A VERILOG RTL SYNTHESIS TOOL FOR HETEROGENEOUS FPGAS

Peter Jamieson, Jonathan Rose

Edward S. Rogers Sr. Department of Electrical and Computer Engineering
University of Toronto
Toronto, Ontario, Canada M5S 3G4
jamieson@eecg.toronto.edu, jayar@eecg.toronto.edu

ABSTRACT

Modern heterogeneous FPGAs contain “hard” specific-purpose structures such as blocks of memory and multipliers in addition to the completely flexible “soft” programmable logic and routing. These hard structures provide major benefits, yet raise interesting questions in FPGA CAD and architecture. To develop high-quality CAD mapping algorithms for these structures, and indeed to measure the quality of proposed new structures in the architectural domain, it is essential to have a flexible tool at the RTL synthesis level that permits heterogeneous FPGA CAD and architecture experimentation. In this paper we present a synthesis tool, called Odin, and an algorithm that permits flexible targeting of hard structures in FPGAs. Odin maps Verilog designs to two different FPGA CAD flows: Altera’s Quartus, and the academic VPR CAD flow. We have expended significant effort to make the quality of this tool comparable to an industrial front-end synthesis tool, and we present mapping results for our benchmarks that show the quality of our results.

1. INTRODUCTION

The advent of Field-Programmable Gate Arrays (FPGAs) with specific-purpose heterogeneous structures [1, 2, 3] raises the need for quality Computer Aided Design (CAD) algorithms to target these structures and the need for flexible architecture exploration aiding in the selection of appropriate structures to place on modern FPGAs. While traditional “soft” logic mapping into Lookup Tables (LUTs) [4, 5] is done after technology-independent logic optimization [6, 7, 8] the mapping into coarse-grain structures such as multipliers and memories is much more appropriately done at the Register Transfer Level (RTL) synthesis level where these structures are more directly recognizable.

The purpose of this paper is to present a public-domain open source Verilog RTL Synthesis tool, called Odin, the algorithms that allow it to flexibly target different heterogeneous structures, and the approximate parity with the quality of industrial-strength RTL synthesis tools. Odin can be used as part of a CAD flow that connects to a commercial flow and one commonly-used academic flow.

The remainder of this paper introduces Odin and shows how Odin compares against Quartus’ synthesis tool. Section 2 defines relevant terminology for heterogeneous FPGAs and describes previous work on high-level synthesis including existing front-end synthesis tools. Section 3 describes the basic structure of Odin, including its’ ability to receive a simple description of a specific-purpose hard structure, and a basic mapping algorithm for those structures. Section 4 describes mapping techniques within Odin that raise the quality of its results. We present comparison results for Odin versus Altera’s Quartus RTL synthesis tool in Section 6, and we also show how the combination of mapping techniques in Odin result in smaller and faster designs. Finally, Section 8 concludes our paper.

2. BACKGROUND

2.1. Definitions for Modern FPGAs

A basic FPGA consists of a routing fabric and base logic units. Base logic units are most typically implemented as LUTs [9], but NAND gates [10], transistors [11], and multiplexers [12] have also been used. A *homogeneous* FPGA is an FPGA that consists of only base logic units and programmable routing. The base logic unit or a group of base logic units (sometimes referred to as a cluster) is collected together with programmable routing to form a *tile*. Tiles are abutted together to form the soft fabric of an FPGA, which can implement any logic function.

A *heterogeneous* FPGA contains *hard* specific-purpose circuits in addition to base logic units and a routing fabric. A hard circuit is a specific circuit included on an FPGA to perform specific logic functions, *which could also be implemented using the base logic units and the routing fabric* [13].

Given these definitions, there are two common ways hard circuits are added to FPGAs. The first is to add specific circuits into all the tiles in an FPGA. In this case, all the tiles are the same, and the FPGA remains a homogeneous array of tiles. We call this *soft fabric heterogeneity*. The hard circuits within each tile allow certain functions to be implemented with better area-efficiency and faster speed. For example, flip flops are commonly paired with LUTs to save both speed and area in, arguably, all cases [14] compared to implementing the flip flop in the base soft logic fabric. We call a unit of soft fabric heterogeneity that consists of the base logic unit and additional hard circuits a Soft Fabric Logic Unit (SFLU). The SFLU is called a Logic Element (LE) by Altera and a slice by Xilinx.

The second way to employ hard circuits in an FPGA is to include a specific circuit as a differentiated tile, which separately abuts the SFLU tiles. We call this *tile-based heterogeneity*. Examples of heterogeneous tiles are multipli-

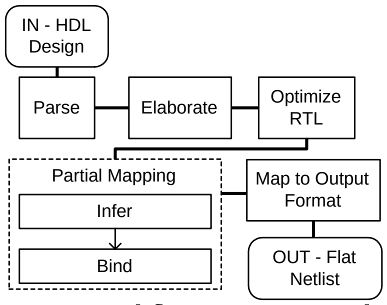


Fig. 1. This is a general flow to convert Verilog designs to logic netlists.

ers [1, 2, 3], which are large circuits added to an FPGA as unique tiles.

2.2. Hardware Description Language (HDL) Synthesis

Typically, the CAD flow is broken into a series of smaller tasks, and in this paper we focus on the front-end conversion of an HDL design into a netlist of basic gates and more complex logic functions. In this section we discuss the general approach taken in these front-ends.

Figure 1 illustrates several steps that may take place in a front-end synthesis tool. The initial input to the tool flow is an RTL design written in an HDL, and the output is a netlist of basic gates, flip-flops, and higher-level structures that serve to target specific hard circuits.

After basic language parsing, an elaborator transforms the HDL design into a netlist, which consists of input and output ports, primitive logic gates, specific functions such as arithmetic operations, memory units including registers and memory blocks, and HDL control structures. In most schemes, elaboration preserves information from the original HDL design, meaning that instead of converting constructs like an arithmetic operation into logic gates, the arithmetic operation is preserved.

After elaboration, a partial mapper converts the netlist to a lower-level netlist that more closely conforms to both the FPGAs' hard circuits and soft logic fabric. Given a library of predefined components available on a target FPGA, the partial mapper decides whether parts of the high-level netlist should map to tile-based heterogeneous structures, soft fabric heterogeneity, or logic gates. Partial mapping performs this mapping through two tasks. Firstly, an inferencing stage tries to determine the function of each part of the circuit, and secondly, partial mapping binds functions in the design to an implementation on the FPGA. After partial mapping, all elements of the netlist bind to either hard circuits on the FPGA, to logic gates that must be mapped to the FPGAs' soft logic fabric, or a mixture of both.

The partial mapping stage, in essence, performs some of the FPGA technology-mapping [15, 5, 4]. We call front-end technology mapping, partial mapping, since only some of the netlist is bound to target technology, while the rest of the netlist remains unbound. Partial mapping performed during front-end synthesis can be advantageous since the additional information that is present at this stage makes inferencing easier than it would be at later stages of the CAD flow.

There are a number of commercial front-end synthesis tools that target FPGAs. Both Altera and Xilinx incorporate front-end synthesis into their FPGA CAD flow tools called Quartus [16] and ISE [17] respectively. Other popular front-end synthesis tools for FPGAs include Synplify [18], Blast FPGA [19], LeonardoSpectrum [20], and Design Compiler FPGA [21].

To our knowledge there are no available open source front-end synthesis tools, but some HDL parsers do exist. For example, Icarus [22] is a front-end Verilog parser, which does have a simple back-end implementation that targets Xilinx's defunct "xnf" netlist.

3. OVERVIEW OF ODIN

In this work we present, Odin, an HDL synthesis tool that takes a Verilog design as input and converts the Verilog design into a flattened structural netlist. This conversion process includes elaboration and partial-mapping stages, where the final netlist consists of primitive gates and complex logic functions targeting heterogeneous structures on an FPGA. This netlist can be passed into various CAD flows including Quartus, ModelSim, and a VPR CAD flow, although the latter cannot include complex logic functions.

Figure 1 shows the major stages of the Odin tool. First, a front-end parser, Icarus [22], parses the Verilog design and generates a hierarchical representation of the design.

Second, Odin has an elaboration stage that traverses the intermediate representation of a design to create a flat netlist that consists of structures including logic blocks, memory blocks, *if* and *case* blocks, arithmetic operations, and registers. Each of these structures within the netlist we refer to as a node in the netlist.

Third, some simple synthesis and mapping is performed on this netlist. This includes examining adders and multipliers for constants, collapsing multiplexers, and detecting and re-encoding finite state machines to one-hot encoding. These mappings are discussed in more detail in the next Section.

Fourth, an inferencing stage searches for structures in the design that could be mapped to hard circuits on the target FPGA. These structures are connected sub-graphs of nodes that exist in the design netlist. One of the inputs to the Odin tool that allows it to flexibly target this and like structures, is a text file that describes the node subgraph to search for this kind of hard circuit mapping. We search for these structures in the netlist using a matching algorithm discussed in Section 4.1.

Fifth, a binding stage guides how each node in the netlist will be implemented. This is done by mapping nodes in the netlist to either hard circuits, soft programmable logic, or a mixture of both. One way to do this is to map structures to Library Parametrized Modules (LPMs), which later stages of the industrial CAD flow will bind to an implementation on the FPGA whether that be a hard or soft implementation.

The output from Odin is a flat netlist consisting of connected complex logic structures and primitive gates.

4. MAPPING TECHNIQUES IN ODIN

One of the key goals in this work is to create a tool that achieves quality of results comparable to an industrial front-end synthesis tool. To achieve this we present several of the key optimizations. First and foremost, we discuss the partial

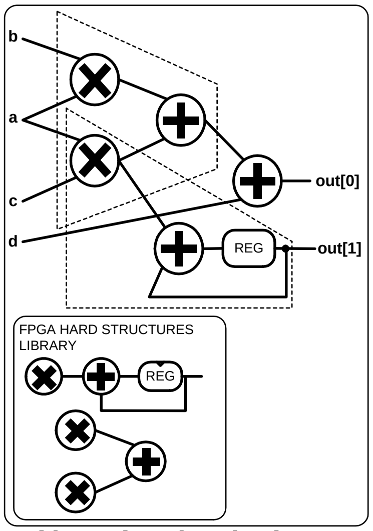


Fig. 2. Shows a library describing hard circuits on an FPGA and a matching of these structures in a netlist.

mapper steps that explicitly target the tile-based heterogeneous structures. In subsequent sections we discuss arithmetic optimizations, finite state machine re-encoding, and multiplexer collapsing optimizations.

4.1. Mapping to Tile-based Heterogeneity on FPGAs

Proper use of hard circuits is important since they can considerably decrease area and improve speed. The most common tile-based hard circuits are multipliers and block memories.

Memories and multipliers are easily identified in designs, and these structures are mapped to LPMs (or similar macro structures depending on the device family target) that later CAD stages configure for the target FPGA. Multipliers are instantiated in Verilog with the "*" symbol, which is treated as single node within the netlist. Similarly, memories are single nodes and are easy to identify within Verilog designs. Since both of these structures consist of one node, no additional processing is needed, and each node is mapped to an LPM representing the FPGAs hard circuit.

If a hard circuit has additional (typically programmable) functionality, it is a more difficult problem to detect the possible structures that can make use of it. For example, the Altera DSP block can be configured to implement multiplication, multiply accumulation, and multiply summation. Multiply summation is the addition of two or more multiplications.

The inferencing stage in Odin detects structures that map to hard circuits, such as these multiply accumulates and multiply summations. Once the text file that describes the functionality of a hard circuit is read in (for example the Altera Stratix DSP block), a matching algorithm searches for instances of the DSP block in the netlist of the design. During the binding stage, each matched node in the design netlist is bound to a specific function available in the DSP block. Currently, if a node can be bound to multiple DSP block functionalities, then the match which covers the most nodes in the design is chosen.

Figure 2 shows a sample netlist and a technology library, the latter of which is described by the input text file. In the Figure, the dotted lines represent matchings of the hard circuits within the library.

This matching problem is a form of sub-graph isomorphism, which has been used in instruction generation for reconfigurable and processor systems [23, 24] and sub circuit extraction for technology mapping [25].

We have found matching to be simple and successful as the size of the sub-graphs representing hard circuits is very small and consist of at most five nodes - which makes the matching very fast. Secondly, while matching DSP blocks we use multiply nodes as a seed node to start each search, and multiplies appear infrequently in designs.

4.2. Mapping to Soft Fabric Heterogeneity on FPGAs

To achieve industrial-quality results on modern FPGAs it is essential to correctly target the special-purpose adder structures that exist on all modern FPGAs and the features of the flip-flops that are provided within the soft fabric of the FPGA. We assume, for this discussion, that the target SFLUs have one register and can implement either a logic function or one or more bits of addition or subtraction. This is similar to the SFLUs on modern FPGAs available from both Altera and Xilinx.

The important part when mapping flip-flops is detecting logic that can be used for the enable and reset signals present on the flip-flop. To make use of the special adder logic in modern FPGAs [1, 2] all possible uses of addition and subtraction must be detected and mapped to that special logic.

4.3. Mapping other Structures to an FPGA

4.3.1. Arithmetic Optimizations

For multiplies and additions, it is possible to shrink the size of the arithmetic operation through constant propagation. These operations should be done at RTL synthesis instead of logic synthesis since the results of this optimization affects partial mapping decisions.

Other optimizations include replacing "0" low order constant inputs to an adder with a wire, and similarly, removing most significant "0" constant multiplier inputs. We also implement $A + B + 1$ in one adder using the carry in signal that commonly exists in the adder capabilities of an SFLU.

4.3.2. One-hot Re-Encoding of Finite State Machines

Another beneficial transformation is the re-encoding of finite state machines to one-hot encoded. In Odin, state machines are first detected, and then states are re-encoded to a one hot-encoding. This results in one flip-flop per state, which reduces the number of bits needed to be written during each calculation of the next state. Since less bits are written per cycle, the amount of multiplexing and routing is also reduced, which in many cases makes the finite state machine implementation both faster and smaller [26]. The additional flip-flops needed for each state are available on FPGAs in each SFLU

We identify state machines by checking if 3 characteristics of a potential state machine are satisfied. These characteristics are:

```
Verilog HDL
module control (clock, a, b, reg1, reg2);
        input [1:0]a, [3:0]b, clock;
        output [1:0]reg1, reg2;
        reg [1:0]reg1, reg2;
        always @(posedge clock)
                case(a)
                2'b00: reg1<={b[0], 1,b1}
                2'b01: reg1<={b[1], b[2]}
                2'b10: reg1<={b[2], b[3]}
                2'b11: reg1<={b[3], 1'b0}
                endcase
        always @(posedge clock)
                if (a == {b[0], b[1]})
                        reg2 <= reg1[0];
                else if (b == {3'b0, b[1])
                        reg2 <= reg1[1];
endmodule
```

Fig. 3. Example of control statements in Verilog, which become multiplexers.

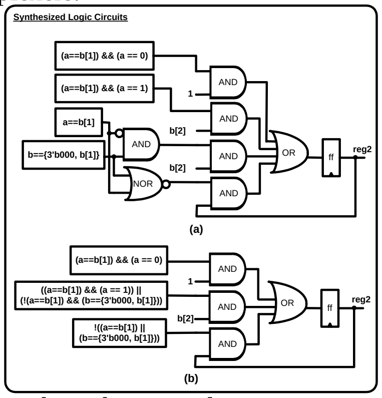


Fig. 4. Shows how the example in Figure 3 collapses into one implementation.

1. There is a case statement in a Verilog combinational *always* block.

2. The signal being compared in the combinational case statement comes from a register. This register is the state register.

3. There is a feedback loop to the state register through a multiplexer, and the only inputs to this multiplexer come from either a feedback loop from the state register or constant inputs that represent state encoded values.

These rules are sufficient, but not necessary, and therefore, not all finite state machines will be detected using these characteristics.

4.3.3. Multiplexer Collapsing

Multiplexers are frequently used in circuit designs. Figure 3 shows Verilog code which has both a *case* and an *if* structure. Both *case* and *if* structures are implemented as multiplexers.

Beyond these simple implementations of control structures, we can collapse *case* and *if* structures together and group common signals together. Figure 4(a) shows how the multiplexers in Figure 3 can be collapsed, and Figure 4(b) shows how the common inputs of the multiplexers can be joined. Both transformations increase the depth of logic for the control signals, with the benefit of amortizing logic and decreasing the number of logic levels on the data path. Currently, Odin collapses multiplexers assuming that the additional logic complexity added to the control signals will not affect the overall speed of the design. This is not always the case, and to improve multiplexer collapsing, we need to estimate path delays of the circuit and determine if the additional delay on the control paths will effect the overall speed of the design.

5. CAD FLOW AND VERIFICATION

Odin is a front-end HDL synthesis tool that is designed to target both industrial and academic CAD flows. Specifically, Odin can interface with Altera's Quartus CAD flow [16], and a VPR [27] CAD flow. To attach to each of these CAD flows, first, Odin converts Verilog designs into structural Verilog netlists consisting of gate primitives and LPMs (or the equivalent of LPMs) targeted for a particular flow. In the case of the VPR CAD flow, the output netlist is in BLIF format [28] consisting of only primitive logic gates and flip-flops. These outputs are passed to the industrial or academic flow that follows with downstream synthesis, placement, and routing to generate area and timing results. One of the major benefits of targeting industrial CAD flows such as Quartus is that it allows us to obtain real speed and area results for modern FPGAs.

To verify that Odin generates working designs, we have built test benches for three of our benchmarks. These test benches include both the original Verilog design and a Verilog gate/LPM-level netlist mapped by Odin. The test bench generates inputs to both designs from one common source, and the outputs are joined through XOR gates so that during simulation if any of the outputs generate a "1" it means that the two designs are not generating the same values, and there is an error in the design mapped by Odin. Odin has been verified by simulating benchmarks cf_cordici_18_18, md, and fir_24_16_16 in ModelSim with some random vectors.

6. RESULTS

One of our key goals is to build a front-end synthesis tool that generates results comparable to industrial front-end synthesis tools that target FPGAs. In this section we describe our benchmarking methodology and benchmark circuits. Then we present and discuss the head-to-head comparison. Finally we show the specific value of each of the different mapping optimizations described in Section 7.1.

6.1. Benchmarking Methodology

For this comparison we use Quartus' CAD flow and map a set of benchmarks to Stratix FPGAs [1]. We use version 4.1 of Quartus and run our benchmarks through two CAD flows consisting of Quartus alone and Odin interfaced with Quartus.

Our benchmarks are Verilog HDL designs and have been gathered from various sources including: Opencores organization [29], SCU-RTL [30], Texas-97 [31], and the Benchmarks for Placement 2001 [32]. We have also converted applications developed at the University of Toronto from VHDL to Verilog. These designs include, Raytrace [33], Stereo Vision [34], and Molecular Dynamic system [35].

Table 1. Area results for a comparison between designs mapped by Odin joined to the Quartus CAD flow versus just the Quartus CAD flow mapping to Stratix FPGAs

	Number of Logic Elements			Number of 9x9 DSP blocks		
Designs	**A** - Quartus	**B** - Odin with Quartus	Ratio (B/A)	**C** - Quartus	**D** - Odin with Quartus	Ratio (D/C)
fft_258_6	2374	3190	1.34	28	32	1.14
iir1	289	501	1.73	7	7	1.00
iir	297	338	1.14	12	10	0.83
fir_3_8_8	84	84	1.00	4	4	1.00
fir_24_16_16	1598	1591	1.00	48	48	1.00
fir_scu_rtl	998	548	0.55	0	17	0.00
diffeq_f_systemC	221	271	1.23	24	40	1.67
diffeq_paj_convert	512	369	0.72	24	40	1.67
sv_chip1	17765	17145	0.97	80	96	1.20
sv_chip2	35554	36194	1.02	176	144	0.82
sv_chip2_no_mem	34379	33803	0.98	176	144	0.82
rt_raygentop	2622	2679	1.02	27	27	1.00
rt_raygentop_no_mem	2118	2815	1.33	27	27	1.00
rt_top	25056	28653	1.14	112	112	1.00
rt_top_no_mem	21557	29507	1.37	112	112	1.00
oc45_cpu	2191	3101	1.42	2	2	1.00
reed_sol_decoder1	1151	1183	1.03	13	13	1.00
reed_sol_decoder2	1799	1957	1.09	9	9	1.00
md	10542.6	14867	1.41	112	112	1.00
cordic_8_8	591	838	1.42	0	0	
cordic_18_18	2830	4104	1.45	0	0	
MAC1	2864	2812	0.98	0	0	
MAC2	9828	9720	0.99	0	0	
CRC33_D264	102	102	1.00	0	0	
des_area	1481	1305	0.88	0	0	
des_perf	4592	3838	0.84	0	0	
sv_chip0	12433	12729	1.02	0	0	
sv_chip0_no_mem	7281	7122	0.98	0	0	
sv_chip3_no_mem	170	134	0.79	0	0	
rt_frambuf_top	546	784	1.44	0	0	
rt_frambuf_top_no_mem	766	909	1.19	0	0	
rt_boundtop	1519	3895	2.56	0	0	
		Average	1.11			

7. COMPARISON BETWEEN ODIN AND QUARTUS' FRONT-END SYNTHESIS TOOL

Table 1 and Table 2 show the comparison of speed and area of both CAD flows for each benchmark. To reduce the experimental "noise" associated with placement and routing, the results are averaged over 5 random seeds for the placement. In Table 1, columns 2 and 3 show a comparison between the number of LEs used on a Stratix FPGA mapped by Quartus and mapped by Odin interfaced with Quartus. Column 4 is a ratio that indicates how Odin performs compared to Quartus. For any comparison ratio, when it is less than one means that Odin is performing better than Quartus. Similarly, in Table 1 column 5, 6, and 7 show the number of 9-bit by 9-bit Digital Signal Processing (DSP) blocks and a comparison ratio. Table 2 shows a speed comparison where column 2 and 3 show the maximum frequency of each circuit mapped by a pure Quartus and the Odin+Quartus flow. Column 4 shows our comparison ratio.

Overall, these results show that Odin generates comparable results to Quartus' front-end as it is only slightly worse in all cases. For the Stratix FPGA the geometrically averaged area comparison ratio is 1.11 and the geometrically averaged speed ratio is 1.05. These ratios indicate that Odin is generating only slightly poorer mapped designs compared to Quartus, but for many of the benchmarks we have quite comparable results. One of the main reasons our tool generates results that are close to Quartus' front-end tool is because we have built Odin to deal with mapping functionality to both soft fabric and tile-based heterogeneity, and we have added the transformations described previously.

7.1. Value of Specific Mapping Techniques in Odin

To see the affect the transformations have on the quality of results generated by Odin we run an experiment in which we turn on and off different mapping techniques. The mapping techniques for this experiment are: (1) Partial Mapping to the DSP block on the Stratix, (2) Arithmetic Optimizations, (3) State Machine Identification and one-hot recoding, and

Table 2. Speed results for a comparison between designs mapped by Odin joined to the Quartus CAD flow versus just the Quartus CAD flow mapping to Stratix FPGAs

	Speed in MHz		
Designs	**E** - Quartus	**F** - Odin with Quartus	Ratio (E/F)
fft_258_6	101.762	146.16	0.70
iir1	84.994	82.53	1.03
iir	115.504	109.158	1.06
fir_3_8_8	251.484	251.928	1.00
fir_24_16_16	83.624	75.042	1.11
fir_scu_rtl	149.75	109.54	1.37
diffeq_f_systemC	45.076	41.11	1.10
diffeq_paj_convert	37.672	29.858	1.26
sv_chip1	116.416	122.31	0.95
sv_chip2	48.17	49.74	0.97
sv_chip2_no_mem	52.99	56.564	0.94
rt_raygentop	134.86	127.24	1.06
rt_raygentop_no_mem	134.33	136.916	0.98
rt_top	45.15	52.05	0.87
rt_top_no_mem	47.778	53.596	0.89
oc45_cpu	86.43	61.63	1.40
reed_sol_decoder1	86.07	82.69	1.04
reed_sol_decoder2	68.26	53.58	1.27
md	41.984	34.942	1.20
cordic_8_8	212.12	256.436	0.83
cordic_18_18	166.946	222.72	0.75
MAC1	107.044	98.532	1.09
MAC2	82.92	74.876	1.11
CRC33_D264	0	0	0.00
des_area	235.312	194.372	1.21
des_perf	199.564	199.44	1.00
sv_chip0	Won't fit	120.16	NA
sv_chip0_no_mem	162.808	146.202	1.11
sv_chip3_no_mem	321	328.94	0.98
rt_frambuf_top	120.15	127.75	0.94
rt_frambuf_top_no_mem	124.804	139.3	0.90
rt_boundtop	187.2	83.35	2.25
		Average	1.05

(4) Multiplexer collapsing.

We ran Odin on all the benchmarks for five different configurations and pass the mapped designs into the Quartus CAD flow to get speed and area results. Each configuration consists of different mapping techniques turned on or off. We use five configurations for this experiment where each new configuration includes the mapping techniques of the previous configuration. The first configuration has all mapping techniques turned off. The second configuration only has mapping technique 1 turned on. The third technique has both mapping technique 1 and 2 turned on, and so on for the remaining three configurations where the fifth configuration has all techniques turned on.

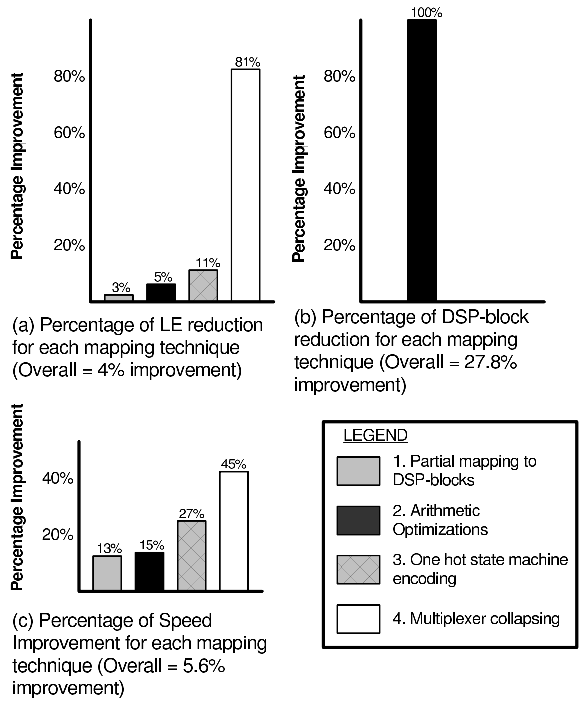
In Figure 5(a), the bar graph shows how each mapping technique contributes to decreasing the number of LEs used to map the benchmarks to a Stratix FPGA. All the mappings together provide a 4% decrease in the number of LEs used compared against the first configuration. We can see that *Multiplexer collapsing* contributes the majority of LE savings at 81% of the 4% improvement.

Figure 5(b) shows the improvement of the number of mapped DSP blocks used. This metric is only affected by *Arithmetic optimizations*, where shrinking the size of multiplication decreases the number of DSP blocks by a total of 27.8%. This emphasizes how important it is to propagate constants at front-end synthesis to ensure multipliers are mapped to the smallest implementation possible.

Figure 5(c) shows how the mapping techniques in Odin affect the speed of the benchmarks. The total speed improvement due to these techniques is 5.6%. Multiplexer collapsing results in the greatest improvement in speed at 2.5%.

8. CONCLUSIONS

In this paper we have presented a front-end Verilog RTL synthesis tool called Odin with available source code. Odin employs mapping techniques to generate designs that are comparable in area and speed to Quartus' front-end tool. We have also shown how certain mapping techniques improve



(a) Percentage of LE reduction for each mapping technique (Overall = 4% improvement)

(b) Percentage of DSP-block reduction for each mapping technique (Overall = 27.8% improvement)

(c) Percentage of Speed Improvement for each mapping technique (Overall = 5.6% improvement)

Fig. 5. Shows the results for turning each of 5 configurations on and how each technique affects the final results.

the results generated by Odin. These techniques, though simple, are important in mapping designs automatically and efficiently to modern FPGAs. Finally, we provide this software to the academic community hoping that the availability of this tool will allow researchers to pursue other avenues in front-end synthesis for FPGAs (*www.eecg.toronto.edu/∼jayar/odin/*).

This research is funded by National Science and Engineering Research Council of Canada under a post-graduate scholarship and a Discovery Grant.

9. REFERENCES

[1] *Stratix Device Handbook*, Altera, Jul 2003.

[2] *Virtex-II Pro Platform FPGAs*, Xilinx, Oct 2003.

[3] *Eclipse Family Data Sheet*, QuickLogic, 2003.

[4] R. J. Francis, J. Rose, and Z. Vranesic, “Chortle-crf: Fast technology mapping for lookup table-based fpgas,” in *Proc. 28th ACM/IEEE DAC*, 1991, pp. 613–619.

[5] K. Chen, J. Cong, Y. Ding, A. Kahng, and P. Trajmar, “Dag-map: Graph-based fpga technology mapping for delay optimization,” pp. 7–20, Sept. 1992.

[6] J. Cong, J. Peck, and Y. Ding, “RASP: A general logic synthesis system for SRAM-based FPGAs,” in *FPGA*, 1996, pp. 137–143.

[7] J. Bhasker and H.-C. Lee, “An optimizaer for hardware synthesis,” *IEEE Design & Test*, vol. 7, no. 5, pp. 20–36, Oct. 1990.

[8] R. E. Bryant, “Graph-based algorithms for boolean function manipulation,” *IEEE Transactions on Computers*, vol. C-35, no. 8, pp. 677–691, Aug. 1986.

[9] *The Programmable Gate Array Data Book*, Xilinx, 1989.

[10] *Plessey Semiconductor ERA60100 Advance Information, Data Sheet*, Plessey, 1990.

[11] D. Marple and L. Cooke, “An MPGA compatible FPGA architecture,” in *ACM/SIGDA Workshop on FPGAs*, 1992.

[12] *ACT 1 Series FPGAs*, Actel, 1996.

[13] J. Rose, “Hard vs. soft: The central question of pre-fabricated silicon,” in *34th International Symposium on Multiple-Valued Logic (ISMVL’04)*, Toronto, ON, May 2004, pp. 2–5.

[14] J. S. Rose, R. J. Francis, P. Chow, and D. Lewis, “The Effect of Logic Block Complexity on Area of Programmable Gate Arrays,” in *IEEE CIC*, May. 1989, pp. 5.3.1 – 5.3.5.

[15] D. Gregory, K. Bartlett, A. de Geus, and G. Hatchel, “Socrates: a system for automatically synthesizing and optimizaing combinational logic,” in *Proceedings 23rd DAC*, pp. 79–85.

[16] Altera, *Quartus II Handbook, Volumes 1, 2, and 3*, 2004.

[17] *Xilinx ISE 6 Software Manuals and Help*, Xilinx, 2004.

[18] Synplicity, “Synplify pro,” 2003.

[19] Magma Design Automation Inc., “Blast fpga,” 2005.

[20] Mentor Graphics, “Leanardospectrum,” 2001.

[21] Synopsys, “Design compiler fpga,” 2004.

[22] “ICARUS Verilog at www.icarus.com/eda/verilog/.”

[23] R. Kastner, S. Ogrenci-Memik, E. Bozorgzadeh, and M. Sarrafzadeh, “Instruction generation for hybrid reconfigurable systems,” in *ICCAD*, San Jose, CA, 2001, pp. 127–131.

[24] J. Cong, Y. Fan, G. Han, and Z. Zhang, “Application-specific instruction generation for configurable processor architectures,” in *FPGA ’04*, 2004, pp. 183–189.

[25] M. Ohlrich, C. Ebeling, E. Ginting, and L. Sather, “Subgemini: Identifying subcircuits using a fast subgraph isomorphism algorithm,” in *DAC*, 1993, pp. 31–37.

[26] S. Golson, “One-hot state machine design for FPGAs,” in *3rd PLD Design Conference*, Santa Clara, CA, 1993, pp. 1–6.

[27] V. Betz and J. Rose, “Directional Bias and Non-Uniformity in FPGA Global Routing Architectures,” in *14th IEEE/ACM Int’l Conference on CAD*, 1996, pp. 652–659.

[28] U. of California Berkeley, “Berkeley logic interchange format (BLIF),” 1992.

[29] “www.opencores.org.”

[30] “www.engr.scu.edu/mourad/benchmark/RTL-Bench.html.”

[31] “www-cad.eecs.berkeley.edu/Respep/Research/vis/texas-97/.”

[32] “www.cs.nthu.edu.tw/∼ylin/.”

[33] J. Fender and J. Rose, “A high-speed ray tracing engine built on a field-programmable system,” in *IEEE Internation Conf. On Field-Programmable Technology*, 2003, pp. 188–195.

[34] A. Dharabiha, J. Rose, and W. MacLean, “Video-rate stereo depth measurement on programmable hardware,” in *IEEE Computer Society Conference on Computer Vision & Pattern Recognition*, 2003, pp. 203–210.

[35] N. Azizi, I. Kuon, A. Egier, A. Darabiha, and P. Chow, “Reconfigurable molecular dynamics simulator,” in **Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines**, April 2004.