Multiplexer Restructuring for FPGA Implementation Cost Reduction

Paul Metzgen
Altera European Technology Center
High Wycombe
Buckinghamshire, UK
pmetzgen@altera.com

Dominic Nancekievill
Altera European Technology Center
High Wycombe
Buckinghamshire, UK
dnanceki@altera.com

ABSTRACT
This paper presents a novel synthesis algorithm that reduces the area needed for implementing multiplexers on an FPGA by an average of 18%. This is achieved by reducing the number of Lookup Tables (LUTs) needed to implement multiplexers. The algorithm relies on reimplementing 2:1 multiplexer trees using efficient 4:1 multiplexers. The key to the algorithm's performance lies in exploiting the observation that most multiplexers occur in busses. New optimizations are employed which pay a small cost in logic that is shared across the bus to achieve a reduction in the logic required for every bit of the bus.

Categories and Subject Descriptors
B.6.3 [**Logic Design**]: Design Aids – Automatic Synthesis, Optimization; J.6 [**Computer-Aided Engineering**]: Computer-aided design (CAD).

General Terms
Algorithms, Performance, Theory.

Keywords
FPGA, Multiplexers, Restructuring, Recoding, Busses, Logic Optimization, Synthesis.

1. INTRODUCTION
Multiplexers are a common building block for data-paths, and are used extensively in a number of applications including processors[1], processor busses, network switches, and even DSP designs with resource sharing. Analysis of the Altera Benchmark set of 120 real customer designs[2] has estimated that multiplexers typically account for over 25% of the area of an FPGA design. Indeed, optimizing a design for an FPGA is often the problem of optimizing the multiplexers.

This paper presents a novel *multiplexer restructuring* algorithm for reducing the implementation area for multiplexers in a 4-input lookup table (4-LUT) based FPGA architecture[3]. Benchmark results will show average overall area reductions of 4.5% which is equivalent to a reduction of 18% in the area needed for multiplexers.

DAC 2005, June 13–17, 2005, Anaheim, California, USA.

HDL code (e.g. from VHDL or Verilog) can be synthesized into simple logic gates; with logic optimization[4] and technology mapping[4][5] being applied to the resulting netlist. Rather than working with a netlist of simple logic gates, multiplexer restructuring relies on the creation of busses of multiplexer trees with identical structures. Logic optimizations can change multiplexer structures or reduce them to simple logic gates, and will prevent busses from forming. For this reason, multiplexer restructuring benefits from being performed before most other logic optimizations.

Section 2.1 shows how 2:1 multiplexers (rather than simple logic gates) can be synthesized directly from behavioural HDL code. Sections 2.2 and 2.3 show how designs often create trees of multiplexers and busses of multiplexers, and section 2.4 shows how these structures are implemented using 4-LUTs, and describes an efficient implementation of a 4:1 binary multiplexer using just two 4-LUTs.

Section 3 introduces a new technique called *Compression* which is able to re-implement groups of 2:1 multiplexers as efficient 4:1 multiplexers at the cost of some additional control logic. This results in a reduction in the number of 4-LUTs needed to implement the multiplexers for every bit in the bus, however the additional cost in control logic can be shared across the entire bus. Optimizing entire busses of multiplexers is the key to the multiplexer restructuring algorithm.

Section 4.1 describes how the multiplexer restructuring algorithm constructs busses of multiplexer trees from a design. The restructuring technique in section 4.2 is used to perform the Balancing algorithm defined in section 4.3. *Balancing* increases the number of efficient 4:1 multiplexers that can be generated during *Compression*.

Section 5 summarizes the overall algorithm which has been implemented in Altera's Quartus II integrated synthesis. Section 6 shows the results achieved across Altera's benchmark set yielding area reductions of over 20%.

2. BACKGROUND
2.1 Multiplexer Synthesis from HDL
It is not common for multiplexers to be instantiated explicitly in HDL code, and it can be difficult to reliably infer multiplexers from simple logic gates. However, some HDL constructs can be synthesized directly to a multiplexer based representation. This

section shows the type of HDL code that generated multiplexers for the restructuring algorithm.

Figure 1 shows an example of a case statement in VHDL, together with its implementation as a tree of 2:1 multiplexers. A similar result is achieved with a Verilog case statement using the 'parallel case' directive. Case statements can have any number of cases, limited only by the bit-width of the select expression (sel); large cases result in deeper trees of 2:1 multiplexers.

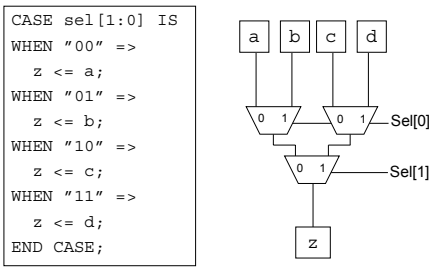

Figure 1: Logic generated from a case statement

Note that not all case statements describe all possible cases explicitly, and may rely on a 'default' or 'others' condition to be described also. In these cases, it is still possible to represent the case statement as a tree of 2:1 multiplexers, although the tree may no longer be balanced.

Figure 2 shows how 'if-then-else' statements generate a chain of 2:1 multiplexers. Verilog's '?:' and non-parallel cases also generate a similar structure. Note that the chain of multiplexers ensures that if the first if-condition is true, then the 'a' data input will be selected, and all the remainder of the multiplexers will be ignored.

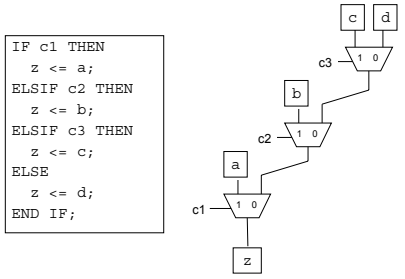

Figure 2: Logic generated from an if-then-else statement

2.2 Multiplexer Trees

It is common in HDL designs for if-then-else and case statements to be nested within each other. For example, the HDL in Figure 3 shows an if-then-else statement surrounding a case statement. This will lead to multiplexers feeding other multiplexers, which will be termed multiplexer trees.

A multiplexer tree is defined as a fanout-free cone of logic containing only 2:1 multiplexers, and with no internal reconvergent fanouts (i.e.: a tree rather than a directed acyclic graph). In the set of all possible multiplexer trees in a design, the set of *maximal* multiplexer trees is defined as a subset of those trees which are not contained in any other.

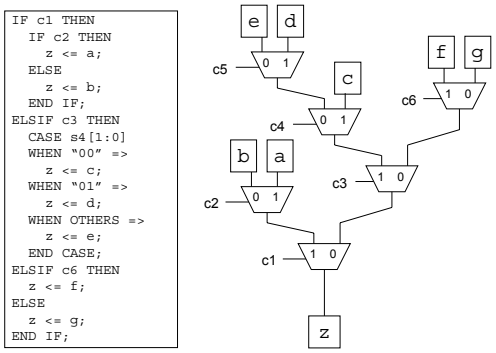

Figure 3: Multiplexer tree generated from HDL

Recognizing large multiplexer trees will be important in order to maximize the area saving achieved by the multiplexer restructuring algorithm. This is described further in Section 3.

2.3 Multiplexer Busses

A VHDL Signal or Verilog wire is often more than one bit wide. When used in if-then-else and case statements, this creates a number of identical multiplexer trees with different data inputs, an example of which is shown in Figure 4.

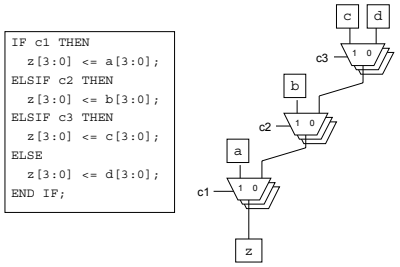

Figure 4: Generating a bus of multiplexers

A set of multiplexer trees with identical structures is called a *bus of multiplexer trees*. The multiplexer restructuring techniques presented in this paper take the novel approach of optimizing entire busses of maximal multiplexer trees.

A *bus* of maximal multiplexer trees is defined as the set of all maximal multiplexer trees with identical structures. Two multiplexer trees have identical structures when they have the same arrangement of 2:1 multiplexers, with each corresponding pair of 2:1 multiplexers have exactly the same control inputs. The techniques used for identifying and forming busses are described further in section 4.1.

Multiplexer restructuring should be performed early in the synthesis flow, so as to ensure that individual 2:1 multiplexers are neither moved nor reduced to logic gates in local optimizations that do not consider their effect on busses. The average bus width found in the Altera set of benchmarks was 14 bits wide.

2.4 The Cost of Multiplexers

Conventional technology mapping techniques[4][4] are unable to pack the structures shown in Figure 1 and Figure 2 into less than three 4-LUTs, even if the 2:1 multiplexers are reduced to simple logic gates. This means that every 2:1 multiplexer tends to require a separate 4-LUT, despite the fact that only 3 of the 4 inputs of those 4-LUTs may ever be used.

This section shows an implementation of a 4:1 multiplexer that requires just two 4-LUTs (rather than three 4-LUTs). This

implementation has been chosen for this paper as it does not rely on any special hardware features to implement, although any equally efficient implementation of a 4:1 can be used in the restructuring algorithm.

Figure 5 shows the operation of the area efficient 4:1 multiplexer when the control input, S0, is held low. The 4:1 multiplexer is realized using two 4-LUTs (shown as shaded boxes) chained together. Each 4-LUT is configured to behave as though it contains the logic shown inside the shaded boxes. The multiplexer has four inputs, a, b, c and d, and two control inputs, S0 and S1, shown in the white boxes.

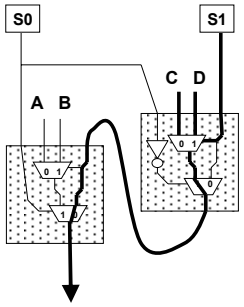

Figure 5: An efficient 4:1 multiplexer in two 4-LUTs (S0=0)

When S0 is low, the lower control bit, S1, selects between inputs c or d, the result of which is passed to the second 4-LUT where it bypasses the a or b inputs.

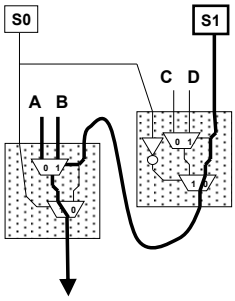

Figure 6: An efficient 4:1 multiplexer in two 4-LUTs (S0=1)

Figure 6 shows the operation of the efficient 4:1 multiplexer when the control input, S0, is high. The lower control bit, S1, now bypasses the 'C' or 'D' inputs to go direct to the second 4-LUT where it selects between the 'A' or 'B' inputs, the result of which is now passed to the output of the second 4-LUT.

The *Compression* algorithm described in section 3 achieves an area reduction by reimplementing groups of 2:1 multiplexers using this efficient 4:1 multiplexer.

3. COMPRESSION

Compression is the process of converting groups of 2:1 multiplexers into the more area efficient 4:1 multiplexer implementation described in Section 2.4. Figure 7 and Figure 8 will show that any group of three 2:1 multiplexers can be converted into a 4:1 multiplexer. However, additional control logic is needed to convert the control lines into the two-bit binary control needed for the 4:1 multiplexer.

The multiplexer restructuring algorithm takes the new approach of converting entire busses of multiplexers so that the control logic can be shared between all multiplexer trees in the bus, whilst the benefit of the improved multiplexer implementation can be realized for every multiplexer tree in the bus. This is summarized in Equation 1.

$$\underbrace{W}_{\text{bus bit width}} \cdot Area_{before} \longrightarrow W \cdot Area_{after} + \underbrace{C}_{\text{control logic}}$$

Equation 1: Area Reduction Estimate for Compression

Without considering busses, this type of transformation would seldom yield an area benefit as the extra control logic can outweigh any advantage gained by the more efficient multiplexer implementation.

Figure 7 shows the how a priority-chain of three 2:1 multiplexers can be converted into a 4:1 binary multiplexer. This conversion requires at most two additional 4-LUTs of control logic. (Fewer LUTs may actually be needed to implement the control logic, either because it may pack with existing LUTs or the control lines are not independent logic functions). Given that the original cost of the multiplexers would be at least three 4-LUTs, and the 4:1 multiplexer can be implemented using just two, Equation 1 shows that this transformation yields a net area saving for busses of width W > 2 (W = 2 is area neutral).

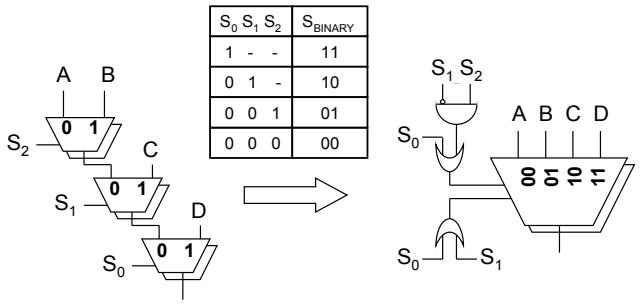

Figure 7: Recoding a 2:1 multiplexer triplet (linear)

Figure 8 shows how a group of three 2:1 multiplexers arranged as a tree can be converted into a 4:1 multiplexer. In this case, by carefully choosing the encoding for the 4:1 multiplexer so that S0 selects between (A,B) and (C,D), it is possible to restrict the additional control logic to at most just one 4-LUT. In this case, Equation 1 shows that this transformation yields a net area saving for busses of width W > 1.

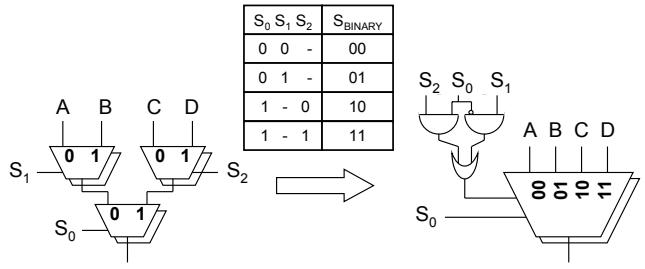

Figure 8: Recoding a 2:1 multiplexer triplet (tree)

Using the transformations in Figure 7 and Figure 8, any cluster of three 2:1 multiplexers can be converted into a 4:1 multiplexer. However, both these transformations must be performed on busses of multiplexers of width 2 or more in order to reduce the overall number of 4-LUTs needed.

Recognizing 2:1 multiplexer triplets within busses is the core of the multiplexer restructuring algorithm. The next section will

present some novel techniques to maximize the number of triplets that can be formed.

4. BUSSES OF MULTIPLEXERS

4.1 Construction of the Depot

Compression gives an area reduction for every 2:1 multiplexer triplet it recodes; hence *Compression* works best when it is applied to wide busses of large multiplexer trees. The set of all busses of multiplexer trees is called the Depot. This section shows the Depot can be constructed.

Note that any given 2:1 multiplexer can be part of at most one maximal multiplexer tree, and when the output of one 2:1 multiplexer feeds the input to another and nothing else they are both part of the same maximal multiplexer tree. Using these observations it is possible to construct the set of all maximal multiplexer trees in a linear pass of the 2:1 multiplexers in a design.

By sorting the list of maximal multiplexer trees based on their structure, trees with identical structures will be adjacent in the resulting list. Busses formed from trees with identical structures can then be constructed in a linear pass of the sorted list.

4.2 Restructuring

Section 4.3 will describe *Balancing* which aims to maximize the area reduction that is achievable through *Compression*. However, *Balancing* is based on Restructuring, which is defined in this section.

Restructuring moves a 2:1 multiplexer together with one of its inputs through the 2:1 multiplexer that it fed. Figure 9 shows the restructuring of the shaded multiplexer together with its 'f' input. In order to maintain the functionality of the original bus of multiplexers, some additional control logic is needed. Once again, this control logic can be shared for every multiplexer tree in the bus.

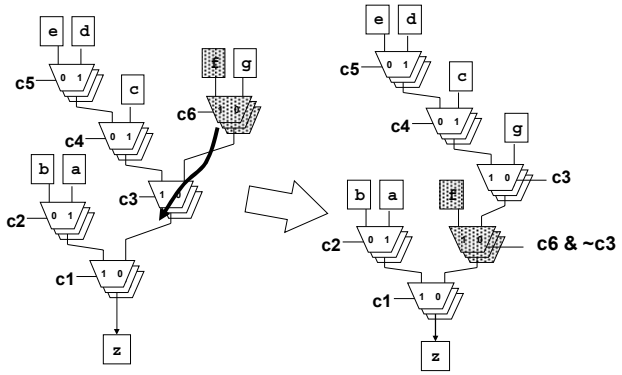

Figure 9: Restructuring a bus of multiplexer trees

For Figure 9, the recode logic (c6 AND (NOT c3)) must ensure that input 'f' is selected when c1 is false, c3 is false and c6 is true (as was the case before the transformation). Likewise, note that input 'g' is selected when c1 is false, c3 is false, and c6 is false, which means that (c6 AND (NOT c3)) is also false.

The restructuring transformation moves the selected multiplexer one step further towards the head of the multiplexer tree. Hence, by repeatedly applying the restructuring transformation, it is possible to move any number of places further towards the head.

4.3 Balancing

Compression is able to reduce the number of 4-LUTs needed to implement a bus of multiplexers by converting 2:1 multiplexer triplets. However, there are some structures for which it is not possible to get the best triplet clustering without restructuring the multiplexer trees slightly. For example, it is not possible to cluster all the 2:1 multiplexers in Figure 10, as every possible clustering leaves a 2:1 multiplexer stranded.

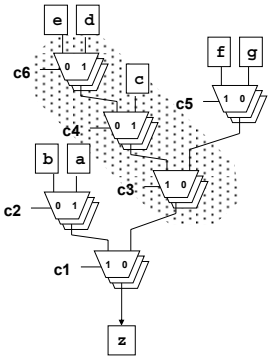

Figure 10: Multiplexer tree with sub-optimal Compression

Balancing aims to perform a minimal amount of restructuring in order to achieve the best *Compression*. For the example shown in Figure 10, restructuring would be applied as shown in Figure 11 below to achieve two sets of 2:1 multiplexer triplets.

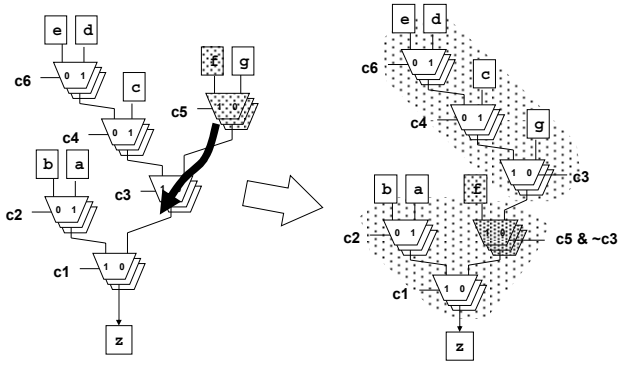

Figure 11: Balancing to improve Compression

The *Balancing* algorithm is defined recursively as shown in Figure 12, and results in an algorithm runtime linear in the tree size. Starting at the head multiplexer, the algorithm first balances the multiplexer trees on its left and right data inputs. The *Balancing* function returns the number of 2:1 multiplexers that have not yet been combined into triplets. *Balancing* will guarantee that as many triplets are formed as possible, so there will only ever be zero, one or two 2:1 multiplexers left over.

At any stage during *Balancing* there will be between one and five 2:1 multiplexers that need to be balanced (up to 2 from the left branch and 2 from the right branch as well as the current 2:1 multiplexer itself), . With one or two multiplexers, no additional triples can be formed, and any combination of three multiplexers can always be formed into a triple. However four or five multiplexers may need some restructuring in order to not leave any stranded multiplexers and this restructuring is shown in Figure 13 and Figure 14 respectively.

```
Function Balance( m )
  if (is_a_multiplexer(m)) then
    num_muxes = 1          // Count this mux
      + Balance(m.left)
      + Balance(m.right);
    if (num_muxes >= 3) then
      if (num_muxes == 4) then
        Apply transformation shown in Figure 13;
      else if (num_muxes == 5) then
        Apply transformation shown in Figure 14;
      end if
      // remove muxes in triplet
      num_muxes = num_muxes – 3;
    end if
    return num_muxes;
  else
    return 0;   // input to multiplexer tree: 0 muxes
  end if
```

Figure 12: The balancing algorithm

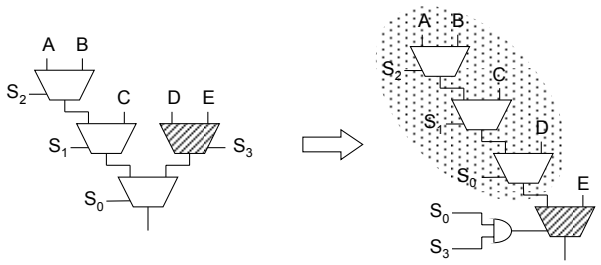

Figure 13: Restructuring 4 multiplexers for Balancing

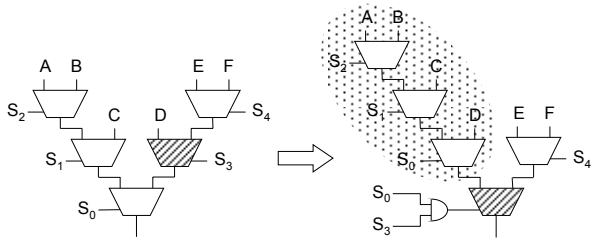

Figure 14: Restructuring 5 multiplexers for Balancing

Note that there are other types of restructuring that could be performed instead. For instance, the A-B multiplexer in Figure 13 could be moved to the head instead of the D-E multiplexer. The restructuring in Figure 13 and Figure 14 was chosen so as to (locally) minimize the amount of additional control logic. A different choice of restructuring could lead to a smaller implementation in LUTs depending on what logic feeds the multiplexer control signals, but this was not explored.

5. ALGORITHM SUMMARY

This paper has introduced the notion of busses of multiplexer trees, and described *Balancing* which can be used to improve the performance of *Compression*, which reimplements 2:1 multiplexer triplets into more efficient 4:1 multiplexers. Figure 15 shows the overall multiplexer restructuring algorithm.

```
Convert_Multiplexers_to_2:1s ( )
  Form_Multiplexer_Trees( )
  Merge_Multiplexer_Trees_into_Busses( )
  Foreach bus {
    Balance(bus) // Minimally rearrange 2:1 into triples
    Compress(bus) // Convert triplets into efficient 4:1's
  }
```

Figure 15: The multiplexer restructuring algorithm

The quality of the Multiplexer Restructuring algorithm relies upon recognizing large busses of multiplexer trees. Because the multiplexer restructuring algorithm relies on sharing control logic across an entire bus, the wider the busses, the greater the benefit from *Compression*. Optimizations that might reduce the similarity between multiplexer trees, and hence reduce the size of the busses found, should be avoided until after multiplexer restructuring.

Multiplexer restructuring begins by decomposing any large multiplexers into 2:1 multiplexers. All the 2:1 multiplexers in the design are used to form multiplexer trees as described in section 4.1. Multiplexer trees with similar structures are then merged to form busses as described in section 4.1.

The main part of the algorithm optimizes each bus in turn. *Balancing* rearranges the 2:1 multiplexers into triplets, so that each triplet can be recoded to the efficient 4:1 multiplexer during the *Compression* stage.

6. RESULTS

The algorithms presented in this paper have been integrated into Altera's Synthesis software within Quartus II 4.2. Figure 16 shows the resulting area reduction when applied to all 120 real customer designs in Altera's benchmarking suite.

The results show a number of designs that have achieved a 20% reduction in the number of LUTs required to implement them, and over 40% of the designs experienced a reduction in area of over 5%. The mean improvement across all designs is 4.5%.

It has been estimated that around 25% of all LUTs were being used to implement multiplexers, and multiplexer restructuring is only able to optimize this 25%, so an average of 4.5% means that multiplexers have been reduced by 18% on average.

Figure 16 shows that only 8% of designs have increased in size (3.6% at worse). Initial investigations show that the majority of these cases are due to inefficiencies introduced by multiplexers with related or constant data inputs, which may be exploited better by traditional logic optimization techniques. Also, Restructuring tends to push surplus 2:1 multiplexers (that cannot be combined into a 4:1 multiplexer) to the head of the multiplexer tree, however in some cases those multiplexers may have been able to pack with logic at the inputs to the multiplexer trees.

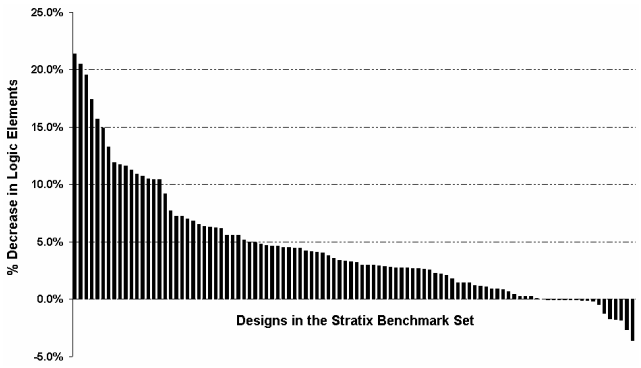

Figure 16: Multiplexer restructuring benchmark results

Although the focus of Multiplexer Restructuring was primarily on area (i.e.: reducing the number of 4-LUTs needed), the average speed of the circuits was reduced by only 0.9% (measured from the slowest clocks in each design). The scatter-plot in Figure 17 shows that there is no discernable relationship between the size of the area reduction and the change in speed of the resulting circuit. The restructuring transforms used tend to increase the depth of 2:1 multiplexers in a tree and this should impact speed, however, reducing the number of LUTs needed will also tend to reduce the average routing distance (which accounts for 50% of the overall delays) and hence improve speed.

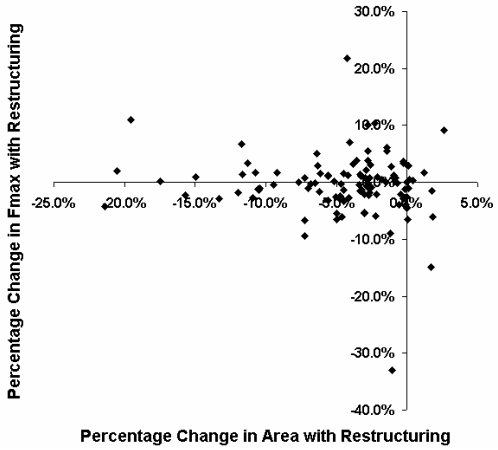

Figure 17: Correlation between Area and Fmax

7. CONCLUSION

This paper has presented the Multiplexer Restructuring algorithm which is able to reduce the number of 4-LUTs needed to implement the multiplexers in a design by an average of 18%.

The key to the multiplexer restructuring algorithm is to optimize across busses of multiplexers. New optimizations allow area reductions to be made in every bit in the bus, even at the expense of additional control logic. This is because control logic can be shared across a bus, and hence its cost can be amortized.

Compression converts triplets of 2:1 multiplexers into efficient 4:1 multiplexers. Although additional LUTs may be needed to recode the control lines for the 4:1's, this cost can be amortized by applying *Compression* to entire busses of multiplexers. In addition, by building maximal multiplexer trees, it is possible to convert a large number of 2:1 multiplexer triplets in this way.

This paper has also introduced *Balancing* which improves the effectiveness of the *Compression* algorithm. *Balancing* modifies the structure of the multiplexers slightly, so as to maximize the number of triplets that can be recoded by *Compression*.

This paper describes algorithms that are targeted to an FPGA based on 4-LUTs, such as Altera's Stratix I or Cyclone devices. New FPGA architectures allow variable sized lookup tables (such as the Stratix II architecture which is able to implement 4, 5, or 6 input LUTs)[8], and it may be possible to get even better area reductions by extending the techniques presented in this paper to exploiting these architectures. This is the subject of current research.

8. REFERENCES

[1] *A High Performance 32-bit ALU for Programmable Logic.* P. Metzgen. Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field Programmable Gate Arrays. Pp 61-70. 2004.

[2] FPGA Performance Benchmarking Methodology, White Paper, www.altera.com

[3] *The Stratix Device Handbook (Vol 1).* Altera Corporation, 2004.

[4] E. M. Sentovich et al. "*SIS: A System for Sequential Circuit Synthesis*." Technical Report, University of California at Berkeley, 1992, Memorandum No. UCB/ERL M92/41

[5] J. Cong and Y. Ding, "FlowMap: An Optimal Technology Mapping Algorithm for Delay Optimization in Lookup-Table based FPGA Designs", IEEE Trans. CAD Vol 13, No 1, pp. 1-12, 1994.

[6] V. Manohararajah, S.D. Brown and Z. Vranesic, "Heuristics for Area Minimization in LUT-Based FPGA Technology Mapping", in Proc. of the Int'l Workshop on Logic Synthesis 2004

[7] D. Lewis et al, C. Wysocki and R. Cliff, "The Stratix Routing and Logic Architecture", in Proc. ACM/SIGDA Int'l Symposium on FPGAs (FPGA 2003), pp.12-20, 2003

[8] David Lewis et al, "The Stratix-II Routing and Logic Architecture". 2005 Int'l Symposium on FPGAs (FPGA 2005)