A Synthesis Oriented Omniscient Manual Editor

Tomasz S. Czajkowski and Jonathan Rose
Edward S. Rogers Sr. Department of Electrical and Computer Engineering
University of Toronto, Toronto, Ontario, M5S 3G4, Canada
{czajkow|jayar}@eecg.toronto.edu

ABSTRACT

The cost functions used to evaluate logic synthesis transformations for FPGAs are far removed from the final speed and routability determined after placement, routing and timing analysis. This distance has given rise to the field of physical synthesis, which attempts to improve logic synthesis by employing cost functions that contain placement, routing and/or timing analysis information.

In this work we take this notion to an extreme that we call *omniscience*, in which post-routing timing analysis is provided in the context of a manual editor in which the user selects logical and physical transformations. After each incremental circuit modification, the user is informed of the circuit performance after routing and timing analysis. Since the computations involved in providing this level of information are large, we restrict the application to relatively small circuits, no larger than 1000 logic elements.

Using this approach on a commercial FPGA, we propose a set of logic transformations specific to the logic and routing architecture of the Xilinx Virtex-E device. On a set of 10 circuits we have achieved an average performance improvement of 10% when both logical and physical changes are used. Another value of the editor is that it reveals new types of automatable physical-synthesis transformations and optimization strategies that arise from architectural properties of the target device.

Categories and Subject Descriptors

B.6.3 [**Logic Design**]: Design Aids – Optimization

General Terms

Algorithms

Keywords

Synthesis, Manual, Virtex-E

1. INTRODUCTION

Typical CAD flows for FPGAs perform independent and sequential optimizations as illustrated in Figure 1(a). The early stages make decisions with poor visibility into the final result, after placement and routing. For example, lookup-table based logic synthesis algorithms [4][11] use LUT depth to measure the delay effect of a logic transformation, which doesn't account for the delay effect of physical distance after placement.

FPGA'04, February 22–24, 2004, Monterey, California, USA

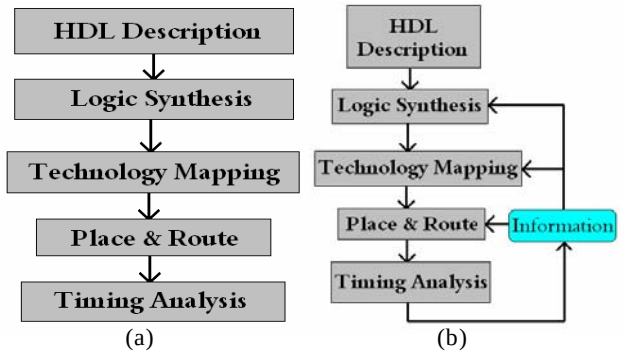


Figure 1: Typical FPGA CAD Flow

(a) single pass, (b) physical synthesis with iteration

Various physical synthesis techniques have been proposed that provide more knowledge from the physical design steps back into the logic synthesis stages as illustrated in Figure 1(b). The basic approach is to iterate between synthesis and placement (including a post-placement timing analysis). The two keys to these efforts are the amount and type of information passed back to synthesis, and the constraints passed forward from synthesis into placement. In [2][3][5] and [10] the post-placement position of logic is used to form more accurate models of net delays that are then used to evaluate logic transformations. The output of synthesis also indicates where newly created logic should be placed. While [5] and [10] execute logic synthesis and placement as discrete "batch mode" steps, [2] constantly moves back and forth between the two, maintaining the most up-to-date view of the placement.

The work in [1] performs physical synthesis in a different way: the logic synthesis stage provides several mapping solutions for a group logic from which the placement stage selects the final choice.

In most cases, these approaches to physical synthesis group many logic transformations together and pass them on to a placement phase for legalization and recalculation of delay effects. Even in the approaches that iterate more frequently between synthesis and placement, the true effect of a single change is not known because it only becomes apparent after routing and timing analysis.

In this work we propose a far more precise decision making process: each *single* transformation will be evaluated using the *post-routed* timing analysis. This approach captures the exact effect of an incremental circuit transformation on the post-routing circuit speed. We call this kind of cost function *omniscient*, because it knows everything that is possible to be known about the result of a transformation. This work is done in

the context of a manual editor, because the computation required to provide omniscience is large, but can be tolerated in the human-interaction scale of time. To reduce the time spent on computation we limit the size of the circuit to less than 1000 logic elements.

The work in [6] applied an omniscient approach to manual clustering and placement and achieved significant speed improvements. We extend their work to the synthesis domain, allowing changes that simultaneously encompass the logic functionality, the physical placement, and the routing. An interesting outcome of this new work is the invention of transformations that target a mixture of architectural features across the logic, placement and routing domains. For example, we show how adjustments in the choice of the carry structure give benefit once the placement and routing flexibility is taken into account.

Our manual editor, Augur, targets a real FPGA architecture, the Xilinx Virtex-E family [7]. The exact delay modeling provided by Xilinx FPGA Editor enables the manual editor to perform accurate timing analysis. The performance improvement obtained with this approach is real.

Augur gives the user the ability to perform logic transformations on a fragment of the circuit. The user selects the circuit fragment and specifies the transformation to be performed on it. Augur applies the transformation and requires the user to provide the placement for the newly created logic elements. The editor then re-routes and timing analyzes the circuit and provides the user with the resulting circuit speed.

The remainder of this paper is organized as follows: Section 2 describes the relevant architectural features of the Virtex-E FPGA and the work in [6] in more detail. Section 3 describes how the user employs the editor, which we call Augur. Section 4 details the logic transformations available in the Augur, and Section 5 provides benchmark results. In Section 6 we look at the optimization strategies used in the manual operation of the editor and propose automatable procedures that could be implemented as scripts in our tool or employed as physical synthesis algorithms.

2. BACKROUND

Some of the logic transformations available in Augur hinge on structures specific to the Xilinx Virtex-E architecture. In this section we review that device architecture and provide a short overview of the prior manual editor [6] our work is based on.

2.1 The Xilinx Virtex-E Architecture

The FPGA devices in the Xilinx Virtex-E family are island-style FPGAs [7]. Each Configurable Logic Block (CLB) is divided into two *slices*. Each slice contains two flip-flops, two 4-input LUTs, carry logic, some extra multiplexors, and flip-flop control logic, as shown in Figure 2. The flip-flops in each slice must have the following common control signals: Clock, Set, Clear and Enable. In Section 4 we describe how these signals are manipulated to allow tighter packing of flip-flops.

The Virtex-E Routing Architecture

The routing architecture consists of four types of wires: single-length wires between neighboring CLBs, the Nearest-Neighbour interconnect [8] length six wires, connecting CLBs that are six rows/columns apart, and significantly longer wires.

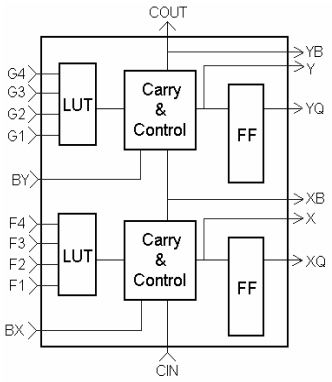


Figure 2: A Virtex-E Architecture "Slice"

The key type of wire that is used by our optimization strategies is the Nearest Neighbour (NN) Interconnect [8]. It is a dedicated set of wires that connects horizontally adjacent CLBs. This interconnect has very low delay and can significantly improve the performance of the circuit when used properly. There are two pairs of unidirectional wires between each pair of CLBs.

2.2 Omniscient Manual Placer & Packer

Our effort is based on the work in [6], which introduced an omniscient manual editor that also targeted the Virtex-E architecture. It enabled the user to modify the placement and packing of circuits (that are first created in the usual automated way) and afterwards conducted the timing analysis to inform the user about the new circuit performance, thereby providing omniscience to the user. The precise delays used in the timing analysis were obtained by querying the Xilinx FPGA Editor software. All changes to the circuit were made both within the editor itself and the Xilinx FPGA Editor back end, so that a full functioning circuit was always available. The timing analysis, which was restricted to single-clock circuits, was done in a somewhat incremental fashion to decrease the response time seen by the user. On a set of eight benchmark circuits, [6] achieved an average increase in operating frequency of 12.7% compared to the best of 10 runs of the standard synthesis, placement and routing flow.

3. USER EXPERIENCE

We first describe how the user interacts with Augur, and then discuss the details of the specific logic synthesis transformations it provides.

To begin, the user performs a fully-automated synthesis, placement and routing flow on the circuit, as illustrated in Figure 1(a). We use the Synplicity Synplify Pro version 7.1 tool and Xilinx ISE version 5.1. The results are read into Augur, which creates a display such as the one illustrated in Figure 3. The LUTs, flip-flops and carry logic are separately displayed. The user can set a desired operating frequency, or the critical path delay, and cause Augur to display components and nets that are critical or close to critical in red, as illustrated in Figure 3.

3.1 Placement and Packing Modifications

To modify the packing or the placement, the user selects a set of components and moves them into a new location. Changes

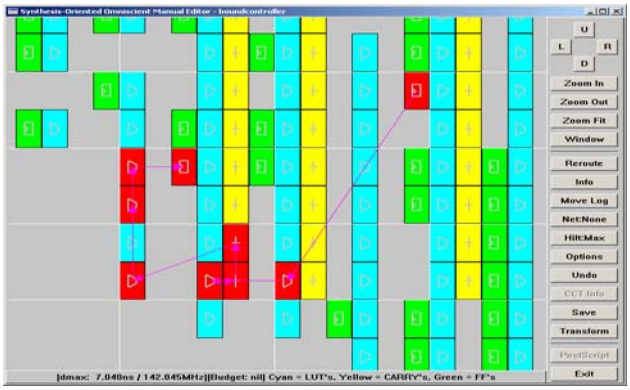


Figure 3: Editor screenshot: LUTs are cyan muxes, flip-flops are green rectangles, carry logic is yellow +; The critical path is highlighted in red.

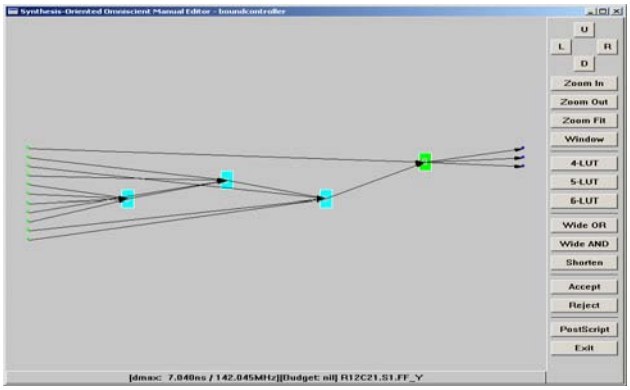


Figure 4: Schematic View with Transformation List

to the placement and packing are made within Augur's data structures and are transmitted through named pipes to a live version of the Xilinx FPGA Editor, which returns the delays on any modified connections. Augur performs the timing analysis and presents the updated circuit speed to the user.

3.2 Logic Synthesis Changes

Upon inspection of the view presented in Figure 3, the user can select a set of logic components for synthesis transformations. The user then selects the "Transform" button to change the view to a schematic form as illustrated in Figure 4.

This view presents the selected sub-circuit with the inputs on the left and outputs on the right side of the screen. The user can select the components in this view and perform logic synthesis transformations on them. The user must determine the new placement for the new circuit components produced in the transformation. It is possible that there are more, or fewer, components than before the circuit was transformed, which require positions. To choose those positions, the user switches back to the grid view, as illustrated in Figure 5. This latter view is important to ensure that routing features, such as the Nearest-Neighbour interconnect, are profitably leveraged.

4. LOGIC SYNTHESIS TRANSFORMS

Now that we have provided the context for Augur, we describe the logic synthesis transformations provided to the user. The transformations available to the user are: remapping,

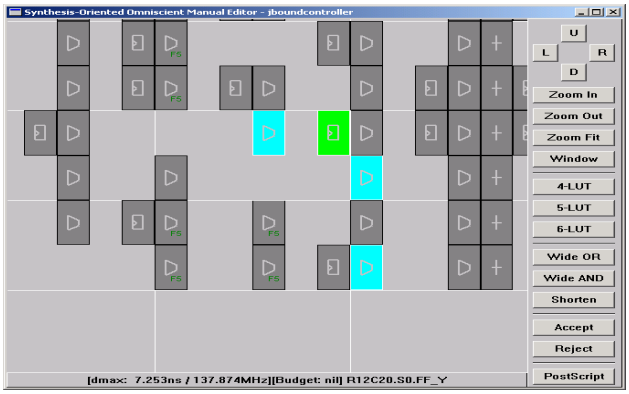


Figure 5: Placement of Newly Synthesized Logic

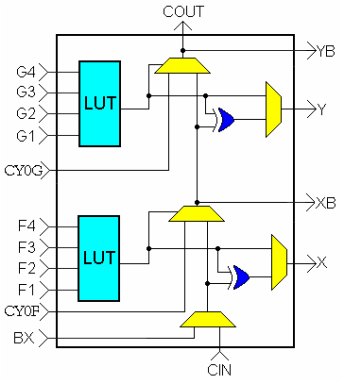


Figure 6: Slice with a carry chain

dupl

ication, merging, carry chain shortening and control signal extraction.

The input to every transformation is a connected graph of logic components selected by the user. When the user selects a specific transformation for that logic, the tool determines if the selected logic can be successfully transformed. If it can, then the application of a transformation results in a new set of logic components that the user can place. In the following we describe the set of logic synthesis transformations available in Augur.

4.1 Remapping

The remapping operation transforms a set of logic components into a functionally equivalent set that fits into a single slice or a CLB. The following sub-sections describe two mapping algorithms: carry chain mapping and multiplexer mapping.

4.1.1 Mapping into Carry Chain

In this subsection we focus on how the Virtex-E carry chain logic can be utilized to implement an AND or an OR gate, and the algorithm that makes this mapping possible in Augur.

The carry chain logic in the Virtex-E slice connects the two LUTs together, as shown in Figure 6. This structure can be manipulated to implement an AND or an OR by assigning constant values to inputs CY0F, CY0G and CIN. To convert the carry chain into an AND/OR gate we set CY0F=CY0G=0/1 and CIN=1/0.

The advantage of using the carry chain structure is that a pair of serially connected LUTs can sometimes be converted into a parallel pair of LUTs connected through this fast AND or

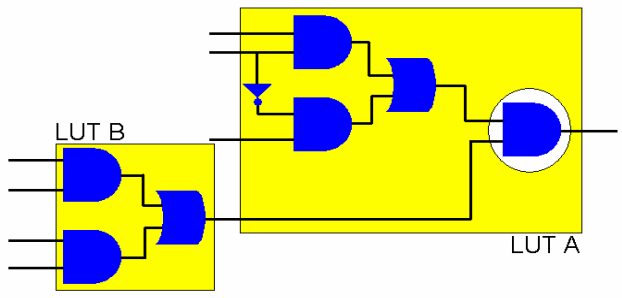


Figure 7: Example Circuit for AND Gate Mapping into Carry Chain

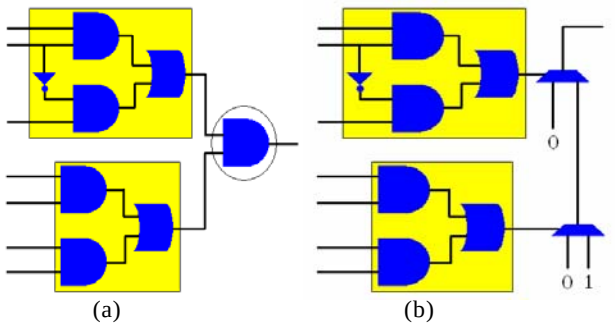


Figure 8: Remapping: **(**a) AND gate extracted from LUT A, (b) Implementation of AND in Carry

OR gate. Since the carry multiplexor is very fast, the transformation can lead to an overall reduction in delay if the original pair of LUTs is on the critical path.

Consider the example pair of LUTs (A and B) illustrated in Figure 7, which shows the logic function of each LUT as a schematic inside each LUT box. The highlighted AND gate at the right of LUT A can be implemented in the Carry Chain of Figure 6 because one of its inputs comes directly from LUT B. Figure 8(a) illustrates the isolation of this AND gate and Figure 8(b) shows the final implementation.

To determine if a pair of serially connected LUTs can be mapped into a slice, we need to find an AND gate (or OR gate) as shown in Figure 8(a). To do this we perform a Shannon expansion of LUT A's function with respect to LUT B. Let A be the function of LUT A, and B be the signal generated by LUT B. The inputs to A are x1, x2, x3 and B. From Shannon's theorem [9] we have:

A = A(x1,x2,x3,0) B' + A(x1,x2,x3,1) B

If the function A(x1,x2,x3,1) evaluates to a constant 0, then the function A reduces to:

A = A(x1,x2,x3,0) AND B',

which gives us the AND gate we were looking for to map into the carry chain of the slice. An OR gate can be detected by following a similar procedure.

The algorithm in Figure 9 determines if a pair of serially connected LUTs can be transformed in this manner. Its input is a user-selected pair of serially connected LUTs, and the output is either the mapping of those LUTs into a slice, or a declaration that the mapping is not possible.

In addition, it is possible to map a pair of serially connected LUTs in which the LUT B has fanout greater than 1. As shown in Figure 6, the carry chain structure allows the bottom LUT to produce an output signal through pin X. To properly generate

Input: A pair of series-connected LUTs A and B, with LUT B driving LUT A.
Output: On success, a mapped Slice

Let A be the function of LUT A, and B be the function of LUT B.
f_0 = Shannon expansion of A with respect to B=0
f_1 = Shannon expansion of A with respect to B=1

if either f_0 or f_1 is a constant (i.e. is 0 or 1)
    then it is possible to map A and B into a slice:
    let h be the co-factor (f_0 or f_1) that is not a constant
    implement h in the top LUT of the slice, inverting if necessary as per Section 2.1
    implement B in the bottom LUT
        create a LUT that inverts the output of LUT B if B uses output pin X
    return function implemented in a Slice
else
    return fail.

Figure 9: Algorithm Mapping AND & OR into the Carry Chain

this secondary output it may be necessary to add a single LUT that inverts the output of pin X, since the function of LUT B may need to be inverted to implement an AND gate (or an OR gate) in the carry chain.

4.1.2 Mapping into Joint-LUTs and Joint-Slices

The Virtex-E slice contains fast multiplexers that combine two or more LUTs to create higher fanin functions. The inputs to the multiplexer are the outputs of both LUTs in the slice, as shown in Figure 10. This structure, which we will call the *Joint-LUT* structure, allows some 9-input functions to be implemented in a single slice.

To implement a 9-input logic function in the Joint-LUT structure we have to decompose the function such that one of its inputs will drive the selector input of the multiplexer, while the remaining inputs will be assigned to the LUTs in the slice. Shannon's decomposition theorem [9] allows us to do this. The basic idea of the procedure is to perform Shannon's expansion with respect to every variable of the function [15]. A valid mapping of the 9-input function into the Joint-LUT structure is found when both cofactors of the function in the Shannon's expansion are functions with at most 4 inputs.

The advantage of using the Joint-LUT structure is that it exploits parallelism, which can reduce the delay of signals passing through it. For example, when a pair of serially connected LUTs is mapped into this structure, the function of both LUTs in the slice is evaluated in parallel. Thus, the delay through the Joint-LUT structure is the delay of one LUT and that of the dedicated multiplexer.

It is possible to implement even more complex functions by merging two Joint-LUT structures with a multiplexer available in the Virtex-E CLB, as illustrated in Figure 11. We call this the *Joint-Slice* structure, and it can have up to 19 total inputs. The mapping algorithm is very similar to the Joint-LUT mapping algorithm above, except that the algorithm looks for a successful mapping of each cofactor into a Joint-LUT structure instead of a LUT.

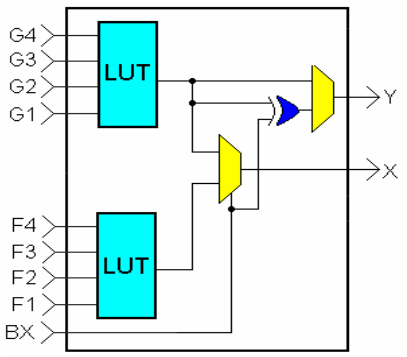


Figure 10: Joint-LUT structure

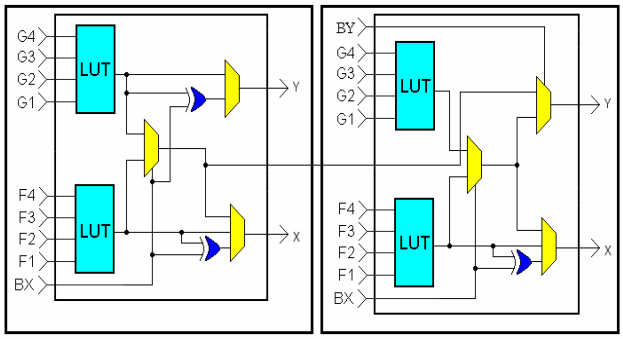


Figure 11: CLB in Joint-Slice configuration

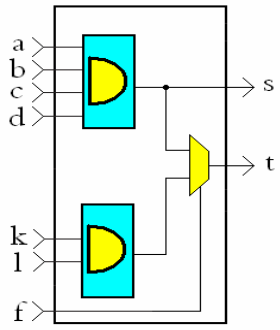


Figure 12: Mapping two functions into a Joint-LUT

This approach is similar to the one proposed in [15]. We have enhanced the algorithm to take advantage of extra outputs available in the Joint-LUT and the Joint-Slice, as illustrated in Figure 12. Observe that pin X of the slice, as shown in Figure 10, generates the primary output of the slice and that pin Y can be set to generate the function of the upper LUT.

Consider mapping a 9-input/2-output function into the Joint-LUT structure. The primary output function can be easily determined, as before. To implement a second function, it has to be the primary output function for the top LUT, which is the substructure of the Joint-LUT. Similarly, mapping the same function into the Joint-Slice means that the second function has to be either the primary output function for a Joint-LUT or a LUT.

We exhaustively search for a matching between the outputs of the multi-output function and the outputs of the target structure. The algorithm to map a set of functions into a Joint-LUT structure is presented in Figure 13. The algorithm for Joint-

```
Function map_into_Joint_LUT(F)
Input:   F - a set of at most 2 functions
Output:  TRUE/FALSE

Let t be the primary output function
Let s be the other function
For all variables x in the support of t
    Perform Shannon's Expansion of t with respect
        to x to obtain cofactors t_x̄ and t_x
    if each cofactor fits in a single LUT then
        if   (|F| = 2 and t_x̄ = s or t_x = s ) or
             (|F| = 1) then
                return TRUE;
end for
return FALSE;
```

Figure 13: Algorithm for mapping into Joint-LUT

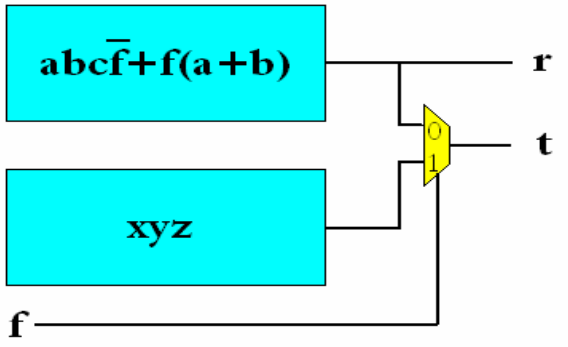


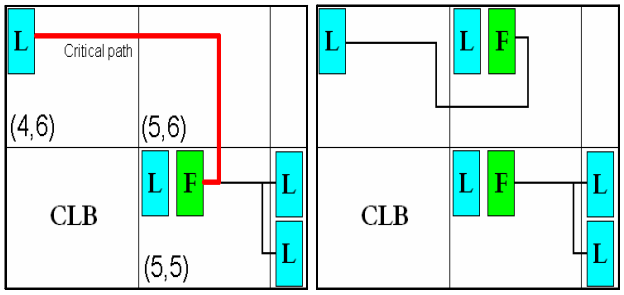
Figure 14: Mapping not found by our algorithm

Slice mapping is similar. Instead of checking for mapping of the cofactors into a single LUT, we check for mapping of cofactors into the Joint-LUT.

Note that this algorithm will be unable to map structures like those illustrated in Figure 14, even though this is a correct mapping. This is because in Shannon's expansion one variable is removed from the equation of each cofactor and it is assigned to drive the multiplexer selector input. However, the function *r* in Figure 14 depends on the variable *f*.

4.2 Duplication

The second transformation available in Augur is duplication, which creates a copy of a selected component. By duplicating a component on the critical path, one can increase the freedom to position and route critical connections [13]. We have found that duplication is particularly useful feature that enables the use of fast Nearest-Neighbour (NN) interconnect for critical connections. For example, consider the circuit in Figure 15(a). The critical path starts at a flip-flop at location (5,5) and goes through the LUT at location (4,6). We generally try to put critical connects on NN interconnect to speed them up, but in the current placement this strategy cannot be used, because the first LUT on the critical path is not in the adjacent CLB. Moving the flip-flop from (5,5) to (5,6) permits this connection to use the NN interconnects, but removes the NN connections from the LUTs at (6,5). By duplicating the flip-flop and placing the duplicate in the CLB at location (5,6), as shown in Figure 15(b), NN connections can be used for all outputs of the flip-flop.



(a) (b)

Figure 15: Duplication example: a) before transformation, b) after duplicating the LUT and FF and placing them in the CLB above

4.3 Merging

It is sometimes beneficial to reverse duplication that has occurred in previous synthesis, which we allow in a transformation called *merging*. For example, after the placement and routing it becomes clear that the distribution of connections between two duplicated components is causing the performance to suffer.

This transformation allows us to explore other mapping solutions or to redistribute connections between duplicate components. To redistribute connections we first merge two identical components and then perform duplication again, but with different connection distribution.

4.4 Carry Chain Shortening

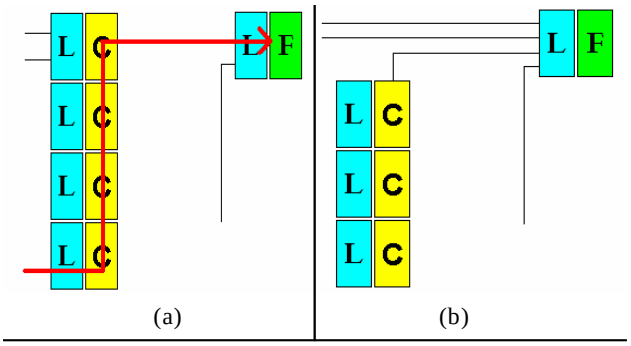
Carry chains have long been part of FPGA architectures [12][14] because they provide a high-speed path for long bit addition and arithmetic operations. Their use often reduces the time along the critical path, or removes the arithmetic operation from the critical path entirely. They come with some drawbacks, however, that reduce their positive impact: most importantly, carry structures force the logic blocks that use them to be a fixed vertical or horizontal structure. This lack of flexibility is the flip side of the greater speed of connectivity.

In addition, the blind use of a carry chain may prevent other beneficial optimizations. For example, consider the circuit in Figure 16(a), which illustrates a 4-bit carry chain that feeds one more 2-input LUT and then a flip-flop. Assume that the most significant bit of the carry chain, including the LUT, implements a 3-input function. The most significant bit calculation, including carry, and the final 2-input LUT function can all be implemented in a single 4-input LUT as illustrated in Figure 16(b).

We call this operation *carry-chain shortening*, as it removes the carry primitive from the top of the chain. Most synthesis tools are not permitted to optimize carry primitives away, and so this opportunity is typically unexplored. The procedure to test if this optimization is possible is quite straightforward.

4.5 Register Control Signal Extraction

Recent versions of synthesis tools have used flip-flop control signals to implement greater logic functionality in a single slice. For example, attaching a logic signal to a flip-flop's synchronous clear input has the effect of ANDing that signal with the flip-flop's D input. While this kind of optimization can



(a) (b)

Figure 16: Carry chain shortening example: (a) critical path travels through the carry chain, (b) the path is shortened by one carry component

Table 1 – Benchmark Statistics and Baseline Maximum Operating Frequency

Benchmark Name	Size		Maximum Operating Frequency (MHz)
	# LUTs & Carry Cells	# Flip-Flops	
Batcher	253	436	298.6
Miim	162	119	155.0
Vision	310	243	197.4
Banyan	177	335	359.3
Trap	186	486	381.0
Boundcontroller	472	466	131.5
Linearmap	460	72	108.0
Vidout	447	220	134.4
Raygencont	211	118	162.1
Mult	29	21	122.2

be beneficial with respect to logic depth, it can also have a negative side-effect: a flip-flop synthesized this way cannot be packed into a slice with another flip-flop that does not use exactly the same clear signal (most FPGA logic blocks impose this kind of restriction).

This poses a restriction on the packer and may degrade the final performance. The alternative, which we implement as a logic synthesis transformation, is to implement the synchronous clear function in a separate LUT. This certainly costs extra logic, but it can potentially improve speed because it increases the packing flexibility and therefore local connectivity and access to NN interconnects.

5. EXPERIMENTAL RESULTS

In this section we give the results of using Augur on ten benchmark circuits. We begin by describing the set of circuits and how we used commercial tools to create an extremely high quality (and therefore fair) baseline synthesis, placement and routing for comparison. Since the method for producing the baseline is different from that of [6], we first compare our packing and placement only modifications to that of [6]. We then show the further gains that are possible with our new synthesis transformations.

5.1 Benchmark Circuits

Table 1 provides a summary of the size of each of the ten benchmark circuits. These circuits come from designs made at the University of Toronto and IP cores available through the

Table 2 – Results using packing and placement only

Benchmark Name	Baseline Frequency (MHz)	Freq. after Packing + Placement	% Imp
Batcher	298.6	314.0	5.1
Miim	155.0	155.2	0.1
Vision	197.4	197.8	0.2
Banyan	359.2	367.8	2.4
Trap	381.0	398.6	4.6
Boundcontrol	131.5	137.9	4.8
Linearmap	108.0	109.3	1.3
Vidout	134.4	140.0	4.1
Raygencont	162.1	173.2	6.8
Mult	122.2	122.3	0.1
Average			3.0

Table 3: Results with New Synthesis Transformations

Benchmark Name	Frequency After Pack/Place & Synthesis (MHz)	% Improved	% From Synth Only
Batcher	374.8	25.5	19.4
Miim	155.6	0.4	0.2
Vision	210.2	6.5	6.3
Banyan	367.8	2.4	0.0
Trap	418.4	9.8	5.0
Boundcontroller	149.5	13.7	8.5
Linearmap	125.7	16.4	14.9
Vidout	155.6	15.8	11.2
Raygencont	173.2	6.8	0.0
Mult	124.3	1.7	1.6
Average		9.9	6.7

internet. A more detailed description of each circuit can be obtained from [16].

5.2 Baseline Circuit Generation

To achieve the best possible baseline circuits we created a very rigorous procedure using the best-in-class tools. Figure 17 summarizes the procedure, the key of which is that each circuit is placed and routed using 100 different seeds each for at least five different target frequency settings. Although this is not practical for large circuits, the limit we imposed on the circuit size enabled us to do this in reasonable amount of time. As a result our baseline circuits have baseline performance that on average is 2.4% better than using the method in [6].

5.3 Placement and Packing Only Changes

In order to separate out the additional advantage of the synthesis transformations described in Section 4, we first improve the baseline circuits using only packing and placement modifications, in the manner of [6]. Table 2 provides a summary of these results. We obtained an average of 3.0% performance improvement across the 10 circuits. This is significantly less than the 12.7% achieved in [6], which we believe is due to the following reasons:

1. We are using newer placement and routing tools (Xilinx ISE 5.1 service pack 3 vs. 3.3 SP7).
2. We are using newer, better Synthesis tools (Synplicity Synplify 7.1 Pro vs. Synplify 6.2 Pro).

1. Set target frequency to initial setting F_{init} and synthesize
2. Synthesize using Synplify 7.1 Pro
3. Place and route using Place and Route tool (par) provided with Xilinx ISE 5.1 Service Pack 3 tools. Set effort to maximum and number of P&R attempts (with different seeds) to 100.
4. Record best result
5. Repeat 2-4 for target frequency -10%, -5% +5% and +10% with respect to F_{init}.
6. If a better solution was obtained for frequency other than F_{init} then repeat 2-5 using that frequency as F_{init}.

Figure 17: Procedure used to generate baseline circuits results and circuits

3. Our new method of generating 100 seeds and choosing the best, obtains better baselines.
4. We have 10 circuits in the suite versus 8 in [6], where only 4 are common to both suites.
5. The results in both cases are obtained with humans in the loop, and human-based operations are not very reproducible.

We note that the Synplify 7.1 Pro version utilized the multiplexers in slices more often than version 6.2, essentially making use of the Joint-LUT structure. While this can reduce delay it also places some restrictions on placement perhaps contributing to diminished improvements using just placement and packing.

5.4 Results with Synthesis Transforms

We now provide results for the use of Augur employing manual packing, placement and synthesis transformations. Table 3 summarizes the results, giving the new maximum operating frequency and then separating out the additional gains from synthesis only. The latter was determined by calculating the percent difference between the frequency in column 3, Table 2, and column 2, Table 3. Below we summarize the key steps involved in improving each circuit:

1. **Batcher** - the majority of the flip-flops in this design were synthesized to use distinct control signals. These control signals were used to minimize the combinational logic part of the circuit. However, the resulting packing allowed only one register to occupy a slice, which spread the circuit over a much larger area than necessary. The application of Flip-Flop control signal extraction allowed previously incompatible flip-flops to share a slice, resulting in an overall 25.5% improvement over the baseline logic circuit speed.
2. **Miim** - duplication was used to improve performance of some of the paths in the circuit. However, the complexity of the design, as well as the congestion in the critical region, allowed for only minor (0.6%) improvements.
3. **Vision** - this logic circuit suffered from improper synthesis of logic that controlled Flip-Flop enable signals. Analysis of these signals showed that each of the enable signals was functionally identical, while the logic function for these signals was a 7-input AND gate. The logic synthesis tools implemented this logic function as a pair of serially connected LUTs and duplicated the forward LUT to reduce fanout. Logic merging was first used to create a single logic function to implement the enable signal. Then both LUTs were duplicated to reduce the fanout and distribute the connection properly. Further improvement was obtained by

implementing these two LUTs in a carry chain, using the Carry Chain Remapping transformation.

4. **Banyan** - each path in this circuit contains at most one LUT. This was made possible through the use of flip-flop control signals to reduce the logic depth of the logic circuit. We were only able to improve the performance of the circuit by modifying the placement and packing.
5. **Trap** - in this circuit a few registers employed flip-flop control signals to decrease logic depth. However, it was critical to circuit performance that these flip-flops had the freedom to share a slice with other flip-flops. We used control signal extraction to achieve placement flexibility for these flip-flops. Once these flip-flops had the flexibility to share a slice with other flip-flops, we were able to modify the placement and packing of the circuit effectively. In combination with logic duplication the logic circuit speed was increased by 9.8%.
6. **Boundcontroller** - this design contained a number of Joint-LUT structures. A closer examination revealed that remapping certain pairs of them into Joint-Slice structures with multiple outputs was possible. After these pairs of Joint-LUTs were remapped into Joint-Slice structures the placement of the logic components was rearranged to promote usage of NN interconnect.
7. **Linearmap** - the design contains mostly carry chain logic. The problem was that a few registers were driving multiple carry chains and could not use NN interconnect for all connections. Duplicating some of those registers and re-synthesizing non-carry chain logic into Joint-LUT and Joint-Slice structures improved the logic circuit speed by 16.4%.
8. **Vidout** - this circuit contained a carry chain that was unnecessarily long. The output of the top carry cell was not driving the local register, which made it a candidate for the carry chain shortening transformation - the functionality implemented by the top segment of the carry chain and the LUT driven by the carry chain could be implemented in a single LUT. This modification allowed for further logic optimization resulting in the improvement in the logic circuit speed by 15.8% compared to the baseline logic circuit speed.
9. **Raygencont** - the critical path of this logic circuit traverses LUTs that could be re-synthesized into wide AND gate carry chain structures. However, that causes the near critical paths to become critical with longer delay. Without logic synthesis transformations we were able to modify the placement and packing of the circuit to improve the speed by 6.8%.
10. **Mult** - the original placement and synthesis was good, however performing logic duplication and remapping improved the logic circuit speed by 1.7%.

6. OPTIMIZATION STRATEGIES

Even though Augur is a *manual* editor, one of the long-term goals of this research is to discover new optimization strategies that could be automated. In this section we propose several such strategies, which could form the basis of future algorithms.

6.1 Promoting NN Interconnect

The first strategy focuses on the Nearest-Neighbour routing (NN) architecture of the Virtex-E. The best optimization opportunities were those that enabled many critical connections to use NN interconnect: we moved logic elements so that they could take advantage of available direct connections between logic blocks, as well as created available direct connections by liberating those occupied by non-critical logic. The latter can be done by logic transformations such as remapping and merging.

Remapping can be used to liberate an NN interconnect link by transforming the implementation of serially connected pair of LUTs to a Joint-LUT or Joint-SLICE structure. This replaces the NN connection between LUTs by an internal-to-the slice connection, freeing the NN for use by others.

Here is an outline of an algorithm that could be used as part of a larger automated optimization strategy:

For each pair of LUTs on a critical path that could employ an NN interconnect, but is prevented from using it by the presence of another connection:

- Select the logic that is using the desired NN interconnect.
- If that logic can be moved to another location without hurting performance then do so. Otherwise, apply remapping or merging to liberate the NN interconnect, while maintaining the performance of the non-critical path.
- If successful, this should allow the critical logic to acquire the liberated NN connection.

6.2 Liberating Space for Critical Logic

The focus of the second strategy is to free the logic components in a slice or CLB that can be utilized by critical logic. For this strategy we look at logic components on the critical path and search for a suitable placement for them. The desired placement may conflict with other, non-critical, logic. Thus, we focus our attention on the non-critical logic in an effort to move it out of its current location, while preserving its performance.

The logic transformations liberate space occupied by non-critical logic by:

- Speeding up non-critical logic, allowing it to move from its current location, without a performance penalty
- Duplicating components with high fanout, allowing them to be placed in different locations of the device, while maintaining the circuit performance

Here is an algorithm that employs the space-liberation approach to improving performance:

For each area containing a critical, or close-to-critical, path:

- Locate the non-critical logic
- Move the non-critical logic away only if it does not decrease the circuit performance
- For logic that will become critical if moved, apply remapping and duplication to speed it up, thus making it possible for it to move away
- Move the critical logic into the liberated space

6.3 Increasing Packing flexibility of FFs

We noticed that modern logic synthesis tools often use flip-flop control signals to reduce the amount of logic. This can adversely affect the flip-flop's placement flexibility, as

```
Bin 10: 1.715ns-3.314ns, count = 16
Bin 9: 3.314ns-4.379ns, count = 323
Bin 8: 4.379ns-5.090ns, count = 386
Bin 7: 5.090ns-5.563ns, count = 444
Bin 6: 5.563ns-5.879ns, count = 238
Bin 5: 5.879ns-6.089ns, count = 210
Bin 4: 6.089ns-6.230ns, count = 92
Bin 3: 6.230ns-6.323ns, count = 42
Bin 2: 6.323ns-6.385ns, count = 10
Bin 1: 6.385ns-6.427ns, count = 5
```

Figure 18 – Delay profile, showing the ten slowest bins in the **miim** circuit

discussed in Section 4.5. The ability to trade logic for flip-flop placement flexibility is the focus of this optimization strategy.

Here is an algorithm to automate this approach:

- Move all non-critical flip-flops that have unique control signals out of congested areas of the circuit, provided this doesn't hurt performance.
- For each critical flip-flop **A,** find another critical flip-flop **B** that would benefit (i.e. make the circuit faster) from sharing a slice with **A** in the congested area of the circuit. Extract control signals for both flip-flops.
- Put both **A** and **B** in the same slice and repeat for other critical flip-flops
- When all critical flip-flops have been processed, move the non-critical flip-flops back into the congested areas, extracting their control signals only if they need to share a slice with a non-compatible flip-flop

6.4 Stopping Criterion

A key aspect of any automated iterative algorithm is to determine when to stop the iteration. As our primary goal in this work is to improvement the maximum clock frequency, this question becomes "how do we know when to stop trying to improve the speed of the circuit?"

In our use of the manual editor we inspected the distribution of the delay of paths in the circuit. We divided paths into bins based on delay. The first bin contains the slowest paths. It contains all the paths that needed to be improved to increase maximum circuit operating frequency by 1 MHz. The delay range of each consecutive bin was increased by 50% with respect to the previous bin, in a geometric fashion. We call these bins the *delay profile*, an example of which is given in Figure 18. We use the delay profile to determine when to stop improving the circuit.

Clearly it is easier to improve the speed of a circuit that has just a few critical paths in the slowest bin, rather than many. If there are many, then it will take gargantuan effort to gain any speed. We used the following criterion to stop optimization:

1. The two slowest bins contained 15 or more paths and
2. The paths in the two slowest bins were situated in close physical proximity to each other

The first criterion says that there is little point in continuing if there are too many paths that must be improved in order to gain speed. The second notices that close-to-critical paths pose a problem if improving one of them has a strong likelihood of increasing the delay of the other close-to-critical paths by virtue of their close physical proximity.

An example of the application of this strategy is shown in Figure 19. The circuit in Figure 19, **miim**, has the 15 slowest

Figure 19 – 15 slowest paths in the Miim circuit

paths (highlighted in red) in close proximity to one another. No further optimization of the circuit was performed, as none of our strategies could further improve the circuit in reasonable amount of time.

7. CONCLUSION

In this paper we have introduced a synthesis oriented manual editor, Augur, which provides a form of "omniscience" to help guide the user in the optimization process. On a set of 10 benchmark circuits we have achieved an average performance improvement of 9.9%, targeting a real, commercial FPGA. The knowledge we have gained though using the editor lead us to suggest optimization strategies that we believe can be automated.

In the future we plan to explore the possibility of automating the omniscient environment and applying it in the domain of FPGA architecture exploration.

8. ACKNOWLEDGEMENTS

The authors are grateful to William Chow for creating the base for this research. This research was funded by a grant from MICRONET under project S.4.T2 and Xilinx Inc.

9. REFERENCES

[1] J. Lou, W. Chen and M. Pedram, "Concurrent Logic Restructuring and Placement for Timing Closure," Proc. of the 1999 ACM/IEEE Int. Conf. on CAD, November 1999, pp. 31-35.

[2] J. Lin, A. Jagannathan and J. Cong, "Placement-Driven Technology Mapping for LUT-Based FPGAs," ACM/SIGDA Int. Symp. on FPGAs, February 2003, Monterey, California, USA, pp. 121-126.

[3] D. P. Singh and S. D. Brown, "Incremental Placement for Layout-Driven Optimizations on FPGAs," Proc. of the 2002 ACM/IEEE Int. Conf. on CAD, November 2002, pp. 752-759.

[4] J. Cong and Y. Ding, "FlowMap: an optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs," IEEE Trans. on CAD of Integrated Circuits and Systems, vol. 13, issue 1, 1994, pp. 1-12.

[5] T. Kutzschebauch and L. Stok, "Layout Driven Decomposition with Congestion Consideration," Proc. of

the 2002 Design, Automation and Test in Europe Conference and Exhibition, pp. 672-676.

[6] W. Chow and J. Rose, "EVE: A CAD Tool for Manual Placement and Pipelining Assistance of FPGA Circuits," Proc. of ACM/SIGDA Int. Symp. on FPGAs, February 2002, Monterey, California, USA, pp. 85-94.

[7] Xilinx Inc., "Virtex-E Production Product Specification", Online: http://direct.xilinx.com/bvdocs/publications/ds022.pdf, accessed on July 7, 2003.

[8] A. Roopchansingh and J. Rose, "Nearest Neighbour Interconnect in Deep Submicron FPGAs", Proc. of IEEE CICC, May 2002, pp. 59-62.

[9] S. Devadas, A. Ghosh and K. Keutzer, " Logic Synthesis," McGraw-Hill Inc., 1994, ISBN 0-07-016500-9

[10] A. Lu, H. Eisenmann, G. Stenz, and F. M. Johannes, "Combining Technology Mapping with Post-Placement Resynthesis for Performance Optimization," Proc. of Int. Conf. on Computer Design: VLSI in Computers and Processors, October 1998, pp. 616-621.

[11] R.J Francis, J. Rose, Z. Vranesic, "Technology Mapping Lookup Table-Based FPGAs for Performance" Proc. 1991 IEEE Int'l Conf. on CAD (ICCAD), Nov. 1991, pp. 568-571.

[12] Xilinx XC4000 device data sheet, Online: http://direct.xilinx.com/bvdocs/publications/4000.pdf. Accessed September 22, 2003.

[13] K. Schabas and S. D. Brown, "Logic Synthesis and mapping: Using logic duplication to improve performance in FPGAs," Proc. of ACM/SIGDA Int. Symp. on FPGAs, February 2003, Monterey, California, USA, pp. 136-142.

[14] S. Hauck, M. M. Hosler, and T. W. Fry, "High-performance carry chains for FPGAs," Proc. of the 1998 ACM/SIGDA 6th Int. Symp. on FPGAs, February 1998, Monterey, California, USA, pp. 223-233.

[15] J. Cong and Y. Hwang, "Boolean Matching for LUT-Based Logic Blocks With Applications to Architecture Evaluation and Technology Mapping," IEEE Trans. on CAD of Integrated Circuits and Systems, Vol. 20, No. 9, September 2001, pp. 1077-1090.

[16] T. Czajkowski, "A Synthesis Oriented Omniscient Manual Editor for FPGA Circuit Design," Master of Applied Science thesis, University of Toronto, 2004.