Generation of the Optimal Bit-Width Topology of the Fast Hybrid Adder in a Parallel Multiplier

Sabyasachi Das
Synplicity Inc
Sunnyvale, CA, USA
sabya@synplicity.com

Sunil P. Khatri
Texas A&M University
College Station, TX, USA
sunilkhatri@tamu.edu

***Abstract*— In state-of-the-art Digital Signal Processing (DSP) and Graphics applications, multiplication is an important and computationally intensive operation, consuming a significant amount of delay. The final carry propagate hybrid adder inside a multiplier plays an important role in determining the performance of the multiplication block. This paper presents an algorithmic approach to generate the optimal bit-width configuration of each of the sub-adders present inside the hybrid adder. Our technique is useful in selecting the best configuration (out of a large number of possible configurations) of the hybrid adder, thereby improving the overall performance of the chip. Our experiments involve different combinations of designs, technology libraries and timing constraints, and the results show that our algorithm successfully predicts the best hybrid-adder topology with a very low runtime.**

I. Introduction

The complexity and the performance requirement of the datapath operations implemented in systems on chips has increased considerably over the years. This is especially true in the chips for communication, multimedia and graphic applications, which have highly parallel implementations of signal processing algorithms.

The multiplier is one of the most widely used arithmetic datapath operations in modern digital design. Since multiplier blocks require intensive computations, they exhibit a significant amount of delay, and therefore tend to be typically found in the timing-critical path of the chip. Developing an efficient architecture would reduce the delay of the multiplier block and thereby improve the performance of the chip. Hence there is great interest in generating the optimal architecture for parallel multipliers.

Inside the multiplier, a carry propagate adder is typically used, contributing about 30% to the total delay of the multiplier. Hence the selection of the optimal architecture of the adder plays a key role in determining the critical path delay of the multiplier. In [1], the authors present a very effective timing-driven strategy to design a fast parallel multiplier. They divide the adder module into multiple sub-adder blocks and use different architectures for the different sub-blocks. In this paper, we propose an algorithmic approach to determine the optimal topology[1] of the fast hybrid adder present inside a multiplier. We use a timing-driven approach to generate the optimal bit-widths of each of the sub-adder blocks inside the hybrid adder. To the best of our knowledge, there is no other published work focusing on automated generation of exact topologies of the sub-adders inside the hybrid adder.

[1]By topology, we mean the number, bit-width, and type of each of the sub-adders comprising the hybrid adder.

```
module mult (z, a, b);

  parameter N = 16;

  input [N-1:0] a, b;
  output [2*N-1:0] z;

  assign z = a * b;

endmodule
```

Fig. 1. Verilog RTL of the 2-input multiplication block

We have organized the rest of the paper as follows: In Section II, we present some background information. In the Section III, we present the definition of the problem we are addressing in this paper. In Section IV, we discuss our proposed approach in detail. Section V presents the experimental results. Conclusions are drawn in Section VI.

II. Preliminaries

We briefly explain the concept of multipliers by using the example of a 2-input multiplier block, and discuss how it is typically synthesized. Figure 1 represents an RTL depicting a 2-input multiplier. In this block, there are two 16-bit wide inputs (a and b), which produce the 32-bit wide output z.

Once the multiplier block is synthesized, the resulting netlist consists of the following three parts:

- The Partial Product Generator
- Partial Product Reduction Tree
- Final Carry Propagate Adder

The partial products in a multiplier are generated by performing a bit-wise multiplication (2-input AND operation) between the appropriate bits of the multiplicand and the multiplier. Each partial product is shifted by one or more bits, depending on the bit number of the multiplicand. If PP_i is the i^{th} partial product of $a * b$, and b has n bits, then partial products can be represented by the following expression:

$$PP_i = a * b_i * 2^i \qquad for\ i = 0, 1, ...(n-1)$$

After the generation of the partial products, all the n partial products need to be reduced to two vectors. To perform this operation, a *Partial Product Reduction Tree* is implemented. To design the partial product reduction tree, the *column-compression* technique is widely used [3], [4]. Column compression reduces the addition operations of any bitslice into a cascade of full adder operations, finally yielding two addends per bit-slice in the partial-products.

After the reduction of partial products, the resulting two vectors (n-bit wide) are fed to a final carry propagate adder (CPA) which produces the final result of the multiplication block. Since carry propagation is an expensive operation, this contributes roughly to 30% of the total delay of the multiplication block [1]. Hence any improvement in the delay of this addition sub-block becomes quite significant in determining the performance of the multiplier module.

III. Problem Definition

To design a fast multiplier, it is essential to implement a fast partial product reduction tree, and a fast special-purpose *hybrid adder*.

Due to the inherent tree structure of the column-compression operation, the partial-product reduction tree [3], [4], [5] produces a skewed timing profile at the outputs of the reduction tree. This results in a non-uniform input arrival time profile for the final carry propagate adder. In the Figure 2, we plot the timing profile of the inputs to the carry-propagate adder (in a multiplier block which implements the expression $z = a * b$). The tree reduction phase produces the two output vectors x and y, which become the inputs to the final carry propagate adder (CPA). Figure-2 shows that the *middle* bits of x and y have largest arrival time. The bits which are near either the most significant bit (MSB) or the least significant bit (LSB) of x or y have earlier arrival times than the bits in the middle.

Due to this pattern of arrival-times to the CPA, several traditional general-purpose stand-alone timing-driven addition schemes [8] do not work well in the context of multipliers. That is because most of the high-speed adders are designed with an assumption that all the input signals arrive at the same time.

The authors of [1] and [2] have introduced a special-purpose *hybrid adder* specifically to be used for the carry propagate adder in a high-performance multiplier. They split the carry propagate adder into the following three sub-addition blocks:

- A slow adder for *few* bits near LSB. In our implementation, we use the *ripple carry* architecture for this slow adder.
- A fast adder for *several* bits in the middle. In our implementation, we use *Brent-Kung* architecture for this fast adder.

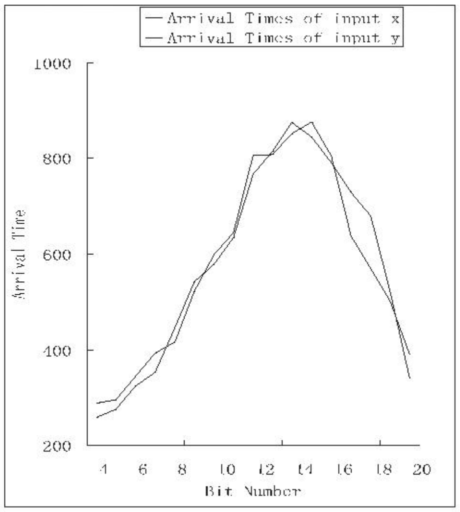

Fig. 2. Arrival time pattern for the two inputs of the CPA (in a multiplier)

- A carry-select adder for the *remaining* bits near MSB. In our implementation, we use *Brent-Kung* architecture for this carry-select adder.

The authors of [1] and [2] have claimed that the improvement in speed of this *hybrid adder* over the commonly used carry-look-ahead addition structure is about 12.5%. While this data is very encouraging, one of the most important tasks in designing this hybrid addition structure is to effectively determine the width of each of the sub-adder blocks. Hence, the *topological structure* of the hybrid adder plays a key role in determining the performance gain of this addition technique.

In this paper, we propose an arrival-timing-skew based algorithmic approach to determine the topology of the hybrid adder. The topology of the hybrid adder is defined as the individual bit-widths of all three sub-adder blocks present within the hybrid adder module. Topology$_{add}$ = (w_1, w_2, w_3), indicates the configuration in which the least significant w_1 bits form the ripple sub-adder, the next w_2 bits form the fast Brent-Kung sub-adder in the middle and the most significant w_3 bits create the carry-select sub-adder. Note that $w_1 + w_2 + w_3 = n$. Our experimental data suggests that using a sub-optimal topology for the hybrid adder can result in performance which is even worse that the traditional fast addition techniques [6], [7]. Hence it is critical to identify the best topology of the hybrid adder.

IV. Our Approach

Throughout the rest of the paper, we would assume that two inputs (x and y) to the hybrid adder are n-bit wide and the output (z) of the adder is $(n+1)$-bit wide. In addition, let us assume that the arrival times of the two inputs are denoted as follows: Let, tx_i be the arrival time of the i^{th} bit of the input vector x. Similarly, Let, ty_i be the arrival time of the i^{th} bit of the input vector y.

Our proposed technique has two steps. In the following subsections, we discuss each step in detail.

A. Determination of the width of the Ripple Carry Adder

In the hybrid adder, the sub-adder for the least significant w_1 bits can be a slower one since these bits arrive early. Hence the traditional Ripple Carry adder is the best choice. To decide the bit-width of the ripple carry adder, we use the approach presented in Algorithm 1. Detailed comments are provided below.

The timing-skew pattern of the inputs to the hybrid adder shows that the arrival times of the bits near LSB are significantly less than that of the bits in the middle. The key idea is to exploit that characteristic and ensure that the ripple adder sub-block produces the output carry to the faster Brent-Kung adder sub-block (for the middle bits) *before* the middle bits arrive.

To use in our ripple adder, we analyze the technology library cells and identify a Full-Adder cell having the least amount of delay from the input to the carry$_{out}$ pin. This Full-Adder cell and its associated input to carry$_{out}$ delay (denoted as δ) is used throughout the algorithm. We start the timing analysis from the LSB (0^{th} bit) and test if the Full-Adder cell would produce the carry$_{out}$ (which becomes carry$_{in}$ to the 1^{st} bit) before the latest arriving signals among x_1 and y_1 arrive. If the result of the test is true, then we include the 0^{th} bit in the ripple carry adder. This test is carried out for the 1^{st}, 2^{nd}, 3^{rd} bits and so on. In addition, if for some bit (let's say, the i^{th} bit), the timing analysis shows that the carry$_{out}$ (or the carry$_{in}$ to the $(i+1)^{th}$ bit) is generated after the latest arriving signals at bit $(i+1)$ arrive, then it is still possible that we need to include the i^{th} bit in the ripple carry adder. This is because the increase in arrival time for the x_i and y_i signals does not have a constant slope as i increases towards the middle bits (as shown in Figure-2). Hence we do not terminate our algorithm immediately. If we terminate our algorithm at this stage and decide that the width of our ripple carry adder is i-bits (0^{th} bit to $(i-1)^{th}$ bit), then there is a risk of accepting a locally optimal solution instead of a globally optimum one. Hence we perform *hill climbing*. Instead of quitting immediately, we analyze upto 2 additional bits (a total of 3 bits i, i+1 and i+2). If in *these* 3 bits, the analysis confirms that the carry$_{out}$ is generated later than the latest arriving input signal of the next bit, then our algorithm quits and returns the bit-number i as the width of the ripple carry adder. On the other hand, if during the 3 bits of hill climbing, the analysis in any bit j indicates that the carry$_{out}$ is generated *earlier* than the latest arriving input signal in the next bit, then we include bit j in the ripple carry adder block. In addition, we reset our hill climbing mode and switch back to the original mode and continue our algorithm from the next most significant bit. Our experiments with different designs and timing profiles have proved the usefulness of the hill-climbing phase to come out of the sub-optimal solution.

Algorithm 1 : Determination of Ripple Carry Adder Width

```
HillClimbing = false
HillPenalty = 0
HillStep = 0

$tmax_m$ = Max ($tx_0$, $ty_0$, $tx_1$, $ty_1$, ..., $tx_{n-1}$, $ty_{n-1}$)
// Hence $m$ is the bit-number having largest arrival-time

for $i$ = 0 to $m$ do
  ArrCur = Max($tx_i$, $ty_i$)
  ArrNext = Max($tx_{i+1}$, $ty_{i+1}$)
  if HillClimbing then
    // Hill-climbing mode
    if (ArrCur + $\delta$ + HillPenalty) $\leq$ ArrNext then
      $w_1$ = $i$
      HillClimbing = false
      HillPenalty = 0
      HillStep = 0
    else
      // Try total of three steps of hill-climbing
      HillPenalty = (ArrCur + $\delta$ + HillPenalty)
                    - ArrNext
      HillStep = HillStep + 1

      if (HillStep > 3) then
        return $w_1$          // No further attempt
      end if
    end if
  else
    // Regular (Non hill-climbing) mode
    if (ArrCur + $\delta$) $\leq$ ArrNext then
      $w_1$ = $i$
    else
      HillClimbing = true
      HillPenalty = (ArrCur + $\delta$) - ArrNext
      HillStep = 1
    end if
  end if
end for

return $w_1$      // The Bit-width of the Ripple-Carry-Adder
```

B. Determination of widths of the remaining Fast Adders

After determining that the ripple carry adder (near the LSBs) should be w_1 bit wide, the remaining (n-w_1) bits need to be split between the fast adder (in the middle) and the fast carry-select adder near the most significant bits (MSBs). In this paper, we use the Brent-Kung (BK) adder architecture [6] inside the two fast adders. Depending on the timing characteristics, the width of the fast BK adder in the middle can also be (n-w_1), obviating the need for the carry-select adder altogether. In the remainder of this section, we refer to the adder of the middle bits as the *middle adder*.

This scheme of using the middle adder and the carry-select adder is very useful in reducing the overall delay of the adder. However, if the widths of these two adders are small, then it is better to use a single (n-w_1) bit adder. Hence if (n-w_1) is less than 16 bits, we use a single middle adder of the Brent-Kung architecture, having a bit-width of (n-w_1) bits. On the other hand, if $(n - w_1) \geq 16$, then we use the algorithm explained in the rest of this subsection.

The Brent-Kung (BK) architecture [6] is a widely used fast architecture and is based on the parallel-prefix network computation. If a BK adder has k inputs, then there can be a maximum of $(2 * \log_2 k - 2)$ levels in the adder [9]. If the number k is a power of 2, then the number of levels at the most significant bit would be $(\log_2 k)$ only.

In this algorithm, we *analyze* several configurations of the hybrid adder (without really building the actual adder structures) and then select the best topology. Let us first define a naming convention for the configurations. A configuration q indicates that the bit-width of the fast Brent-Kung adder in the middle is q bits and the bit-width of the carry-select adder near the MSBs is (n-w_1-q) bits.

To start the algorithm, we select the configuration c_1, where

- $c_1 = 2^m$
- $m = \lfloor log_2(n - w_1) \rfloor$

Now, we have to *estimate* the delay of this configuration. Since the BK adder extensively uses the well-defined *carry operator* [9], we compute the delay of the fastest possible carry operator generated from the technology components available in the library. With that delay available, we can estimate the following two:

1) The time when the sum output will be ready at the most critical sum pin. We refer to this as TMid$_{sum}$.
2) The time when the carry$_{out}$ of the middle adder block will be ready. We refer to this as TMid$_{co}$.

We can similarly compute the time when the sum output will be ready at the most critical sum pin of the carry-select adder. We refer to this as TCS$_{sum}$. Note that for a carry-select adder, we implement two parallel adders whose outputs are connected to a 2:1 Multiplexer (MUX) cell. Therefore, we need to account for the delay of the MUX cell. The delay of the fastest 2:1 Multiplexer (MUX) cell in the provided technology library is denoted by Del$_{mux}$. Now, with all these numbers computed, the time when the sum output of the most critical sum pin of the combined adder (middle adder and the carry-select adder near the MSB) will be ready is:

T =Max (TMid$_{sum}$, (TMid$_{co}$+Del$_{mux}$), (TCS$_{sum}$+Del$_{mux}$)).

The value (T) represents the time when the most critical output will be ready if the bit-width of the middle adder is c_1 and the carry-select adder is (n-w_1-c_1). We mark c_1 as our best configuration so far. Now, we perform similar analysis for several other configurations (bit-widths) described in the next paragraph. Out of all the analyzed configurations, the optimal configuration is the one which has the lowest value of T.

Next, we have to decide whether exploration should be in the direction of the increased bit-width or in the direction of the reduced bit-width of the middle adder. If the width of the carry-select adder (n-w_1-c_1) is less than 4 bits, then we conclude that further exploration should be performed in the direction of reduced width of the middle adder. On the other hand, if the width of the carry-select adder is greater than 4 bits, then we choose the two configurations (c_1+1) and (c_1-1). After computing the T values for both the configurations, if T(c_1+1) > T(c_1-1), then further exploration will be performed only in the direction of reduced bit-width of the middle adder. Otherwise, further exploration will be performed only in the direction of increased bit-width of the middle adder. If further exploration is performed in the direction of reduced bit-width of the middle adder, then the middle adder width range is from 0 bits to c_1 bits. On the other hand, if further exploration is performed in the direction of increased width of the middle adder, then the middle adder width range is from c_1 bits to (n-w_1) bits. In the rest of the section, we only explain the situation where further exploration happens in the direction of increased bit-width of the middle adder. The reverse situation is analogous, hence it is not separately explained.

Once the direction of further exploration (reduced or increased bit-width for the middle adder) is decided, we perform a bit-width exploration in a fashion which is similar to a binary search algorithm. Now, to perform further exploration of other configurations (assuming that further exploration is performed in the direction of increased bit-width of the middle adder), we select the mid-point between $c1$ and (n-w_1) as the next bit-width of the middle order. Let us call that the configuration c_2. After analyzing the delay through that configuration, if we find out that $T(c_2) < T(c_1)$, then

- We mark c_2 as our best configuration so far (and discard the configuration c_1).
- For the exploration of the next configuration, we need to decide whether to explore further in the direction of increased bit-width or in the direction of reduced bit-width of the middle adder. Hence we choose the two configurations (c_2+1) and (c_2-1) and continue the process explained earlier.

On the other hand, if we find that $T(c_2) > T(c_1)$, then

- Since c_1 is our best configuration so far, we discard the configuration c_2.
- For the exploration of the next configuration, we select the mid-point between c_2 and c_1 as the bit-width of the middle-adder.
- We do not need to explore any configuration of the middle adder having $w_2 \geq c_2$ bits.

We repeat this process until the algorithm converges to one configuration. Let us call this configuration c_{w_2}. This indicates that the optimal configuration is to implement a middle adder having w_2 input bits and a carry select adder having (n-w_1-w_2) number of input bits.

By using the above-mentioned algorithms, we can very quickly identify the optimal topology for all three sub-adders (ripple carry adder, fast BK adder, carry-select adder) inside the hybrid adder.

V. Experimental Results

We have implemented our proposed algorithm in the C++ programming language. The experiments were performed with multiplier RTL designs written in the Verilog hardware description language [10]. For all our runs, we used a Linux workstation with dual-2.2GHz processors and 4GB memory. The operating system was RedHat 7.1.

To collect different data-points regarding the quality of results of our algorithm, we used the following variations:

- Multiplier designs of different input bit-widths:
  In the Table-I, we report the different configurations of the multiplier designs that have been used in our experiments. The table contains design-name (1^{st} column), the widths of the two inputs to the multiplier (2^{nd} column), the width of the output of the multiplier (3^{rd} column) and the width of each of the two inputs to the hybrid adder in the multiplier (4^{th} column). We do not restrict the two inputs of our multiplier designs to have same bit-widths. If the bit-widths of the two inputs to the multiplier are m bits and n bits, then the final output is of width (m+n) bits. In this multiplier, each of two inputs to the hybrid adder are of width (m+n-2) bits and the output of the hybrid adder is (m+n-1) bits wide. This is because the least significant bit (0^{th}) of the final result is obtained directly from the partial product generator without going through the hybrid adder.

Mult Name	Mult i/p Width	Mult o/p Width	Adder i/p Width
Mult-1	36 x 29	65	63
Mult-2	24 x 24	48	46
Mult-3	41 x 36	77	75
Mult-4	32 x 32	64	62
Mult-5	21 x 16	37	35
Mult-6	27 x 27	54	52

TABLE I
MULTIPLIER DESIGN CHARACTERISTICS

- Different technologies and libraries:
  - One library (Lib-X) for 0.13μ.
  - One library (Lib-Y) for 0.09μ.

  These libraries are created by two well-known vendors of commercial libraries.
- Different input arrival time constraints:
  To test the effectiveness of our algorithm in different realistic situations, we used the following scenarios, which generate different input arrival time constraints:
  1) One input of the multiplier design comes from the output of an addition (a+b) block. The other input of the multiplier design comes from a subtractor (c-d) block.
  2) One input comes from the output of another multiplier (p*q) and the other input comes from the output of a divider (r/s) block.
  3) One input comes from the output of a Sum-of-Product (a*b+c*d) block and the other input comes from the output of a Multiply-Accumulator (p*q+r) block.
  4) One input comes from the output of a shifter (a>>b) and the other input comes from the output of an incrementor (a+1) block.
  5) Both the inputs come from the outputs of register banks.

  In the timing constraints 1, 2, 3 and 4, the different bits at the inputs of the multiplier arrive at different times. We believe that these constraints represent most of the real-life timing situations for multipliers. In the timing constraint 5, all the inputs of the multiplier arrive at the same time. With this constraint, we model the situation when the primary inputs of the design are being fed to a multiplier or when the multiplier is placed immediately after a set of register banks.

In our source code, we have two top-level routines. The code in routine-1 implements the approach presented in this paper and it estimates the optimal widths of three different adders present in the hybrid adder. In other words, this code computes (w_{1opt}, w_{2opt}, w_{3opt}) The code in routine-2 takes three width parameters (w_1, w_2, w_3) as inputs and actually generates the hybrid adder having ripple adder of width w_1 bits, a fast Brent-Kung adder with width w_2 bits, and the carry select adder of width w_3 bits.

To measure the effectiveness and correctness of our algorithm, we use the following methodology. Assume that we are trying to validate our algorithm on a multiplier design m_1, with technology library l_1 and timing constraint t_1. If the final carry propagate hybrid adder in the multiplication module has 2 inputs each having n-bits, then there are $\binom{n+2}{2}$ different topologies possible for that hybrid adder. We first execute routine-1 of our source code (which is our algorithm to identify the best topology of the hybrid adder). Suppose our algorithm suggests that the best topology of the hybrid adder is Topology$_{opt}$ = (w_{1opt}, w_{2opt}, w_{3opt}). Note that, the topology of (w_{1opt}, w_{2opt}, w_{3opt}) is *one* out of the possible $\binom{n+2}{2}$ configurations. Next, we execute routine-2 of our source code with each of the possible configurations and note the delays in each case. From the entire set of implementations, we identify the fastest implementation and call it Topology$_f$ = (w_{1f}, w_{2f}, w_{3f}). If we find that Topology$_{opt}$ matches with Topology$_f$ (or in other words, if w_{1opt}=w_{1f}, w_{2opt}=w_{2f} and w_{3opt}=w_{3f}), then we conclude that our algorithm is able to accurately predict the best possible implementation topology or configuration for that hybrid adder module inside the multiplier.

For each combination of design, technology library and timing constraint, we perform the above-mentioned analysis to verify the accuracy of our algorithm. In Table-II, we report the best hybrid-adder topology predicted by our algorithm on 24 different combinations of designs, technology libraries and timing constraints, which are reported one per row of Table-II. In each row of the table, we present the topology of the hybrid adder (widths of all three sub-adder blocks) predicted by our algorithm. Our algorithm is able to predict the best hybrid adder topology for each of the 24 different situations presented in the Table-II. We have experimentally verified that Topology$_{opt}$ = Topology$_f$ in each case.

			Optimal Topology of the Hybrid Adder		
Multiplier Design Name	Technology Library	Input Arrival Timing Constraint	Bit-Width of the Ripple Carry Adder (near the LSB)	Bit-Width of the Brent-Kung Adder (in the Middle)	Bit-Width of the Carry Select Adder (near the MSB)
Mult-1	Lib-X (0.13μ)	Constraint-1	11 bits	45 bits	7 bits
Mult-2	Lib-X (0.13μ)	Constraint-2	4 bits	32 bits	8 bits
Mult-3	Lib-X (0.13μ)	Constraint-3	23 bits	34 bits	18 bits
Mult-4	Lib-X (0.13μ)	Constraint-4	7 bits	43 bits	12 bits
Mult-5	Lib-X (0.13μ)	Constraint-5	3 bits	27 bits	5 bits
Mult-6	Lib-X (0.13μ)	Constraint-1	10 bits	38 bits	6 bits
Mult-1	Lib-X (0.13μ)	Constraint-2	7 bits	50 bits	6 bits
Mult-2	Lib-X (0.13μ)	Constraint-3	13 bits	28 bits	5 bits
Mult-3	Lib-X (0.13μ)	Constraint-4	16 bits	41 bits	18 bits
Mult-4	Lib-X (0.13μ)	Constraint-5	8 bits	43 bits	11 bits
Mult-5	Lib-X (0.13μ)	Constraint-1	6 bits	22 bits	7 bits
Mult-6	Lib-X (0.13μ)	Constraint-2	5 bits	39 bits	8 bits
Mult-1	Lib-Y (0.09μ)	Constraint-3	10 bits	45 bits	8 bits
Mult-2	Lib-Y (0.09μ)	Constraint-4	5 bits	34 bits	7 bits
Mult-3	Lib-Y (0.09μ)	Constraint-5	18 bits	44 bits	13 bits
Mult-4	Lib-Y (0.09μ)	Constraint-1	18 bits	37 bits	7 bits
Mult-5	Lib-Y (0.09μ)	Constraint-2	9 bits	21 bits	5 bits
Mult-6	Lib-Y (0.09μ)	Constraint-3	3 bits	44 bits	5 bits
Mult-1	Lib-Y (0.09μ)	Constraint-4	12 bits	42 bits	9 bits
Mult-2	Lib-Y (0.09μ)	Constraint-5	6 bits	33 bits	7 bits
Mult-3	Lib-Y (0.09μ)	Constraint-1	14 bits	49 bits	12 bits
Mult-4	Lib-Y (0.09μ)	Constraint-2	17 bits	34 bits	11 bits
Mult-5	Lib-Y (0.09μ)	Constraint-3	2 bits	28 bits	5 bits
Mult-6	Lib-Y (0.09μ)	Constraint-4	9 bits	36 bits	7 bits

TABLE II

OPTIMAL TOPOLOGY OF DIFFERENT HYBRID ADDERS GENERATED BY OUR APPROACH

For a 32-bit hybrid adder, our proposed prediction approach (routine-1 of our source code) and one execution of routine-2 to build the hybrid adder with the predicted optimal configuration totally takes about 6 minutes of runtime. In case of the 32-bit wide hybrid adder, the total number of possible topologies is 561. In the exhaustive approach, if we call routine-2 of our source code 561 times and then select the best topology or implementation, then the total runtime is over 19 hours. This shows that to identify the best configuration, it is not practical to try all possible configurations of the hybrid adder. This also indicates that our algorithm is able to derive the optimal configuration of the three adders, with significantly lower runtime. Hence our algorithm is an accurate and extremely fast approach to determine the best topology for the hybrid adder without any necessity to perform an exhaustive exploration.

Our prediction of the topology consistently produces the optimal result across different types of multipliers, timing constraints and technology libraries. This exhibits the strength and wide applicability of our algorithm. Since multiplication is a very compute intensive operation and the selection of the hybrid adder topology plays a significant role in determining the overall performance, we believe that many real-life designs can significantly benefit from our algorithm.

VI. CONCLUSION

In this paper, we have presented an approach for generating the optimal topology of the hybrid adder inside a fast parallel multiplier. This fast technique accurately estimates the bit-widths of each of the three sub-adder blocks inside the hybrid adder. This approach is very useful in optimally synthesizing the computationally intensive and timing-critical multiplication block within a very short period of time. Our estimation technique works seamlessly with different sizes of multipliers, timing constraints and across different technology domains (0.13μ, 0.09μ etc). The exhaustive analysis over several testcases indicates that the topology proposed by our algorithm is always the most optimal one.

REFERENCES

[1] P.F. Stelling, V.G. Oklobdzija, "Design strategies for the final adder in a parallel multiplier," in 29th *Asilomar Conference on Signals, Systems and Computers*, pp. 591-595, Vol. 1, 1995

[2] P.F. Stelling, V.G. Oklobdzija, "Design strategies for optimal hybrid final adders in a parallel multiplier," in *Journal of VLSI Signal Processing*, 14(3):321-31, 1996.

[3] C.S. Wallace, "A suggestion for a fast multiplier," in *IEEE Transactions on Electronic Computers*, EC-13(2):14-17, 1964.

[4] L. Dadda, "Some schemes for parallel multipliers," in *Alta Frequenza*, vol. 34, pp. 349–356, 1965.

[5] K.C. Bickerstaff, E.E. Swartzlander, M.J. Schulte, "Analysis of column compression multipliers," in *Proceedings of* 15th *IEEE Symposium on Computer Arithmetic*, pp. 33–39, 2001.

[6] R.P. Brent, H.T. Kung, "A regular layout for parallel adders," in *IEEE Transactions on Computers*, C-31(3):260-64, 1982.

[7] P.M. Kogge, H.S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," in *IEEE Transactions on Computers*, C-22(8):783-91, 1973.

[8] M. D. Ercegovac, T. Lang, "Digital Arithmetic," *The Morgan Kaufmann Series in Computer Architecture and Design*, 2003

[9] B. Parhami, "Computer Arithmetic, Algorithms and Hardware Designs," *The Oxford University Press*, 2000

[10] S. Palnitkar, "Verilog Hdl: A Guide to Digital Design and Synthesis," *Prentice Hall*, Upper Saddle River, NJ, 2003.