Timing Driven Functional Decomposition for FPGAs

Valavan Manohararajah, Deshanand P. Singh and Stephen D. Brown
Altera Toronto Technology Center
151 Bloor St. West, Suite 200
Toronto, Ontario CANADA
vmanohar|dsingh|sbrown@altera.com

ABSTRACT

This work explores the effect of adding a timing driven functional decomposition step to the traditional field programmable gate array (FPGA) CAD flow. Once placement has completed, alternative decompositions of the logic on the critical path are examined for potential delay improvements. The placed circuit is then modified to use the best decompositions found. Any placement illegalities introduced by the new decompositions are resolved by an incremental placement step. Experiments conducted on Altera's Stratix and Stratix II device families indicate that this functional decomposition technique can provide average performance improvements of 6.1% and 5.6% on a large set of industrial designs, respectively.

1. INTRODUCTION

Recent research effort in *physical synthesis* has strived to eliminate the artificial separation that exists between the various steps in CAD. Most of the existing work is applicable to ASIC CAD flows [1, 3, 4, 5, 6, 7]. However, a few of the more recent efforts explore FPGA CAD flows [2, 8, 9]. Some have concentrated on making the synthesis step more aware of what happens during placement and routing [1, 2], while others have explored the use of synthesis type algorithms during placement and routing [3, 4, 5, 6, 7, 8, 9]. Our work falls into the latter category. It considers the effect of a functional decomposition algorithm that is used after placement.

Most of the delays in an FPGA circuit are due to the programmable routing network [10]. These delays will not be known for certain until the routing step completes. It would be advantageous to perform local optimizations once routing has completed and accurate routing delays are available. However, making changes to the circuit during the routing step is extremely complicated. Here we choose to perform local optimizations at the placement step, which is sufficiently close to the routing step that reasonably accurate delays are known. Furthermore, small changes to the circuit can still be made without much difficulty.

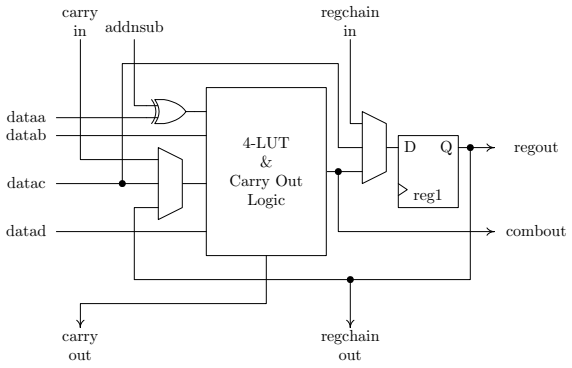

Figure 2: The Stratix logic element.

2. TARGET FPGA ARCHITECTURES

Altera's Stratix [11] and Stratix II [12] chips were used as the target for the functional decomposition experiments. As illustrated in Figure 1, the high level structure of both chips is similar. Both chips are comprised of I/O elements (IOEs), logic array blocks (LABs), digital signal processing blocks (DSPs) and memory elements (M512, M4K and M-RAM). While DSPs and memory elements perform very specific roles in the FPGA, the LABS can be configured to perform arbitrary logic functions. The LABs are also the source of significant differences between the two architectures. A LAB in a Stratix device contains 10 logic elements (LEs) while a LAB in a Stratix II device contains 8 adaptive logic modules (ALMs). The Stratix LE, illustrated in Figure 2, contains a four-input lookup table (4-LUT), a register and some logic that facilitates the creation of arithmetic circuits. Figure 3 illustrates the Stratix II ALM. It contains two registers, two sets of addition circuitry and a combinational logic module that can implement two functions of varying complexity. The combinational logic module can be configured to implement a single 6-LUT, or two LUTs with five or fewer inputs. If the module is configured to implement two 5-LUTs, the LUTs must share at least two of their inputs as there are only 8 inputs connected to the module.

Our work focuses on the LUTs within the LEs and ALMs that make up a circuit. We improve the overall timing of the circuit by restructuring the LUTs on the critical path.

.

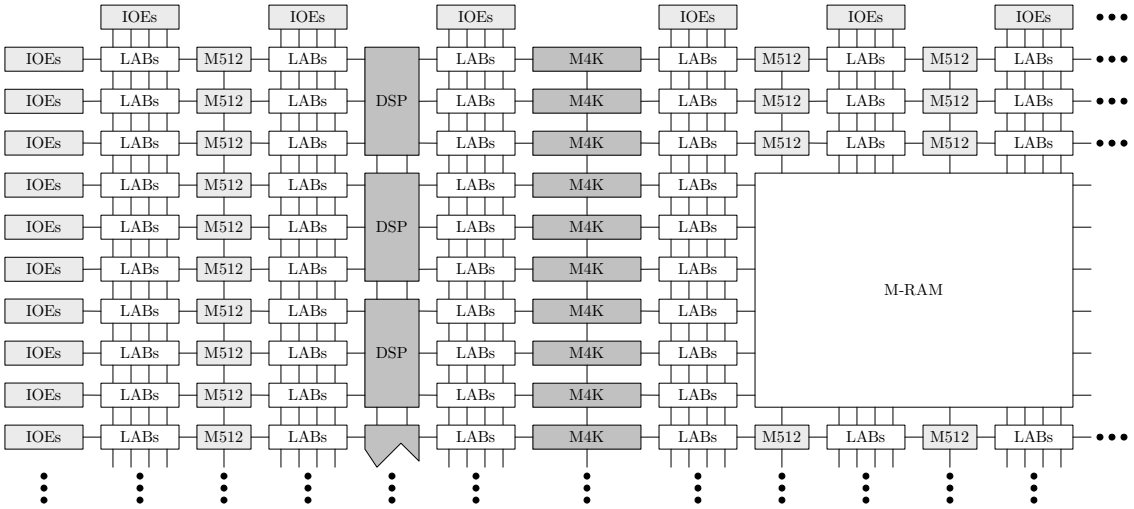

Figure 1: Structure of the Stratix and Stratix II FPGA architectures.

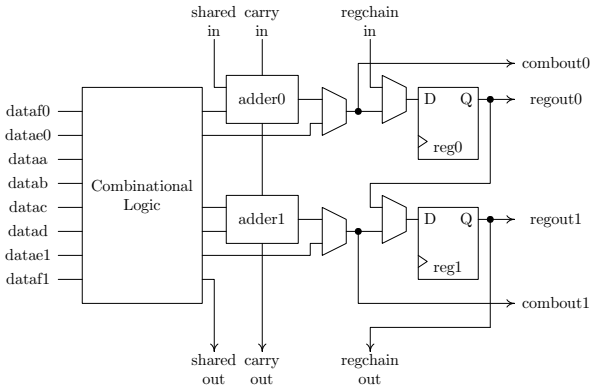

Figure 3: The Stratix II adaptive logic module.

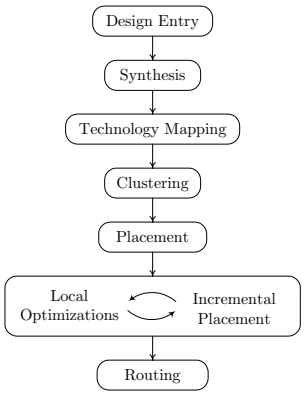

Figure 4: Post-placement optimizations in the CAD flow.

3. A FRAMEWORK FOR POST-PLACEMENT OPTIMIZATION

Figure 4 illustrates the CAD flow used in our work. In the first step, design entry, the design is described in terms of a hardware description language such as VHDL or Verilog. Logic synthesis optimizes the circuit obtained from design entry. During logic synthesis the netlist is represented in terms of a generic gate library. The technology mapping stage converts the netlist to use the logic elements available in the target FPGA architecture. In order to reduce the size of the problem the placer has to deal with, a clustering step is used to group the technology mapped circuit into a set of clusters. In both the Stratix and Stratix II architectures, the clustering step creates a set of LABs. Following clustering, placement determines a position for each cluster in the circuit.

Once placement is completed, various local optimization techniques are used to improve the circuit's critical path. Functional decomposition is one of the many local optimization techniques that can be used during this step. There are three other local optimization techniques present in Quartus II v4.2: retiming, logic replication and rewiring [13]. These techniques may make a circuit modification that results in an invalid placement. For example, a functional decomposition algorithm may create new wires that violate the constraint on the number of wires entering a LAB. A logic replication algorithm may create new LEs or ALMs which would then require placement. Incremental placement is used to integrate the modifications made by the local optimization techniques into the existing placement. It uses an iterative improvement algorithm whose goal is to integrate the circuit modifications while peturbing the existing placement as little as possible. Logic elements that have been added to the circuit and logic elements within illegal LABs are moved according to a cost function that consists of three components:

- *LAB illegality cost*: Each LAB is penalized if it contains architectural violations. The cost is proportional to the number of constraints violated.
- *Timing cost*: The timing cost is used to ensure that critical logic elements are not moved into locations that would drastically increase the critical path delay.
- *Wirelength cost*: Wirelength estimation is used to ensure that the circuit is easily routable after the logic element moves.

The total cost is a weighted sum of these components. Cost lowering moves are made until no further illegalities exist in the placement. A novel hillclimbing strategy is used to ensure that the iterative improvement algorithm does not get stuck in a local optima where none of the proposed moves seem to improve the cost even though there is remaining illegality. The interested reader is referred to [14] for a detailed description of the incremental placement step.

The final step in the CAD flow, routing, determines the wires that will be used to connect the elements that make up the circuit.

4. TIMING DRIVEN FUNCTIONAL DECOMPOSITION

4.1 Preliminaries and Prior Work

If the delay within circuit components and the delay of connections between circuit components are known, timing analysis can be used to establish the *slack* [15] of every connection. The slack of a connection is defined to be the amount of delay that can be added to the connection before it becomes *critical*. A connection is critical if the length of a path it belongs to exceeds the path-length constraint set by the user. Timing analysis also establishes a *slack ratio* for each connection. The slack ratio is a value between 0 and 1 which indicates the relative importance of each connection to overall circuit timing. Connections that have a significant effect on circuit timing have slack ratios closer to 0 while connections that have negligible effect on circuit timing have slack ratios closer to 1. A precise definition of slack ratios is beyond the scope of this paper. However, from an optimization prespective, slack ratios provide the most accurate information as the formulation accounts for multi-cycle clocks, inverted clocks and clock skew.

Given a function $f(X, Y)$ defined over two sets of variables X and Y, functional decomposition finds subfunctions

$$g_1(Y), g_2(Y), \ldots, g_k(Y)$$

such that f can be reexpressed in terms of the gs:

$$f(X, g_1(Y), g_2(Y), \ldots, g_k(Y))$$

The set of variables X is referred to as the *free set* and the set of variables Y is referred to as the *bound set*. If there are no variables common to X and Y, the decomposition is said to be *disjoint*. Otherwise the decomposition is *non-disjoint*. In this work, we limit the type of non-disjoint decompositions considered. During non-disjoint decompositions, the the number of variables common to X and Y is limited to 1. Unconstrained non-disjoint decomposition had little observed benefit over the constrained non-disjoint decomposition we consider here.

The LUTs in an FPGA are capable of implementing any function of k variables. Thus, functional decomposition (as opposed to algebraic decomposition) can be used to find subfunctions that fit naturally into LUTs. A number of recent works have explored the use of functional decomposition for FPGA logic synthesis [21, 22, 23].

Early methods for functional decomposition [16, 17] used data structures that were exponential in the number of variables present in the function being decomposed. Newer methods [19] are based exclusively on the BDD data structure. Although BDDs are exponential in the worst case, most of the functions encountered in practice can be represented compactly as BDDs.

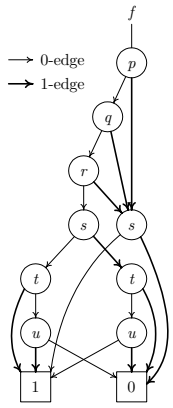

Figure 5: A reduced, ordered BDD for the function $f = (p+q+r)\overline{d} + \overline{(p+q+r)}(\overline{s}t + \overline{s}u + s\overline{t}\overline{u})$.

A BDD is a directed acyclic graph that contains two types of vertices, *non-terminal* and *terminal*. Non-terminal vertices are associated with variables and terminal vertices are associated with boolean constants. Every non-terminal vertex has two outgoing edges: a 0-edge representing the result of assigning 0 to the variable at the vertex and a 1-edge representing the result of assigning 1 to the variable at the vertex. Most of the benefits of BDDs cannot be realized unless the BDD is reduced and ordered. A BDD is reduced if it does not contain redundant vertices or multiple vertices implementing the same function. A vertex is redundant if both the 0- and the 1-edge point to the same destination vertex. Two vertices implement the same function if they are associated with identical variables and their 0- and 1-edges point to the same vertices. A BDD is ordered according to a specified variable order if a vertex associated with variable u points to a vertex associated with variable v only if v follows u in the specified order. In this work, the term BDD will always refer to a reduced, ordered BDD unless otherwise specified.

Figure 5 illustrates a reduced, alphabetically ordered BDD for the function

$$f = (p+q+r)\overline{s} + \overline{(p+q+r)}(\overline{s}t + \overline{s}u + s\overline{t}\overline{u})$$

Clearly, this function has a disjoint decomposition with bound set $\{p, q, r\}$ and free set $\{s, t, u\}$. A subfunction, $g = p+q+r$, defined over the bound set can be extracted from f, and f can be reexpressed in terms of g:

$$f = g\overline{s} + \overline{g}(\overline{s}t + \overline{s}u + s\overline{t}\overline{u})$$

This decomposition can be obtained from the BDD as follows. A *cut* in the BDD establishes two sets of variables. The variables above the cut constitute the bound set and the variables below the cut constitute the free set. Figure 6 illustrates a cut in f that separates the bound set, $\{p, q, r\}$, from the free set, $\{s, t, u\}$. The portion of the BDD above the cut references two distinct functions, f_0 and f_1, below the cut. Thus, the portion of the BDD above the cut can be replaced by a single boolean variable g that determines whether f_0 or f_1 is to be selected. A separate BDD computes the value for g, and in the new BDD for f, f_0 is selected when $g = 0$ and f_1 is selected when $g = 1$. Note that this

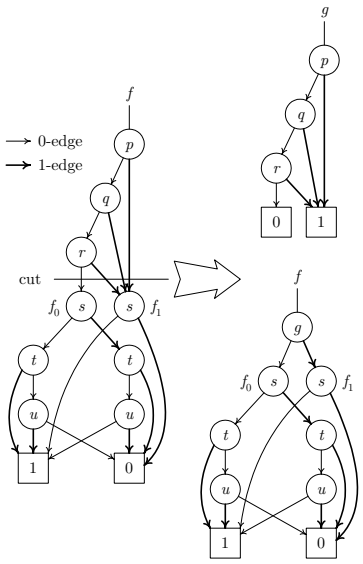

Figure 6: Decomposition of f using bound set $\{p, q, r\}$ and free set $\{s, t, u\}$.

encoding is abitrary. We could have just as easily selected f_0 when $g = 1$ and f_1 when $g = 0$.

When there are more than two distinct functions being referenced below a cut, a single variable is not adequate to encode the information needed to select one of the functions. If there are n distinct functions below the cut then $\lceil \log_2 n \rceil$ variables will be needed to encode the selection information.

Some common functions can only be simplified using non-disjoint decomposition. For example, a 4-to-1 multiplexer with data inputs $\{a, b, c, d\}$ and selection inputs $\{s_0, s_1\}$ cannot be implemented with two 4-LUTs unless non-disjoint decomposition is used. The method used for disjoint decomposition can be extended to handle non-disjoint decompositions as well. Figure 7 illustrates the BDD for the multiplexer function. We decompose the function using the bound set $\{s_0, s_1, a, b\}$ and the free set $\{s_0, c, d\}$. Non-disjoint decomposition with a single shared variable can be viewed as two disjoint decompositions, one where the shared variable has been set to zero and another where the shared variable has been set to one. The non-disjoint decomposition is then formed by combining the two disjoint decompositions using the shared variable. The two specialized functions of m are illustrated in Figures 7a and 7b. Both functions are decomposed using the bound set $\{s_1, a, b\}$. Finally, the non-disjoint decomposition in Figure 7c is created by using s_0 to select between the two disjoint decompositions. Observe that the resulting non-disjoint decomposition allows the function to be implemented using two 4-LUTs.

4.2 Algorithm Overview

An overview of the timing driven functional decomposition algorithm is given in Figure 8. Timing analysis is the first major task performed by the algorithm. The delays within circuit components are known at this point. However, the delays of connections between circuit components are not yet known as routing has not been performed. We estimate the delay between components as the delay of the fastest route between the component locations. Although

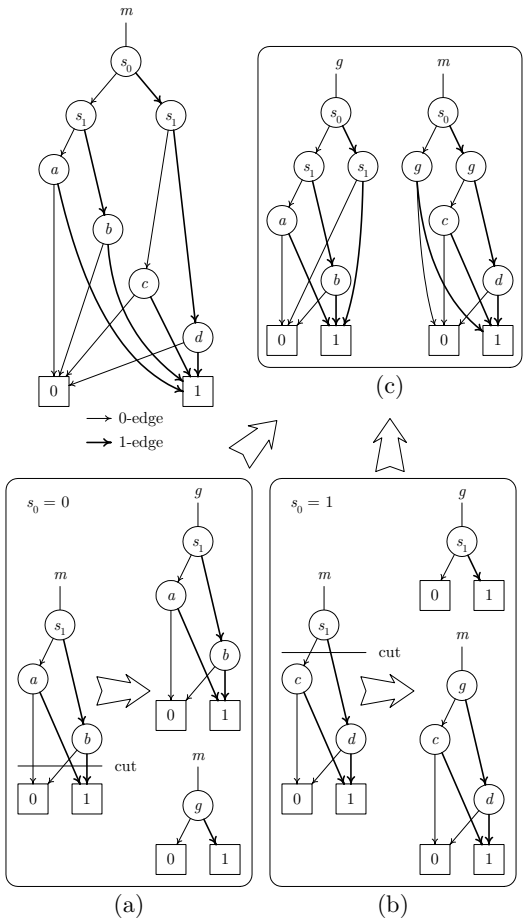

Figure 7: Non-disjoint decomposition of a 4-to-1 multiplexer.

```
1   bestt ← TimingAnalysis()
2   bestc ← Circuit
3   InitParams(params)
4   for   i ← 1 upto MaxIterations
5         S ← SelectSeeds(params)
6         S ← Sort(S)
7         for   s ∈ S
8               C ← ExpandCone(s)
9               C′ ← DecomposeCone(C, params)
10              if   C′ ≠ ∅
11                    Circuit ← (Circuit − C) ∪ C′
12              end if
13        end for
14        IncrementalPlacement()
15        t ← TimingAnalysis()
16        if    t ≥ bestt
17              bestt ← t
18              bestc ← Circuit
19        end if
20        AdjustParams(params)
21  end for
22  Circuit ← bestc
```

Figure 8: An overview of the functional decomposition algorithm.

the assumption of the fastest route is inaccurate for non-critical signals, it is quite accurate for critical and near-critical connections which tend to use fastest possible routes.

Timing analysis establishes both the slack and slack ratio of every circuit connection, and returns the worst slack seen during the analysis. The algorithm uses the worst slack to determine whether the timing driven decompositions help improve the overall timing of the circuit.

The algorithm performs a number of decomposition iterations until a user specified maximum is reached. A set of parameters, *params*, controls how some of the functions within an iteration behave. This set is initialized before the iterations begin and is adjusted at the end of every iteration. Each iteration begins with the selection of *seed LUTs* (line 4). Seed LUTs are the LUTs in the circuit that pass some criticality criteria (to be defined Section 4.3). Each seed LUT is assigned a cost that reflects the potential benefits of performing a timing driven decomposition around the LUT. The seed LUTs are sorted according to cost (line 5) so that the LUTs with a higher potential for improvement are considered first. This is necessary because a decomposition of the logic around one seed LUT may affect the availability of a decomposition around another seed LUT. Thus, by considering the most beneficial decompositions first, we avoid the possibility that a lower valued decomposition prevents the discovery of a higher valued one. Seed LUTs are used as the starting point for a cone expansion procedure (line 7) which returns a cone rooted at the seed LUT. The returned cone is then decomposed (line 8), and if the decomposition is successful the restructured cone replaces the original cone in the circuit. Once all seed LUTs have been considered, incremental placement (line 13) is called to integrate the changes made by the decomposition algorithm into a legal placement. A timing analysis is then performed to see if the circuit's timing has improved. The algorithm stores the circuit with the best timing in *bestc*, and at the end, the best circuit replaces the circuit being restructured.

We consider the procedures for seed selection, cone expansion and cone decomposition in greater detail below. We also consider the iteration parameters that control the behavior of seed selection and cone decomposition.

4.3 Selecting Seed LUTs

The selection of seed LUTs depends on the current iteration's *slack ratio threshold* parameter (see Section 4.6). The slack ratio threshold determines which LUTs are to be included in the set of seed LUTs. Any LUT with an input whose slack ratio is below the slack ratio threshold is added to the set of seed LUTs. The cost of a seed LUT is defined based on the potential benefits of performing a functional decomposition at the LUT. We define the cost, c, of a seed LUT as

$$c_{LUT} = (1.0 - r_{min})^{\alpha}(r_{avg} - r_{min}) \quad (1)$$

where r_{min} is the minimum of the input slack ratios and r_{avg} is the average of the input slack ratios. The exponent, α ($\alpha = 4$ in the experiments), is used to control which of the two components of cost has a higher effect on the final cost. Higher values of cost indicate that the potential for improvement is higher. A LUT is assigned a high cost if it has a very low minimum slack ratio or if it has an average slack ratio which is significantly higher than the minimum slack ratio. If the minimum slack ratio is low then the potential improvements at the LUT benefits the entire circuit. If the average slack ratio is high relative to the minimum slack ratio then a decomposition at the LUT has a greater chance of succeeding; the connections with high slack ratios can be slowed down in order to speed up the connections with lower slack ratios.

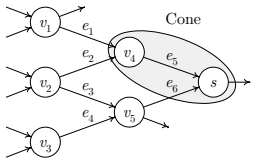

Figure 9: A cone being expanded around s.

4.4 Cone Expansion

The cone expansion process begins at a seed LUT and proceeds towards the *primary inputs*. For the purposes of functional decomposition, any circuit component that is not a LUT is considered to be a primary input. The first LUT added to the cone is the seed LUT itself. Then the cone is grown by adding LUTs that provide inputs to the cone. As the cone is grown, we update the slack of the inputs to the cone. Slacks are computed under the assumption that there is a large LUT implementing the functionality of the cone at the same position as the seed LUT. For example, in Figure 9, a cone is being grown with LUT s as the seed. The new slack of e_1, $s'(e_1)$, is given by

$$s'(e_1) = s(e_1) + d(v_4) + d(v_4, s) - d(v_1, s)$$

where $s(e_i)$ is the slack of e_i as computed by timing analysis, $d(v_i)$ is the delay through LUT v_i and $d(v_i, v_j)$ is the delay of the fastest route from LUT v_i to LUT v_j. LUTs v_1, v_2 and v_5 provide the current inputs to the cone, and one of these LUTs will be the next to be selected for inclusion. The LUT driving the cone input with the lowest slack will be the first to be considered for inclusion. This LUT is added if it passes one of two criteria:

1. All of the LUT outputs must be present as inputs to the cone, or

2. an input that the LUT provides to the cone must have a slack that is sufficiently close to the minimum slack observed at the seed LUT.

If the LUT does not pass either of the two criteria, the next LUT, in order of slack, will be considered for inclusion. Expansion stops if there are no more LUTs that pass the inclusion criteria.

The first criterion avoids cases where a LUT needs to be duplicated in order to be added to the cone. For example, in Figure 9, only one of v_5's outputs is present as an input to the cone. If the LUT is to be added to the cone, it needs to be duplicated. The second criterion allows the temporary violation of the first criterion for the LUTs that have a significant impact on the timing of the cone. Returning to our example, assume that e_5 has the lowest slack at s and that e_6 has a slack close to e_5's. Cone expansion has expanded v_4, and if the second criterion is not present, the expansion stops at this point. However, resynthesis of the cone containing v_4 and s will do little to improve the overall slack at the seed LUT because e_6 will still be present as an input. The second criterion allows the addition of v_5 so that resynthesis may produce significantly better timing at the seed LUT.

As the cone is expanded, a BDD that represents the functionality of the cone is constructed. In addition to the criteria described earlier, cone expansion also stops if the number of BDD nodes exceeds a specified threshold. This is done for three reasons. First, the computational effort of decomposing the cone is kept under control. Second, the circuit changes that result from functional decomposition do not overwhelm the incremental placer which is designed to handle small changes. Third, the effect, on timing, of a change made to a small cone of logic is easier to predict than a change made to a large one.

4.5 Cone Decomposition

We decompose a cone by perfoming functional decomposition on a BDD that represents the cone's functionality. In each step of the decomposition procedure, a single subfunction is extracted and the remaining BDD is reexpressed in terms of the extracted subfunction. The type of LUTs present in the target architecture determine the size of the subfunctions extracted. In Stratix, we extract subfunctions with four or fewer inputs, and in Stratix II, we extract subfunctions with six or fewer inputs. Decomposition continues until the remaining BDD fits into a single LUT in the target architecture.

Every input to the cone and consequently every variable present in the BDD has an associated slack. When a subfunction is extracted from the BDD, a slack is computed for the variable that represents the subfunction in the modified BDD. These slacks will be used to determine an ordering for the variables in the BDD.

Two values extracted from the cone help control the timing driven aspects of the decomposition. The first value, s_{min}, is the minimum slack observed at the seed LUT. It establishes a lower bound on the slack for the decomposition. The decomposition procedure stops and reports failure if it cannot meet the slack requirement at any step. The second value, d_{avg}, is the average delay for a single level of logic (consisting of LUT delay and interconnect delay) in the cone. In Figure 9, if the cone consisted of nodes s, v_4 and v_5 then d_{avg} would be computed as follows

$$d_{avg} = \frac{d(v_4) + d(v_4, s) + d(v_5) + d(v_5, s)}{2}$$

When a subfunction is extracted from the BDD, d_{avg} will be used in computing the slack for the variable representing the subfunction. Specifically, the slack of a new variable v is computed as

$$s'(v) = \min\{s'(u) | u \in \mathit{support}(v)\} - d_{avg} \qquad (2)$$

where $\mathit{support}(v)$ is the set of variables that v depends on.

In addition to the constraint on the lowest allowed slack during decomposition, there is an additional parameter, *area threshold*, which limits the number of extra LUTs that can be created as a result of decomposition. For example, an area threshold of 1.5 indicates that decomposition is allowed to create upto 50% more LUTs than were present in the cone initially. Any decomposition with an area greater than the threshold is rejected.

There are three operations carried out in each decomposition step. First, the variables in the BDD are reordered. Second, a set of bound variables is identified. And finally, the decomposition is performed using the bound variables. We examine the variable reordering and bound set selection operations in detail below.

4.5.1 Variable Reordering

The variable reordering procedure helps move good bound set variables to the top of the BDD. We use the sifting algorithm [18] to reorder the variables. However, unlike traditional sifting whose goal is to optimize the number of nodes in a BDD, our sifting algorithm minimizes a cost function that reflects the suitability of the variable order for decomposition. The cost of a variable order, c_{order}, consists of three components:

1. n_{nodes}: The total number of BDD nodes.
2. $n_{functions}$: The number of distinct functions below every bound set of interest. In Stratix, we are interested in bound sets of size 2–4, and in Stratix II, we are interested in bound sets of size 2–6.
3. s_{total}: The total slack of the first k variables where k is maximum LUT size in the target architecture.

It is defined as follows

$$c_{order} = \frac{n_{nodes} n_{functions}^{\beta}}{s_{total}} \qquad (3)$$

The exponent, β ($\beta = 2$ in the experiments), helps increase the effect that the second component has on overall cost. Each of the components in the cost function has a specific purpose. The first component reduces the complexity of the BDD produced by the variable order. It also ensures that a simple function is produced after a subfunction is extracted from the top of the BDD. The second component reduces the number of new LUTs created as a result of subfunction extraction. And the last component helps move variables with lots of slack towards the top of the BDD where they can be extracted. By extracting the variables with plenty of slack first, we allow the variables with very little slack to go through fewer levels of LUTs.

4.5.2 Bound Set Selection

After the variables in the BDD have been ordered, we select some of the variables at the top of the BDD to be part of a bound set. Both disjoint and non-disjoint decompositions using the bound set are considered. However, the number of new variables that result after subfunction extraction is limited to two and the number of shared variables, in a non-disjoint decomposition, is limited to one. Of the available choices for the bound set, we select the bound set that results in a decomposition with the highest ratio of variables removed to variables added.

4.6 Iteration Parameters

Two iteration parameters, slack ratio threshold and area threshold, control the behavior of seed selection and cone decomposition. Slack ratio threshold begins at a large value (0.12 in the experiments) and is decreased linearly to its final value (0.04 in the experiments). Area threshold begins at a low value (1 in the experiments) and is increased linearly to its final value (2 in the experiments). This allows the early iterations to target a large number of LUTs, but allows very little leeway in the amount of extra area that can be created. And in the later iterations only the most critical LUTs are targeted, but decomposition is allowed to create a lot more area in order to restructure these LUTs.

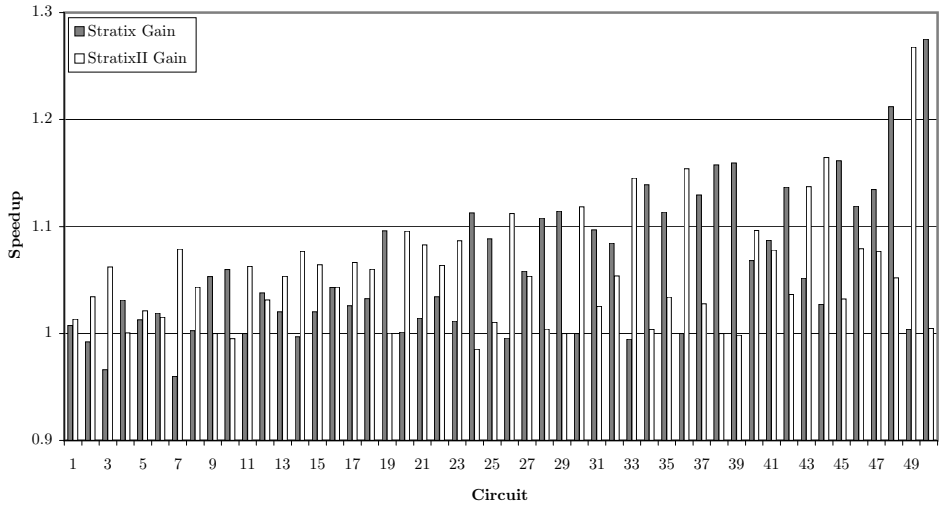

Figure 10: Speedups obtained for 50 industrial circuits.

5. EXPERIMENTAL RESULTS

In our experiments, the first three steps of the FPGA CAD flow, design entry, synthesis and technology mapping, were performed by a leading third-party synthesis tool capable of targeting Altera devices, and the last four steps, including functional decomposition and incremental placement, were performed by a modified version of Quartus II v4.2.

We study the benefits of applying timing driven functional decomposition on 50 industrial circuits. Each circuit was synthesized for both Stratix and Stratix II devices. Circuits synthesized for Stratix devices contained 3400 LEs on average, and circuits synthesized for Stratix II devices contained 1650 ALMs on average. To study the benefits of the decomposition technique, Quartus was run twice, first with the technique turned off and then with the technique turned on. The maximum frequency of operation (FMAX) observed at the end of each run is used to compute the *speedup* observed as a result of applying the decomposition technique. For example, a speedup value of 1.1 indicates that FMAX was 10% higher when the decomposition technique was used. Figure 10 presents the speedups observed for each circuit as a result of applying timing driven functional decomposition. An average speedup of 1.061 was observed on Stratix devices and an average speedup of 1.056 was observed on Stratix II devices. Although the decomposition technique is careful in accepting only those transformations that improve timing, there are a few circuits whose performance is degraded as a result of applying the technique (Stratix circuits 2, 3, 7, 14, 26 & 33, Stratix II circuits 10, 24 & 39). These are a result of the approximation errors in the timing analysis performed by the technique.

The primary goal of the decomposition technique is to improve timing, and there are several instances where a decomposition that is beneficial for timing is selected over a decomposition that is beneficial for area. In obtaining the speedups indicated, the technique increases the number of Stratix LEs by 0.5% and the number of Stratix II ALMs by 2.0%.

The results of Figure 10 were obtained with the maximum number of decomposition iterations set to 4. This value was selected as each decomposition iteration increases compile time and there were very few circuits that benefited from an increased number of iterations. With the number of iterations set to 4, the decomposition technique increases compile time (for the last four steps in the CAD flow) by 81% for Stratix devices and by 102% for Stratix II devices. Although the compile time overhead for Stratix II devices seems larger, the actual time needed for decomposition and incremental placement is similar for both architectures. Due to the larger LABs present in the Stratix II architecture, placement for Stratix II devices is much faster than placement for Stratix devices. Thus, the time taken to perform decomposition and incremental placement appears much larger relative to the time taken by clustering, placement and routing.

6. CONCLUSION

We described a method of performing timing driven functional decomposition on FPGAs. Using the delays obtained from a placed circuit, the method finds alternative decompositions for the critical logic in the circuit. Decompositions were performed using BDDs, and the legalization of any modifications made to the circuit were handled by an incremental placement tool. A set of 50 industrial circuits were used to study the benefits of the method. An average speedup of 6.1% and 5.6% was observed on Stratix and Stratix II devices as a result of using the method.

7. REFERENCES

[1] M. Pedram and N. Bhat. Layout Driven Logic Restructuring/Decomposition. In *Proceedings of the Int. Conf. on Computer-Aided Design*, San Jose, CA, Nov. 1991, pp. 134–137.

[2] J. Y. Lin, A. Jagannathan and J. Cong. Placement-Driven Technology Mapping for LUT-Based FPGAs. In *Proceedings of the ACM Int. Syposium on FPGAs*, Monterey, CA, Feb. 2003, pp. 121–126.

[3] Y. Jiang, A. Krstic, K. Cheng and M. Marek-Sadowska. Post-Layout Logic Restructuring for Performance Optimization. In *Proceedings of the Design Automation Conference*, Anaheim, CA, June, 1997, pp. 662–665.

[4] Y. Lian and Y. Lin. Layout-based Logic Decomposition for Timing Optimization. In *Proceedings of the Asia Pacific Design Automation Conference*, Hong Kong, Hong Kong, Jan. 1999.

[5] G. Stenz, B. Riess, B. Rohfleisch and F. Johannes. Timing Driven Placement in Interaction with Netlist Transformations. In *International Symposium on Physical Design*, Napa Valley, CA, 1997, pp. 36–41.

[6] T. Tien, H. Su and Y. Tsay. Integrating Logic Retiming and Register Placement. In *Proceedings of the Int. Conf. on Computer-Aided Design*, San Jose, CA, 1998, pp. 136–139.

[7] L. Kannan, P. Suaris and H. Fang. A Methodology and Algorithms for Post-Placement Delay Optimization. In *Proceedings of the Design Automation Conference*, San Diego, CA, June 1994, pp. 327–332.

[8] D. Singh and S. Brown. Integrated Retiming and Placement for Field Programmable Gate Arrays. In *Proceedings of the ACM Int. Syposium on FPGAs*, Monterey, CA, Feb. 2002, pp. 67–76.

[9] K. Schabas and S. D. Brown. Using Logic Duplication to Improve Performance in FPGAs. In *Proceedings of the ACM Int. Syposium on FPGAs*, Monterey, CA, Feb. 2003, pp. 136–142.

[10] M. Sheng and J. Rose. Mixing Buffers and Pass Transistors in FPGA Routing Architectures. In *Proceedings of the ACM Int. Symposium on FPGAs*, Monterey, CA, Feb. 2001, pp. 75–84.

[11] Altera. *Stratix Device Handbook (Complete Two-Volume Set)*. v3.1, Sept. 2004.

[12] Altera. *Stratix II Device Handbook (Complete Two-Volume Set)*. v1.2, Oct. 2004.

[13] V. Manohararajah, D. P. Singh, S. D. Brown and Z. G. Vranesic. Post-Placement Functional Decomposition for FPGAs. In *Proceedings of the International Workshop on Logic Synthesis*, Temecula, CA, June 2004, pp. 114–118.

[14] D. P. Singh and S. D. Brown. Incremental Placement for Layout-Driven Optimizations on FPGAs. In *Proceedings of the Int. Conf. on Computer-Aided Design*, San Jose, CA, 2002, pp. 752–759.

[15] R. Hitchcock, G. Smith and D. Cheng. Timing Analysis of Computer-Hardware. *IBM Journal of Research and Development*, Jan. 1983, pp. 100–105.

[16] R. Ashenhurst. The Decomposition of Switching Functions. In *Int. Symposium on Theory of Switching Functions*, 1959, pp. 74–116.

[17] H. Curtis. A Generalized Tree Circuit. *Journal of the ACM*, 1961, 8:484–496.

[18] R. Rudell. Dynamic Variable Ordering for Ordered Binary Decision Diagrams. In *Proceedings of the Int. Conf. on Computer Aided Design*, Santa Clara, CA, 1993, pp. 42–47.

[19] Y-T. Lai, K-R. Pan and M. Pedram. OBDD-Based Functional Decomposition: Algorithms and Implementation. *IEEE Trans. On Computer Aided Design*, Vol. 15, No. 8, 1996, pp. 977–990.

[20] C. Yang and M. Ciesielski. BDS: A BDD-Based Logic Optimization System. *IEEE Trans. On Computer Aided Design*, Vol. 21, No. 7, July 2002, pp. 866–876.

[21] C. Legl, B. Wurth and K. Eckl. A Boolean Approach to Performance-Directed Technology Mapping for LUT-Based FPGA Designs. In *Proceedings of the 33rd Design Automation Conference*, Las Vegas, Nevada, 1996, pp.730–733.

[22] H. Sawada, T. Suyama and A. Nagoya. Logic Synthesis for Look-Up Table Based FPGAs Using Functional Decomposition and Boolean Resubstitution. *IEICE Trans. Inf. & Syst.*, Vol. E80-D, No. 10, October 1997, pp. 1017–1023.

[23] N. Vemuri, P. Kalla and R. Tessier. BDD-Based Logic Synthesis for LUT-Based FPGAs. *ACM Transactions on Design Automation of Elec. Systems*, Vol. 7, No. 4, October 2002, pp. 501–525.