Routing Wire Optimization through Generic Synthesis on FPGA Carry Chains

Hadi Parandeh-Afshar[†]
hadi.parandehafshar@epfl.ch

Grace Zgheib[‡]
grace.zgheib@lau.edu.lb

Philip Brisk[§]
philip@cs.ucr.edu

Paolo Ienne[†]
paolo.ienne@epfl.ch

[†]Ecole Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences, 1015 Lausanne, Switzerland

[‡]Department of Electrical and Computer Engineering
Lebanese American University, Byblos, Lebanon

[§]Department of Computer Science and Engineering
University of California Riverside, 900 University Ave., Riverside CA92521, U.S.A.

ABSTRACT

FPGA logic clusters are comprised of look-up tables (LUTs) and arithmetic carry-chains, which perform specific arithmetic operations such as addition. In this paper, we present a generic logic synthesis technique to utilize such dedicated resources by restructuring the already mapped FPGA cicuits. The basic idea is to replace the interconnection wires between the blocks that are logically in a chain by the carry chains, which are hardwired connections. This reduces the pressure on the routing resources and minimizes the utilization of routing wires. We observed that, on average, more than 70% of logic nodes can be restructured and 39% of them form the logic chains that are mappable to the carry chains. Our synthesis approach comprises a fast and memory efficient Boolean matching technique for identifying restructurable nodes and a chaining heuristic to identify the logic chains. This approach can be used effectively in modern high performance FPGA families from both Altera and Xilinx. Our experiments indicate that, on average, 9% of routing wires are saved using this technique.

1. INTRODUCTION

Field programmable gate arrays (FPGAs) offer greater flexibility than application-specific integrated circuits (ASICs) in the form of post-fabrication configurability. Fully programmable logic blocks and interconnection network in FPGAs provide maximum flexibility; however, this flexibility comes at a cost of poor logic density, performance, and power consumption in comparison. In current FPGAs, on-chip silicon area is dominated by the programmable interconnect fabric. The fraction of silicon dedicated to programmable routing increases with each successive technology generation. This trend directly impacts the performance and power consumption of FPGAs. Moreover, the feasibility of synthesizing a circuit onto an FPGA can be limited by the availability of routing resources, rather than programmable logic.

To narrow the gap, FPGA designers have integrated dedicated non-programmable resources into the FPGA's fabric for commonly occurring operations. The most prevalent of these resources are carry chains for addition and subtraction, embedded multipliers, which have evolved into more complicated DSP blocks, and blockRAMs for dedicated storage. The logic cells employed in modern high-performance FPGAs have evolved from dedicated lookup tables (LUTs) to a combination of LUTs integrated with carry chains. The conventional wisdom has been that carry chains can only be used for addition/subtraction of two or three integers, but potentially they can be exploited for the connection of logic nodes that form a chain.

Figure 1 (a) illustrates how a chain of non arithmetic functions is implemented on a generic FPGA. Each function is mapped onto the LUT in the logic cell and the routing wires are utilized for the interconnections. Figure 1 (b) shows an alternative implementation, which is the key idea of this paper, for the same chain of functions. The idea is to restructure the original functions to be able to map them differently on the same logic cell, by employing the adders and carry chains. The key point of this alternative mapping is that the inter block routing wires in Figure 1 (a) are replaced by the dedicated carry chain wire. This will reduce the required routing resources.

In this paper, we first present a fast and memory efficient Boolean matching technique to specify whether a function can be restructured to be mapped on the FPGA logic cell using the dedicated adder and carry chain. This new configuration of the logic cell is a kind of macro cell: For the Boolean matching, we develop a library, by which we can represent all supported functions of the macro cell. By identifying regular and independent bit patterns, we reduce the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

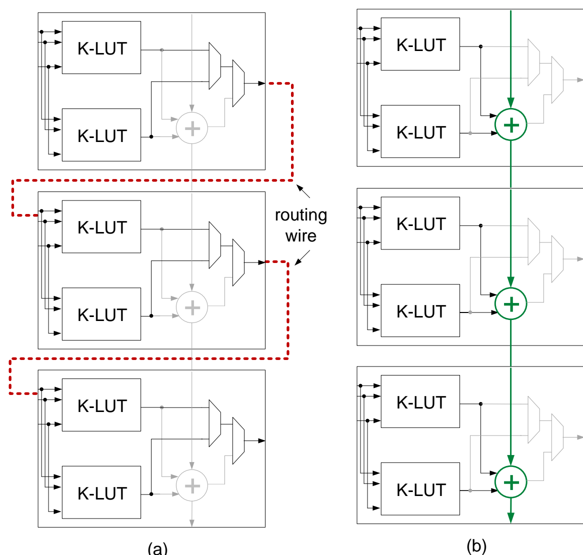

Figure 1: Key idea. (a) Conventional synthesis of logic chains on FPGAs. Routing wires are required for the logic interconnection. (b) Our alternative mapping of logic chains using the available carry logic and chains in logic cells. The interconnection wires are replaced by the hardwired carry chains.

library size significantly. Moreover, we present a chaining heuristic to find the logic chains that can be mapped to the macro cell with the objective of longer chains exploration. Then, we apply the proposed synthesis technique to FPGA circuits already mapped to optimize the required routing resources. Our experiments indicate that on average 9% less wire is utilized using our technique.

2. RELATED WORK

The traditional formulation of the technology mapping problem covers a DAG representing a structural HDL description of a circuit using K-cuts; this formulation assumes that an FPGA consists solely of LUTs, and does not consider carry chains, fracturable LUTs, embedded multipliers, or DSP blocks. Different objectives for this problem are possible: Cong and Ding proved that minimizing the number of LUTs on the longest path has a polynomial-time solution [3]; several others have proven that the decision problems corresponding to minimizing the total number of LUTs used in the covering and minimizing power consumption are NP-complete [5, 6]. Many heuristics to solve different variations of the technology mapping problem have been presented over the years—far too many to enumerate here.

Additionally, several papers have tried to perform logical decompositions to optimize the structural circuit description in conjunction with technology mapping [2, 4]; as logical optimization is NP-complete in the general case, this formulation of the problem is NP-complete as well, although the use of decomposition can significantly improve the quality of the technology mapping that can be achieved.

There have been a handful of papers that have successfully mapped operations other than 2- or 3-input addition/subtraction onto the carry chains of commercial FPGAs. In [8], the carry chains were used to map generalized parallel counters to implement multi-operand addition. However this approach targets a limited set of logic functions and cannot map general logic functions onto carry chains.

Similar to our work, the ChainMap attempts to map arbitrary logic functions onto the carry chain of the Altera Stratix and Cyclone FPGAs [7]; the carry chain in such FPGAs has a carry select structure. Due to its complex structure, this carry chain has been replaced with the widely used and simple ripple carry chain in current FPGAs in both Altera and Xilinx families. In contrast to our work, ChainMap is not usable for ripple carry chains that exist in recent FPGAs as the authors mention in their paper.

3. OUR SYNTHESIS APPROACH

The basic idea is to replace the inter block routing wires of the logic chains with the hardwired carry chains. Therefore, we take an FPGA mapped netlist as the input and try to replace the interconnection wires by the carry chains to reduce the pressure on routing network. The input netlist is atom level (device primitives), where each function is represented by a logic cell configured as a LUT. The first step is to verify whether a given logic node (originally targeted to a universal LUT) can be restructured to be mapped by the logic cell configuration in which the adder and carry chain are used and where only a limited number of logic functions can be mapped. We use Boolean matching for this purpose and we employ a truth table library, which represents all mappable functions. By making some observations on the structure, regularity, and segmentation of the truth table, the library size is reduced significantly; this makes the Boolean matching process memory efficient and fast. Once we found the mappable functions, we need to select the ones that form the chains. In addition to the restructuring possibility, the chain should go through the inputs of functions that have the chain-ability property. This is what that is specified in the Boolean matching process. Moreover, each function can not be placed into more than one chain and the chains should not intersect. The subsequent subsections present more details on the Boolean matching and chaining heuristic.

3.1 Boolean Logic Matching

The carry chain in current FPGAs has a ripple form. In general, there is an adder in the logic cell, which is driven by two LUTs and the hardwired carry output of the previous adder in the logic cell array. In this paper, we do our experiments on *Altera* FPGAs, but the same concept can be applied to *Xilinx* FPGAs. *Stratix II-V* FPGA series [1] are the state-of-the-art *Altera* FPGA devices; all have the same logic cell that is called *Adaptive Logic Module (ALM)*. Each ALM has eight inputs and can be configured to four different operation modes based on how the available resources are utilized. The ALM has a fracturable structure and the resources are organized into two identical sets that form the half-ALM. There is an adder in each half-ALM and it is utilized in two modes that are called *arithmetic* and *shared arithmetic* modes.

In these two modes, several inputs are shared between the LUTs of both half-ALMs, because in ripple carry addition, sum and carry of the same inputs are utilized in consecutive adders of ALM respectively. As the result, at most 6 inputs out of 8 available ALM inputs are utilized. However, the fracturable LUT structure of the ALM allows us to create

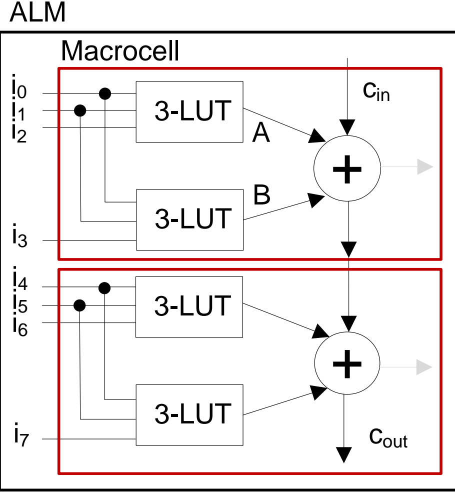

Figure 2: Proposed ALM configuration using the adder and the carry chain for generic logic synthesis. Each ALM can implement two chained 5-input functions with non-shared inputs.

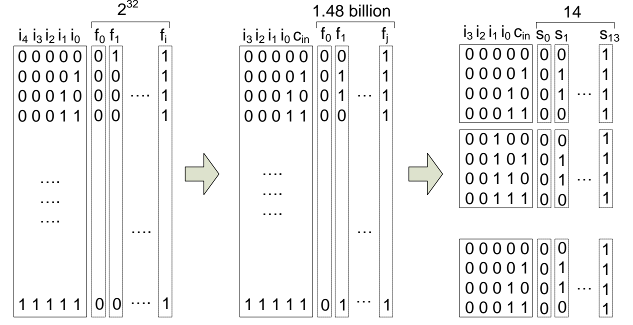

Figure 3: Reduction of the truth table of the macro cell by exploiting the regular and independent patterns.

a new adder based configuration mode, in which each half-ALM utilizes 4 distinct inputs and therefore all 8 inputs of ALM are used. Figure 2 shows this new ALM configuration mode that we use for mapping of generic logic. As shown in this figure, each half-ALM is configured as a 5-input macro cell, where the carry bit is the fifth input. Therefore, a subset of 5-input functions can be mapped to each half and as a whole, one ALM can be configured to implement two chained 5-input functions. Since the two 5-input functions do not have any input in common, we have more options to form the functions chains.

For the Boolean matching, we first need to develop a library, which represents the macro cell shown in Figure 2. This library is generated once and for any given function we search the library to find a match. The library maintains the truth table of all 5-input functions that are supported by the macro cell. The major problem with the Boolean matching approach is the size of the library, which grows exponentially with the number of inputs. This may require a huge memory space and will slow down the search time. The general case is a 5-LUT, which can implement 2^{32} functions, while this number is less for the 5-input macrocells, but it can be in the same order. However, we explored a number of regular patterns related to the macrocell of Figure 2, which allows to represent all implementable functions by few bit patterns. The two left tables in Figure 3 are the truth tables of the mappable functions on the 5-LUT and the 5-input macro cell. It is obvious that for the 5-LUT, we do not need to do logic matching and hence this table (left one) is not required. While for the macrocell, we need to match a given function truth table against all table entries to check if a match is found. This table is quite large and it requires huge memory space and the matching process is too slow. However, we will show that the truth tables of all supported functions can be represented by only 14 4-bit segments, where the the 32-bit mask of all 5-input functions are obtained by combining eight of these 4-bit patterns.

Table 1 lists the output values of the LUTs in Figure 2 with respect to all possible inputs combinations and the LUTs configurations. The output values of the two LUTs are shown with A and B symbols that can have any value depending on the LUTs configurations and the values of the LUTs inputs. As indicated, we can divide the table into four different segments, where in each the output values of LUTs are totally independent of the output values in any of the other three segments. Furthermore, comparing all four segments, we can also notice that they have identical patterns of values. To produce the truth table of the macro cell, it is sufficient to take one segment of this table to represent the two adder's inputs that are driven by the LUTs. Mixing these combinations with the carry input of the adder will give us an alphabet table, by which we can produce the dictionary of all supported functions.

Table 1: Truth table of LUTs in Figure 2 for different input combinations. The first LUT implements function A and the second one implements function B. The table can be divided into four independent sections with identical patterns.

i_3	i_2	i_1	i_0	LUT_1	LUT_2
0	0	0	0	A_0	B_0
0	0	0	1	A_0	B_1
0	0	1	0	A_1	B_0
0	0	1	1	A_1	B_1
0	1	0	0	A_2	B_2
0	1	0	1	A_2	B_3
0	1	1	0	A_3	B_2
0	1	1	1	A_3	B_3
1	0	0	0	A_4	B_4
1	0	0	1	A_4	B_5
1	0	1	0	A_5	B_4
1	0	1	1	A_5	B_5
1	1	0	0	A_6	B_6
1	1	0	1	A_6	B_7
1	1	1	0	A_7	B_6
1	1	1	1	A_7	B_7

Figure 4 shows some, but not all, 4-bit alphabets which are obtained by replacing 0 and 1 values for the carry input, A and B variables. Logically, we have 32 (2^5) 4-bit entries for the four output variables and the carry input, but due to the symmetry property of the adder carry output, several duplicate configurations are generated. Removing these duplicate entries, the table is reduced to fourteen 4-bit entries.

LUT_1	LUT_2	Fourteen 4-bit alphabets of configuration mask dictionary					
A_0	B_0	0	1	0	1	1	...
A_0	B_1	0	1	0	1	0	...
A_1	B_0	0	0	1	1	1	...
A_1	B_1	0	0	1	1	0	...

$A_0 = 0$	$A_0 = 1$	$A_0 = 0$	$A_0 = 1$	$A_0 = 0$
$A_1 = 0$	$A_1 = 0$	$A_1 = 1$	$A_1 = 1$	$A_1 = 0$
$B_0 = 0$	$B_0 = 0$	$B_0 = 0$	$B_0 = 0$	$B_0 = 1$
$B_1 = 0$	$B_1 = 0$	$B_1 = 0$	$B_1 = 0$	$B_1 = 0$
$C_{in} = 0$	$C_{in} = 1$	$C_{in} = 1$	$C_{in} = 1$	$C_{in} = 1$

Figure 4: Macro cell Boolean matching alphabet library. Each column indicates a 4-bit alphabet; each 32-bit combination of these alphabets is a truth table that corresponds to a supported function of macro cell in Figure 2.

Having this alphabet table, the truth table (dictionary) of all supported functions by the macro cell can be produced by putting any eight combination of these 4-bit alphabets together. Theoretically, we have 14^8(1.48 billion) combinations, but we only need to save these 14 entries in the library.

To verify if a given function is mappable to the macro cell, its configuration mask is divided into eight 4-bit segments and each segment is independently compared against the 4-bit alphabet in the library. If a match is found for all eight segments, then it is guaranteed that the given function can be mapped to the macro cell. Note that the configuration mask can change if the order of function's inputs is changed. So it is important to repeat the mentioned process for all input permutations. Apart from being mappable to the the macro cell, it is important to identify which inputs can be placed on the carry chain. Having more eligible inputs increases the chance of finding more and longer chains of logic nodes. Therefore, even if we find a match for a specific input permutation, we continue the process of Boolean matching for all input permutations to find all input candidates.

ALM in its normal operation mode can be configured to LUTs up to 6 inputs and we can have up to 6-input nodes in the input netlist. But the above Boolean matching technique only supports nodes up to 5 inputs. Since we have a considerable number of nodes that have 6 inputs, we need to find a way to restructure 6-input functions as well. This is important as many of these 6-input nodes are placed in between of the other nodes and if no carry chain based mapping exist for them, we loose many chaining opportunities. Our solution is to decompose each 6-input function into two smaller ones and check if the resulting structure can be mapped to two chained macro cells within one ALM using the same technique.

Our decomposition approach is simple. We need to find a divisor function D such that $F = G(D(a, b), b, c)$, where b is the common variable between G and D. With this decomposition, D and G are the mapping candidates to the upper and lower macro cells of Figure 2, respectively. Since D has to be mapped to the top macro cell and G to the bottom one, it is required that D not to have more than 5 and G not to have more than 4 inputs. F is mappable to the macro cells pair, if (1) both D and G are mappable to the macro cell, (2) there is an input in D, which can be placed on the carry chain and is not in the on-set of G, and (3) the G input, which is the D output can be placed on the chain.

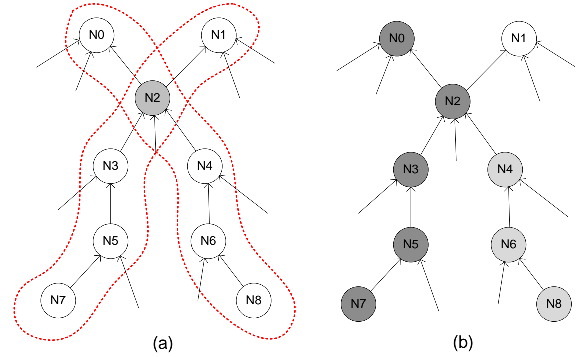

Figure 5: (a) Two chains intersecting at a shared node. (b) The shared node is assigned to one of the chains, breaking the other chain into two smaller sub-chains.

```
1: Chaining_Heuristic(pDAG) {
2:   while(!termination_condition) {
3:     for(i=0 to nDAGOutputs) {
4:       Find_Node_Depth_Rec(pDAG->out[i])
5:     sort_chains();
6:     mark_nodes_in_longest_chain();
7:   }
8: }
//-----------------------------------------------------------------------
1: int Find_Node_Depth_Rec(pNode) {
2:    for(i = 0 to nLeaves-1) {
3:       if(pNode->Leaves[i] == DAGInput) {
4:          pNode->depth[i] = 0;
5:          break; // go to the next leaf
6:       }
7:       else
8:          pTmpNode = pNode->Leaves[i];
9:
10:      depth = Find_Node_Depth_Rec (pTmpNode );
11:      if(pNode->chainable)
12:         pNode.depth[i] = depth + 1;
13:      else
14:         pNode.depth[i] = 0;
15:   } // end of for loop
16:   pNode->max_depth = pNode->Find_Max_Depth();
17:   return  pNode->max_depth;
18: }
```

Figure 6: Pseudo-code of the Chaining Heuristic.

To find the proper D, for each input of F, we first find the product terms that are not dependent on that input. Then the corresponding G will be constructed by first XORing F and D and then adding the product terms of D to G as don't cares.

3.2 Chaining Heuristic

The objective of the mapping heuristic is to identify chains of logic having the maximum possible lengths. The input is a *Direct Acyclic Graph(DAG)*, where each node represents a logic function and each edge represents the input and output dependencies among the functions. The DAG is generated after technology mapping, so each node is a prospective function that can map onto the LUTs. The number of inputs (K) of each node in the DAG cannot exceed the number of LUT inputs; each node has K child nodes and one or more parents based on the fanout of the node output.

The mapping heuristic visits the DAG nodes in *Depth First Search (DFS)* order, starting from the outputs and

working back toward the inputs. The heuristic recursively assigns a depth to each node in the DAG.

DEFINITION 3.1. *A node is* ***chainable****, if it is mappable to proposed ALM configuration, its fanout is one and is not part of another chain.*

DEFINITION 3.2. *The* ***depth*** *of an input node is zero if none of its inputs are chainable; otherwise the depth is the number of chainable nodes that can be accessed consecutively through that input node; the* ***depth*** *of an internal node is the maximum depth among all input nodes from which it is reachable.*

It is not possible to have fanout in the middle of the logic chains and this is a precondition to form the chains. Once a depth has been assigned to all nodes, then we can select specific nodes to map onto the logic chain. In particular, we search for the longest chain of nodes in the DAG. This chain is then mapped onto a carry chain in the FPGA.

Figure 5 (a) provides a simple example. In this figure, there are two chain candidates, each having a length of 5; the chains intersect at node N2. Since each node can be part of one chain, N2 is arbitrarily chosen for one of the chains. As shown in Figure 5 (b), the second chain is then broken into a chain of 3 nodes, and a singleton node, which itself is not part of a chain.

Figure 6 presents the chaining algorithm. The loop in the main function recursively traverses the DAG using the DFS, starting from the outputs, and computes the depth of each node. The depth information then allows the heuristic to identify the logic chains using the *sort_chains* function. The node with the highest depth is selected as the head of the current chain. All of the nodes in the selected chain are marked as *CHAINED*, to avoid placing a node in more than one chain. This process repeats until the maximum length of all remaining chains is less than some threshold value. The time complexity of the heuristic is $O(nh)$, where n is the number of nodes in the DAG and h is the depth of the DAG.

4. TOOL CHAIN FLOW

Figure 7 presents the tool chain flow that we use for our experiments. First, we synthesize each benchmark using *Quartus-II*, a commercial tool provided by Altera; Quartus-II generates a *Verilog Quartus Mapping (VQM)* file in ASCII text, which contains a node-level (or atom-level) netlist. Then the VQM netlist is parsed and a DAG is generated. Each node in the DAG corresponds to an FPGA cell in the VQM netlist and the edges between nodes in the DAG represent data dependencies. This DAG will be the input to our synthesis engine; Boolean logic matching and chaining of the nodes is performed in this step and the modified netlist is written as a new Verilog file. In terms of logic, the Verilog netlist is equivalent to the original VQM nelist; however, the mapping of some cells have been changed. Lastly, the new netlist is placed and routed by Quartus-II targeting an Altera Stratix-III FPGA to obtain accurate estimates of the area and critical path delay. Our area metric includes the number of logic blocks that are used, and also the amount of local and global routing resources required to realize the circuit. Since we just restructure the nodes in the netlist of already mapped circuit, the number of logic blocks is not changed.

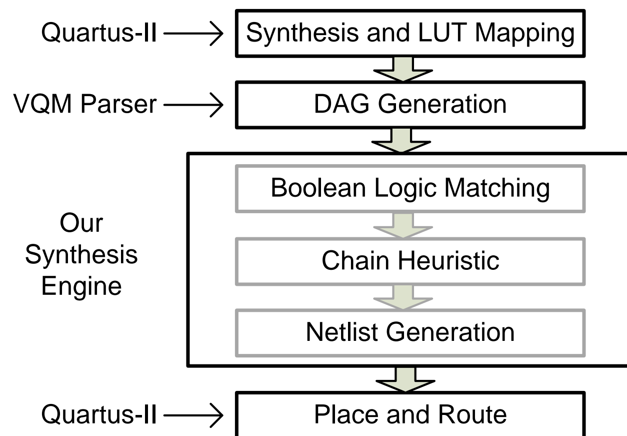

Figure 7: Tool chain flow used for the experiments.

Table 2: Chaining heuristic statics.

Benchmark	Chainable	Chained	MC	AC
alu4	74%	39%	4	3.5
pdc	69%	35%	6	3.9
misex3	68%	42%	4	3.1
ex1010	71%	41%	5	3.4
ex5p	72%	40%	4	3.5
des	65%	31%	3	3.0
apex2	73%	42%	4	3.6
apex4	75%	39%	4	3.7
spla	72%	43%	6	4.2
seq	69%	38%	4	3.4
Average	70%	39%	4.4	3.5

5. EXPERIMENTAL RESULTS

We used the MCNC benchmarks for our experiments and selected the combinational benchmarks exclusively. Effectively synthesizing the sequential benchmarks would involve the integration of our mapping algorithm with some amount of retiming, e.g., to keep registers off of the logic chains, which is an interesting challenge, but for now left open for future work.

Altera's ALM can be configured as two 6-LUTs with identical functions; this allows to implement a small subset of 7-input logic functions. We measured the distribution of the functions in terms of the number of inputs for different benchmarks. The majority of the functions, 72%, have 5 or fewer inputs, and less than 0.5% have 7 inputs.

Table 2 summarizes the statistics relating to the chaining heuristic. The column labeled "Chainable" reports the percentage of functions that can be mapped onto the macro-cell of Figure 2. The column labeled "Chained" reports the percentage of eligible functions that actually form logic chains that are mapped onto the arithmetic carry chains. On average, 39% of all logic functions are "Chained." The last two columns show the maximum and average chain lengths for each benchmark, respectively.

We use the "Aggressive Routing Optimization" option for Quartus-II in all of our experiments. The mapped circuit is optimized for delay, and our synthesis technique attempts to re-optimize it to reduce the routing resource usage by mapping compatible logic chains onto the carry chains.

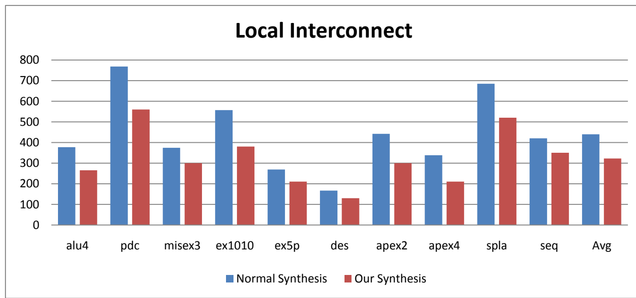

Figure 8: The number of local interconnection wires (i.e., within a LAB) used for each benchmark. On average, the introduction of the logic chain reduces the number of local wires used by 26%.

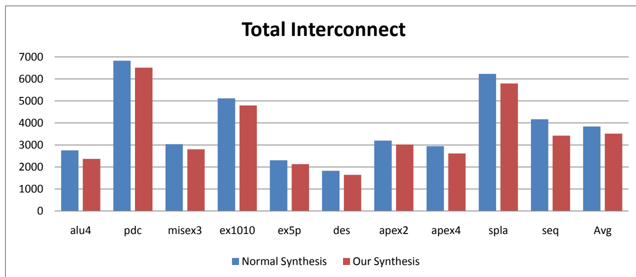

Figure 9: The number of global and local interconnection wires used for each benchmark, scaled by the length of the wires. On average, the introduction of the logic chain reduces the total number of wires used by 9%.

Figures 8 and 9 report the reduction in routing resource usage achieved by our technique. Local (intra-LAB) wiring is reduced by 26%, with a maximum savings of 38% for apex4. On average, we reduce the number of local and global wires used by 9%, with local wires contributing a 2% overall reducing. The total wiring savings ranged from 5% (pdc) to 17% (seq). The results reported in Figure 9 are scaled to account for different horizontal and vertical wire lengths.

We measured the impact of our chaining approach on critical path delay. After optimization for routing, we observe a marginal increase of less than 3%, on average, as reported by Figure 10. The main contributing factors to the delay increase are the delay between the ALM inputs and the adder outputs, and the delay between the sum-input and carry-out of each full adder in the chain. It seems quite likely that the designers never intended for circuits to enter and exit the carry chains at arbitrary points, and as such, the full adders in the chain have been thoroughly optimized for carry-in to carry-out delay, rather than sum-in to carry-out delay. Consequently, re-engineering the carry chains to be more favorable to generalized mapping algorithms could likely eliminate much of the overhead reported in Figure 10.

Nevertheless, we do believe that more aggressive synthesis algorithms could also reduce delay. In particular, it would require new logic decomposition algorithm that recognizes the cascaded structure of the logic chain; such an algorithm could be integrated with a technology mapper. to make better use of the logic chains than the relatively naive and greedy chaining heuristics described here.

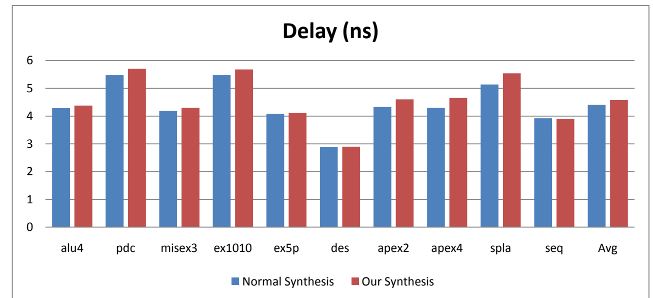

Figure 10: Critical path delay of each benchmark; on average, the delay is increased by 3% due to the big delay between ALM inputs and the adder outputs and also the adder outputs and ALM outputs.

Lastly, the routing wire optimization does not change the required number of logic blocks or ALMs, because we only restructure some of the mapped nodes and fit them into the same number of ALMs. The smallest benchmark is *des*, which has 300 ALMs and the biggest are *pdc* and *spla* with approximately 1400 ALMs.

6. CONCLUSION

This paper proposes to use carry chains for the synthesis of generic logic as a way to improve routing resource utilization in high-performance FPGAs. The dedicated wires between logic cells in the chain reduce pressure on the routing network. This idea is based on the observation that many technology mapped circuits contain linear chains of LUTs after mapping; the basic idea is to restructure the mapped logic nodes to be able to exploit the dedicated adders and carry chains for their implementation. We present a very fast and memory efficient Boolean logic matching technique along with a chaining heuristic to optimize routing resources of the FPGA mapped circuits.

Our experimental results shows that the proposed synthesis approach can reduce the total number of routing resources required by 9%. This reduction impacts the delay slightly with the increase of 3% on average. However, the number of logic cells does not change.

In this paper, we simply searched for logic chain candidates in a netlist that were produced by technology mapper that was unaware of the logic chains. We believe that more chains can be found, and that pressure on the routing network reduced further through the development of logic decomposition and technology mapping algorithms that are aware of and can exploit the logic chains.

7. REFERENCES

[1] Altera Inc. *Stratix II, III, and IV device handbooks.* Available online: http://www.altera.com/.

[2] G. Chen and J. Cong. Simultaneous logic decomposition with technology mapping in FPGA designs. *in Internation Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 48–55, Feb 2001.

[3] J. Cong and Y. Ding. FlowMap: an optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs. *in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(1):1–12, Jan 1994.

[4] T. S. Czajkowski and S. D. Brown. Functionally linear decomposition and synthesis of logic circuits for FPGAs. *in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(27):2236–2249, Dec 2008.

[5] A. H. Farrahi and M. Sarrafzadeh. Complexity of the lookup-tale minimization problem for FPGA technology mapping. *in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(11):1319–1332, Nov 1994.

[6] A. H. Farrahi and M. Sarrafzadeh. FPGA technology mapping for power minimization. *in 4th Internation Workshop on Field-Prog. Logic and Applications*, pages 66–67, Sep 1994.

[7] M. T. Frederick and A. K. Somani. Beyond the arithmetic constraint: depth-optimal mapping of logic chains in LUT-based FPGAs. *in Internation Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 37–46, Feb 2008.

[8] H. Parandeh-Afshar, P. Brisk, and P. Ienne. Exploiting fast carry-chains of FPGAs for designing compressor trees. In *Proceedings of the 19th International Conference on Field-Programmable Logic and Applications*, pages 242–49, Prague, Aug. 2009.

[9] Xilinx Inc. *Virtex-5 User Guide.* `http://www.xilinx.com/`.