EVALUATING THE EFFICIENCY OF DSP BLOCK SYNTHESIS INFERENCE FROM FLOW GRAPHS

Bajaj Ronak, Suhaib A. Fahmy

School of Computer Engineering
Nanyang Technological University
Nanyang Avenue, Singapore
email: {ronak1,sfahmy}@ntu.edu.sg

ABSTRACT

The embedded DSP Blocks in FPGAs have become significantly more capable in recent generations of devices. While vendor synthesis tools can infer the use of these resources, the efficiency of this inference is not guaranteed. Specific language structures are suggested for implementing standard applications but others that do not fit these standard designs can suffer from inefficient synthesis inference. In this paper, we demonstrate this effect by synthesising a number of arithmetic circuits, showing that standard code results in a significant resource and timing overhead compared to considered use of DSP Blocks and their plethora of configuration options through custom instantiation.

1. INTRODUCTION

Field Programmable Gate Arrays (FPGAs) have evolved significantly and found use within new market segments during the past two decades. An increase in the types and capabilities of heterogeneous resources means FPGAs are now well-equipped for use across a much wider range of domains. However, this increases design automation complexity. Tools must now evaluate the possible options for mapping specific functions to logic or hard macro blocks. Standard signal processing circuits, such as finite impulse response (FIR) filters are well understood by the tools and can be mapped efficiently to the embedded DSP Blocks. The difficulty arises when other algorithms that can benefit from hard blocks are synthesised: often, the tools fail to find the most efficient mapping. This is especially true where the hard blocks can be configured in multiple ways, as with the DSP Blocks on modern devices.

For computation-dominated applications, one fundamental step to bridging the gap between FPGAs and ASICs is to ensure that arithmetic is performed as efficiently as possible and this is where hard blocks excel. In recent Xilinx devices, the DSP Blocks can be dynamically programmed to execute many functions. Fig. 1 shows a simplified representation of the DSP48E1 primitive.

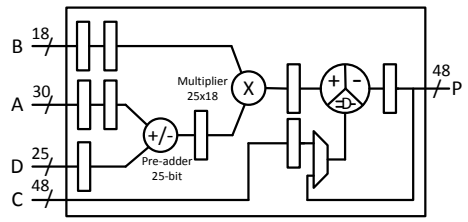


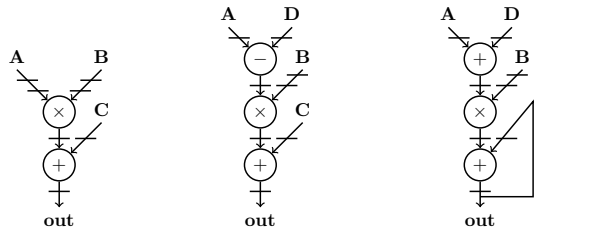
Fig. 1: Basic structure of DSP48E1 primitive.

In FPGA implementations of many DSP-amenable applications, we observed that synthesising standard RTL code and relying on synthesis tools to correctly infer the use of DSP Blocks results in sub-optimal results with significant extra logic, even in cases where none is needed. In this paper, we set out to explore this phenomenon through a comparison between automated synthesis and manual mapping.

We synthesise a set of circuits that should map well to DSP Blocks, and compare the resource usage of two standard RTL implementations of each circuit with a manual implementation where DSP Blocks are manually instantiated, from analysis of their dataflow representations. We find that automated synthesis does indeed result in a significant overhead in terms of resource utilisation and clock frequency.

2. RELATED WORK

In [1], algorithms are represented using synchronous dataflow graphs and techniques are proposed for mapping them to pipelined datapaths on FPGAs. As multiplexers with a large number of inputs are costly on FPGAs, the main focus is to minimise resource usage and latency by minimising the number of multiplexers. [2] extends the Grape-II [3] tool for automatic code generation for task communication and scheduling on FPGAs. [4] proposes techniques to minimise interconnect area by efficient functional unit, register, and interconnection allocation, based on the dependence of operations in the dataflow graphs. A tool is proposed in [5] to convert dataflow graphs into synthesisable VHDL descrip-



(a) $C+(A\times B)$ (b) $C+(D-A)\times B$ (c) $out = out+(A+D)\times B$

Fig. 2: Dataflow graphs of templates

tions, but, it does not deal with hard blocks in the FPGA.

Odin-II [6], is an open-source Verilog synthesis tool. While it can map to embedded multipliers, it does not deal with the more complex DSP Blocks we find today. Previous work [7] has shown how programmability of DSP Blocks can be leveraged to build a general-purpose processor.

We have not found any work that focusses on mapping dataflow graphs directly to modern DSP Blocks. Papers focusing on DSP architectures mainly optimise multiply and multiply-accumulate operations, but DSP algorithms can be implemented more efficiently with considered use of the DSP Blocks available on FPGAs. This has become more important as DSP Blocks are now more complex and dynamically reprogrammable.

It is also important to note where inference can occur within the implementation chain. While the technology mapper can process the synthesis output netlist, there may be sufficient information in an RTL description to make this decision up front. This may impact the efficiency of inference. What compounds this problem is that the mapping is not to a single fixed structure, but rather to a range of possible DSP Block configurations.

3. COMPARISON APPROACH

In this paper, we investigate the disparity between automatic inference of DSP Blocks from dataflow graphs, and direct instantiation of various DSP configuration templates. Conventionally, algorithms are implemented in hardware description languages (HDLs) like Verilog and VHDL. During technology mapping, the implementation tools determine how individual portions of logic should be mapped to the resources available on the target device. This is where heterogeneous resources are usually inferred. Vendors provide specific coding styles that help coax the tools to infer efficiently. While this works for generalised structures like finite impulse response (FIR) filters, other non-regular, non-standard structures can suffer from sub-optimal inference.

We can see in Fig. 1 that there are three main stages in the DSP48E1 primitive: a pre-adder, a multiplier, and an ALU stage. Specific configuration inputs control the types

Table 1: Results

Method	DSPs	LUTs	Regs	Max Freq (MHz)
Comb	1	235	99	518
Pipe	1	0	0	560
Direct	1	0	0	560

```
always @ (posedge CLK)
begin
  pipe_reg_A1 <= A;     pipe_reg_B1 <= B;
  pipe_reg_D  <= D;     pipe_reg_B2 <= pipe_reg_B1;
  pipe_reg_AD <= pipe_reg_D + pipe_reg_A1[24:0];
  pipe_reg_C  <= C;
  pipe_reg_M  <= pipe_reg_AD * pipe_reg_B2;
  pipe_reg_P  <= pipe_reg_M + pipe_reg_C;
end
assign P = pipe_reg_P;
```

Fig. 3: Verilog code of pipelined implementation

of operations and can be set at run time, or fixed when instantiating the DSP48E1 primitive, by hard-wiring those ports. After analysing different configuration options, we prepared a database of 29 different DSP Block configurations. Dataflow graphs of three configurations are shown in Fig. 2. Non-arithmetic functions are not considered in this paper.

If standard arithmetic is coded at RTL level, the implementation tools decide how to use the DSP Blocks, and hard-wire the configuration accordingly. As a first step, we consider an example, of a circuit mirroring the function of a DSP Block $((A+D)\times B)+C$. This expression can be coded in RTL as combinational logic, in a pipelined manner mirroring the structure of a DSP Block, or by direct instantiation of a DSP Block in RTL code with correct configuration options. Code for the pipelined version of the expression mentioned above is shown in Fig. 3. When we process this simple circuit through the implementation tools, we obtain the results shown in Table 1. Extra pipeline stages are added at the output of the combinational implementation to equalise the number of pipeline stages of all three implementation. We observe that for the combinational implementation, the tools use extra resources, which can be otherwise implemented with only one DSP Block. The tools were able to infer the DSP Block, without extra resources from the pipelined RTL code. Hence, it seems there is some intelligence in the tools, but we are interested in seeing how this changes with more complex circuits.

When similar implementations are done for larger circuits, we observe that the tools do not infer DSP Blocks efficiently. In our experiments, we implement a variety of arithmetic circuits using three different methods as we did for the single block. In the first method (Comb), logic is implemented combinationally, with register stages added at the end to match the delay through a DSP Block allowing the tools to re-time the design.

In the second method (Pipe), code is written in syn-

chronous RTL to match the structure of the DSP Blocks, with the expectation that the tools will infer their use. For subsequent DSP stages, we truncate the values to fit. We assume a fixed-point representation with a large number of fractional bits (as is typical in DSP systems), resulting in a loss of precision, but maintaining scale. This truncation is done in all designs.

The third method (Direct) is by direct manual instantiation of templates from the database previously mentioned. Although we use the DSP Blocks to implement as much logic as possible, there is still a need to add external registers to ensure inputs to the DSP blocks are correctly aligned. Furthermore, we must balance the pipeline branches across the graph so that separate DSP Blocks produce results at the correct time for subsequent stages. Segmentation of the graphs is done so that the number of segments is minimised by using as many features of the DSP Block as possible. The circuit is then implemented by direct instantiation of templates. All three implementations are written to have the same number of pipeline stages.

DSP Blocks allow cascading using dedicated wires. The P output of one block can be connected to C input of the adjacent block without using routing resources. However, in many of our designs, the P output must connect to the A or B inputs, for which no dedicated wiring is provided. Hence, to mitigate significant routing delay, we add a pipeline register at the output of each DSP Block, so that the routing delay between two connecting DSP Blocks can be eliminated. This change can result in over a 40% speed improvement.

4. RESULTS

We implemented a variety of algorithms for this investigation. Algebraic expressions of the Savitzky-Golay filter, Mibench2 filter, Quadratic Spline, and Chebyshev Polynomial from [8] were implemented from their dataflow graphs. An example segmentation of the Mibench2 flow graph is shown in Fig. 4. The Polynomial Test Suite in [9] contains a large number of multivariate polynomials. We implemented four of these. Resource usage and timing results for the three methods using Xilinx ISE 13.2 are shown in Table 2 and Fig. 5.

We observe some interesting results. Firstly, the attainable frequency is always highest using direct instantiation. Since all logic is moved into the DSP Block, except two-input adders with no other associated logic, and since we fully pipeline the DSP Block, we are able to achieve the maximum frequency for these designs.

We see that the Comb implementation has the worst timing. Even with register balancing (retiming) enabled and ample register stages added, the tools do not seem to be able to use more than one internal pipeline stage in the DSP Block. Furthermore, the tools only use the multiply operation when synthesising the Comb representation. In some designs, Direct and Pipe implementations use more registers and LUTs. The LUTs are used as route-thrus with same-slice registers, and as shift registers in delay lines.

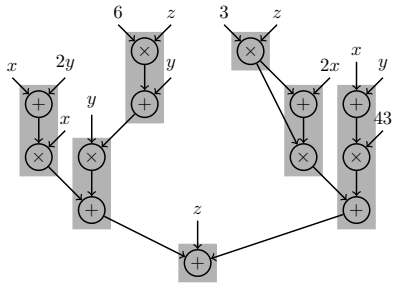


Fig. 4: Mibench2 filter dataflow graph with segmentation

Table 2: Results

	Method	DSPs	LUTs	Regs	Max Freq (MHz)
SG Filter	Comb	7	184	147	57
	Pipe	5	407	729	203
	Direct	6	126	342	486
Mibench2	Comb	7	214	202	88
	Pipe	6	238	600	314
	Direct	7	175	493	487
Q Spline	Comb	12	309	168	54
	Pipe	13	278	615	428
	Direct	14	317	694	514
Chebyshev	Comb	3	181	140	73
	Pipe	3	129	347	253
	Direct	3	40	142	515
Poly1	Comb	4	291	177	106
	Pipe	4	151	206	504
	Direct	4	198	230	560
Poly2	Comb	4	295	174	89
	Pipe	4	259	515	256
	Direct	5	100	180	548
Poly3	Comb	6	184	156	74
	Pipe	6	270	597	212
	Direct	6	152	376	506
Poly4	Comb	3	89	120	126
	Pipe	3	209	532	211
	Direct	3	65	262	507

We can see that the Pipe implementation generally infers DSP Blocks well compared to Comb, given that the code closely mirrors the structure of DSP Block. The tools were also able to infer multiple configurations of the DSP Block. But, sometimes the tools leave some functions in logic that could use a DSP Block, such as a wide three-input add. Though there is no multiply involved, this function can be moved into a DSP Block, and hence save logic area, and optimise timing.

We found the frequency of Pipe to be significantly less than Direct, except in two algorithms (Quadratic Spline and

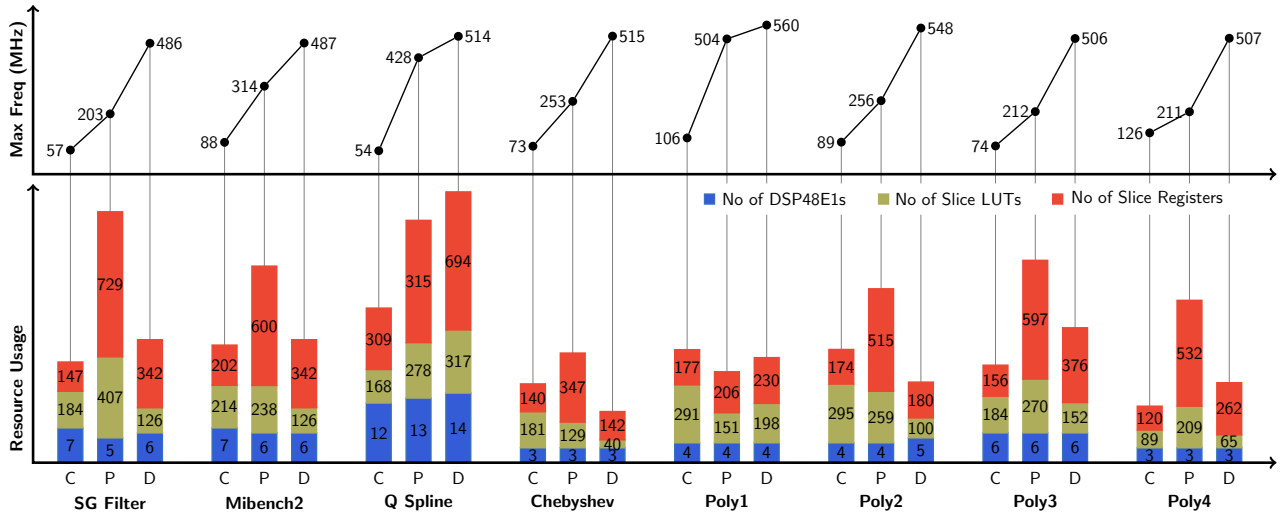


Fig. 5: Resource usage and maximum frequency results.

Poly1). This can be attributed to sub-optimal mapping and inefficient use of the pipeline stages inside the DSP Blocks. The dataflow graphs of Quadratic Spline and Poly1 are symmetric and balanced, and thus the tools are able to map them effectively. The number of LUTs used as route-thrus is also less for these two circuits.

If we compare the LUT usage between the three methods, Direct results in 69% less to 31% more usage compared to Pipe; and 78% less to 2.6% more usage compared to Comb. In terms of maximum frequency, algorithms implemented directly run 1.11 to 2.4 times faster compared to the Pipe method; and 4.01 to 9.48 times faster compared to Comb method. In conclusion, it remains the case that for optimal speed, and efficient area usage, direct instantiation should be used when mapping complex arithmetic expressions to DSP Blocks.

5. CONCLUSION AND FUTURE WORK

In this paper, we analysed the efficiency of DSP Block inference in the synthesis of arithmetic flow graphs. We compared a combinational implementation, a DSP-Block-centric pipelined implementation, and direct instantiation using various DSP Block configurations. We implemented 8 different circuits and showed that the direct instantiation generally results in less resource usage and higher operating frequency compared to conventional approaches.

For future work, we intend to explore how we can automate this mapping from RTL code and high-level descriptions. Since the DSP Blocks are dynamically configurable, it would also be possible to synthesise resource-shared implementations that involve reconfiguring the DSP Block. We intend to investigate this as a way of mapping within resource constraints.

6. REFERENCES

[1] O. Maslennikow and A. Sergiyenko, "Mapping DSP Algorithms into FPGA," in *Int. Symp. on Parallel Computing in Electrical Engineering*, Sept 2006, pp. 208–213.

[2] J. Dalcolmo, R. Lauwereins, and M. Ade, "Code generation of data dominated DSP applications for FPGA targets ," in *Proc. Int. Workshop on Rapid System Prototyping*, Jun 1998, pp. 162–167.

[3] R. Lauwereins, M. Engels, M. Ade, and J. Peperstraete, "Grape-II: a system-level prototyping environment for DSP applications," *Computer*, vol. 28, no. 2, pp. 35–43, Feb 1995.

[4] Y. Hirakawa, M. Yoshida, K. Harashima, and K. Fukunaga, "A method of data path allocation by pattern matching on the data flow graph," in *IEEE Signal Processing Society Workshop on VLSI Signal Processing*, Oct 1995, pp. 254–263.

[5] P. Necsulescu and V. Groza, "Automatic generation of VHDL hardware code from data flow graphs," in *IEEE Int. Symp. on Applied Computational Intelligence and Informatics (SACI)*, May 2011, pp. 523–528.

[6] P. Jamieson, K. Kent, F. Gharibian, and L. Shannon, "Odin II - An Open-Source Verilog HDL Synthesis Tool for CAD Research," in *IEEE Int. Symp. on Field-Programmable Custom Computing Machines (FCCM)*, May 2010, pp. 149–156.

[7] H. Y. Cheah, S. A. Fahmy, D. L. Maskell, and K. Chidamber, "A Lean FPGA Soft Processor Built Using a DSP Block," in *ACM/SIGDA Int. Symp. on Field Programmable Gate Arrays*, Feb 2012, pp. 237–240.

[8] S. Gopalakrishnan, P. Kalla, M. Meredith, and F. Enescu, "Finding linear building-blocks for RTL synthesis of polynomial datapaths with fixed-size bit-vectors," in *IEEE/ACM Int. Conf. on Computer-Aided Design*, Nov 2007, pp. 143–148.

[9] "Polynomial Test Suite." [Online]. Available: http://www-sop.inria.fr/saga/POL/