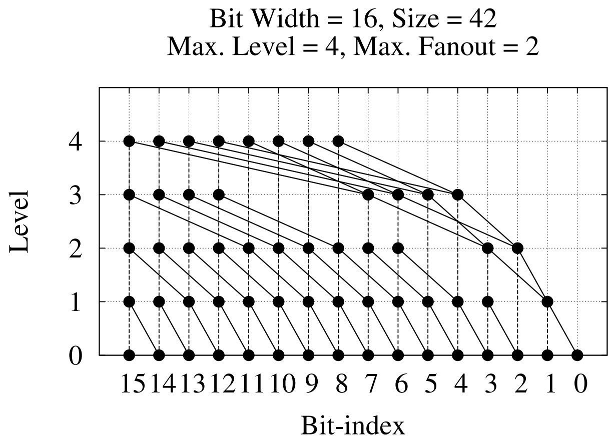
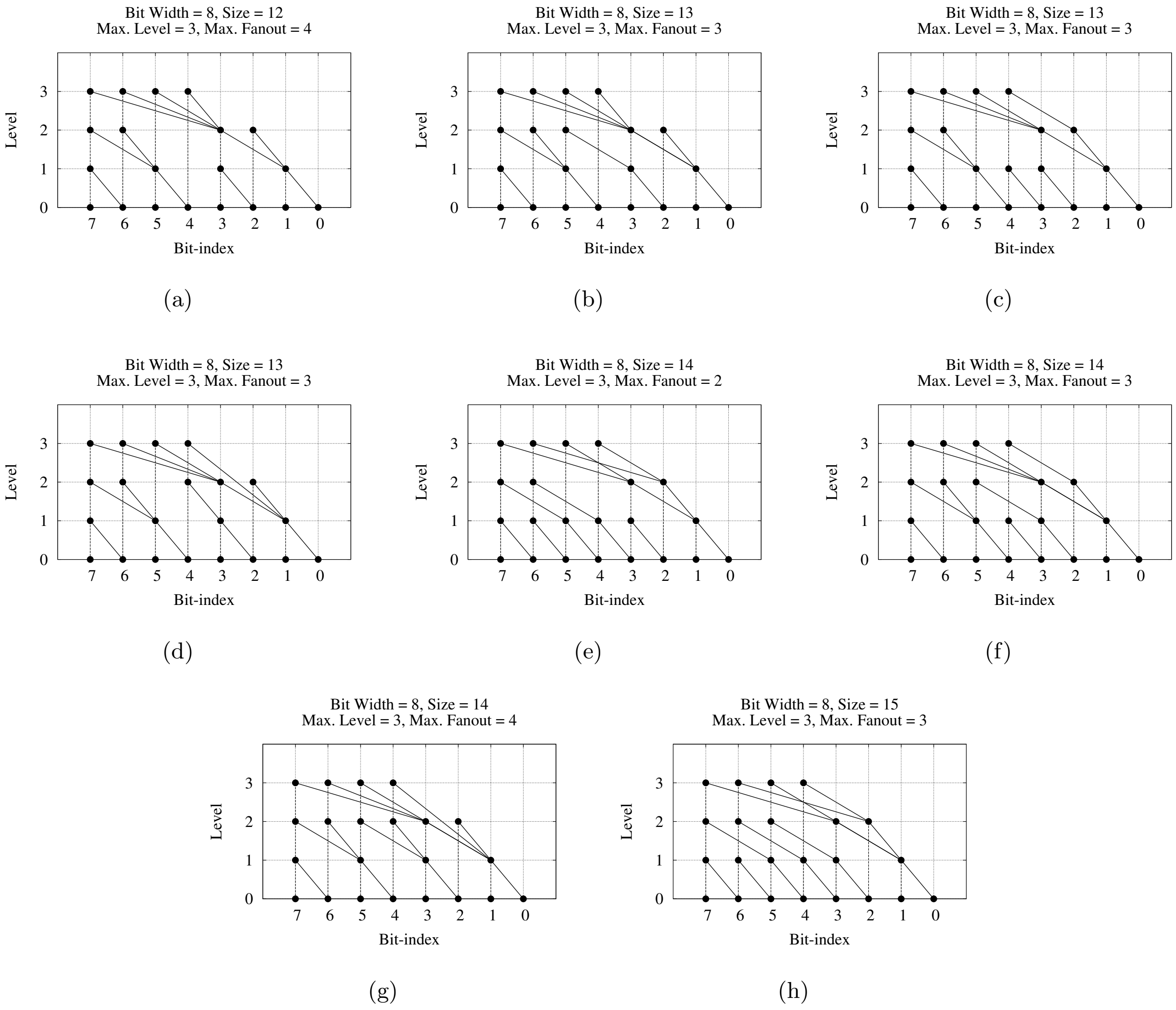


**Figure 10: Size of a 16 bit prefix graph with level 4 and fanout 2 generated by our approach is less than that of Kogge Stone by 7**



Bit Width = 8, Size = 12
Max. Level = 3, Max. Fanout = 4

(a)

Bit Width = 8, Size = 13
Max. Level = 3, Max. Fanout = 3

(b)

Bit Width = 8, Size = 13
Max. Level = 3, Max. Fanout = 3

(c)

Bit Width = 8, Size = 13
Max. Level = 3, Max. Fanout = 3

(d)

Bit Width = 8, Size = 14
Max. Level = 3, Max. Fanout = 2

(e)

Bit Width = 8, Size = 14
Max. Level = 3, Max. Fanout = 3

(f)

Bit Width = 8, Size = 14
Max. Level = 3, Max. Fanout = 4

(g)

Bit Width = 8, Size = 15
Max. Level = 3, Max. Fanout = 3

(h)

**Figure 11: 8 bit prefix graphs with level 3**