Improved Resource Sharing for FPGA DSP Blocks

Bajaj Ronak
School of Computer Science and Engineering
Nanyang Technological University, Singapore
Email: ronak1@ntu.edu.sg

Suhaib A. Fahmy
School of Engineering
University of Warwick, Coventry, UK
Email: s.fahmy@warwick.ac.uk

***Abstract*—Sharing multi-cycle hardware blocks like the DSP48E1 primitive in Xilinx FPGAs can result in significant resource savings, but complicates scheduling. For high-throughput, DSP blocks must be pipelined, which results in a high initiation interval (II) for resource shared implementations. In this paper, we propose a resource reduction technique that minimises DSP block usage while also offering improved II over traditional approaches. This is integrated in a high-level tool which takes datapath descriptions in C and generates synthesisable Verilog RTL with different levels of resource sharing. We demonstrate significantly improved throughput compared to traditional resource sharing while achieving resource reduction compared to resource unconstrained and HLS implementations. The approach explores an otherwise infeasible design space between resource unconstrained and traditional resource sharing methods.**

I. Introduction

Field Programmable Gate Arrays (FPGAs) have evolved significantly over the last two decades from implementing glue logic to platforms for implementing complex architectures. Embedded hard blocks on modern devices implement often-used functions directly in silicon, thus consuming less area and power, and running at a higher clock speed than the equivalent function in logic. These include memory blocks, DSP blocks, embedded processors, and more. The DSP48E1 primitive in modern Xilinx FPGAs consists of multiple sub-blocks which can be combined in different ways to perform up to three different operations per DSP block. They can be pipelined with up to four stages allowing maximum throughput when all three sub-blocks are used. Another key feature is their dynamic programmability, allowing the functionality of the DSP block to be modified at runtime in each clock cycle. This greatly enhances the capabilities of a DSP48E1 primitive, as it can be reprogrammed and multiple different operations can be mapped to a single DSP block. This flexibility has been exploited to create lean soft processors [1] and to support high performance programmable overlays [2].

Hard blocks are typically a constrained resource, and hence resource sharing should be applied where possible to free up more for other uses. Traditionally, operations scheduled in non-overlapping time schedules can be mapped to same hardware resource. A major disadvantage of traditional resource sharing is that it generally increases schedule length and results in high initiation interval (II). DSP blocks require internal pipeline stages to be enabled to achieve high frequencies which significantly impact II when shared.

In this paper, we present a scheduling and implementation technique for resource sharing of DSP blocks that overcomes the II limitations of traditional approaches. We exploit the dynamic programmability to allow different computational patterns to share the same DSP block. Instead of reconfiguring a set of DSP blocks to implement all operations, we use multiple sets of DSP blocks controlled using different state machines ensuring that each set achieves the target II. In traditional resource sharing, the structure of the dataflow graph limits the achievable II. As a result, the choice is often between an implementation with no resource sharing, or a resource shared implementation with very low throughput. We discuss this limitation in detail in Section III-B.

We have integrated the proposed techniques into a high-level tool, which generates synthesisable RTL from C descriptions of complex mathematical expressions. The tool can generate a wide range of implementations, exploring the trade-off between highly pipelined, fully parallel and resource shared implementations with improved throughput, allowing better balancing of LUT usage in large designs.

II. Related Work

A significant amount of research has been done on resource sharing at the RTL level as well as in high-level synthesis. An algorithm was proposed in [3] combining temporal partitioning, resource sharing, scheduling, allocation, and binding to obtain resource efficient implementations. Five heuristics for global resource sharing were proposed in [4] which focuses on inter-basic-block sharing in addition to resource sharing for each basic block. The approach in [5] combined module selection and resource sharing to minimise area while achieving throughput requirements.

Generally, HLS tools use static scheduling to determine the extent of resource sharing possible. Work in [6] proposed source-to-source transformations to improve efficiency and II using dynamic scheduling techniques. Multi-pumping is another approach for reducing utilisation of hard blocks like DSPs [7], [8], by running them at double the frequency of the surrounding circuit, computing two DSP block operations in one system clock cycle.

Various heuristics have been proposed including list scheduling, force-directed scheduling (FDS) [9], and a recent scheduler based on a system of difference constraints (SDC) was proposed in [10]. We are not aware of any work that focuses on multi-cycle flexible hard blocks like the DSP48E1.

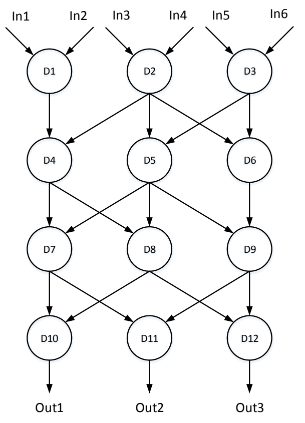

Fig. 1: Dataflow graph of case study example.

These present unique challenges due to their ability to be shared by different computations, and the complex latency constraints enforced by their pipeline configuration. As a result, when previous techniques are applied, the resource sharing is sub-optimal.

III. Scheduling

In this section, we explore traditional resource sharing (TRS) techniques and the proposed improved resource sharing (IRS) approach. Both are built on top of the system of difference constraints (SDC) scheduling proposed in [10] for generating optimised schedules. SDC scheduling formulates the scheduling problem mathematically as a set of linear constraints that can then be solved using a linear programming (LP) solver.

A. Traditional Resource Sharing (TRS)

To apply traditional resource sharing, the scheduling algorithm takes a dataflow graph and a constraint on the number of DSP blocks. The algorithm then uses as many DSP blocks as possible to schedule the computation in as short a latency as possible. SDC formulation typically generates a final schedule in four steps. The first step is to initialise the LP problem with scheduling variables for each node. The second steps adds constrains to guide the final schedule. For TRS, constraints for multi-cycle hardware blocks, dependencies, and resources available are added. The next step formulates the objective function for which the LP is solved. The final step determines the schedule times for each node from the formulated LP problem solution satisfying all the constraints.

B. Example for Traditional Resource Sharing

We illustrate this approach using a simple example as a case study. The synthetic DFG for the case study is shown in Fig. 1. Each node in the dataflow graph represents a configuration of a DSP48E1 primitive.

For the DDFG in Fig. 1, the maximum number of DSP blocks in a schedule time is three due to data dependencies. DSP block constraints higher than three would not improve II any further. For DSP block constraints of 1, 2, and 3, different implementation achieving different IIs can be generated. Here, we define *control step* as the number of clock cycles required to complete the computation for one set of configurations of the DSP blocks, which is five in this case. The schedule length and II are calculated as $((\#ControlSteps \times DSP_{depth}) + 2)$ and $((\#ControlStep - 1) \times DSP_{depth} + 1)$ respectively. The schedule length and II for DSP block constraints of 1, 2, and 3 is shown in Table I. The best II achievable using traditional resource sharing is 16 clock cycles.

TABLE I: Schedule length and II achieved for different TRS constraints.

	#DSP=1	#DSP=2	#DSP = 3
Schedule length	62	32	22
II	56	26	16

TABLE II: Schedule length and number of DSP used for different IRS constraints.

	II=1	II=6	II=11	II=16
Schedule length	22	22	32	22
#DSPs	12	6	4	3

C. Improved Resource Sharing (IRS)

We have seen that the long datapath latency of DSP blocks coupled with the way TRS approaches schedule operations results in high II implementations. Since TRS uses a single bank of DSP blocks to implement the complete computation, the II is determined by the maximum number of configurations required per DSP block, i.e., the level of re-use. To improve on this, we adopt a new approach that optimises for II rather than the number of DSP blocks. We first generate schedules using TRS for all DSP block constraints from 1 to the maximum width of graph, most of which will not meet lower II constraints. For each of these schedules, we identify which shared operations can be split to achieve the II constraint. This results in an increase in DSP block usage but achieves the target II. Among all these possible schedules, we pick the schedule with minimum area-delay product, i.e., product of number of DSP blocks and schedule length.

D. Example for Improved Resource Sharing

Consider again the example DFG in Fig. 1. For a given constraint of II_{max}, with each control step of five cycles, the number of stages which can be implemented using a set of DSP blocks is determined as $(II_{max} + 4)/5$. Table II shows the schedule length and number of DSP blocks used for II constraints of 1, 6, 11, and 16.

We can see from Tables I and II, that we are able to improve on the best II offered by TRS (16) by increasing DSP block usage beyond 3, thereby offering more points in the design space.

IV. Automated Tool Flow

We adapt the tool flow in [11] to generate synthesisable RTL for TRS and IRS implementations. In addition to TRS and IRS, we also generate Vivado HLS implementations to understand how Xilinx's HLS tool performs with different II constraints. Vivado HLS uses *directives* to guide RTL generation.

The tool in [11] accepts a computational kernel description in C. LLVM passes are used to translate the input C file in to a set of DOT files, one for each function. These DOT files are then parsed to generate a dataflow graph, which is then partitioned into sub-graphs that can be mapped to DSP block configurations. The partitioned graph, called the DSP dataflow graph (DDFG), comprises nodes that are either DSP48E1 primitive configurations or adder/subtractors to be implemented in FPGA logic. These add/sub nodes are those that cannot be merged with multipliers in the original graph to map to DSP block configurations. The partitioning algorithm is discussed in detail in [11].

After generating the DDFG, we perform scheduling (according to TRS and IRS techniques discussed in Section III) and register balancing, and generate a Vivado HLS implementation. For fair comparison, we ensure the same wordlengths for all operations across TRS, IRS, and HLS implementations. The Vivado HLS directive for pipelining is used with the input II constraint.

The next stage involves generating RTL for all three techniques. For TRS and IRS, DSP blocks have a latency of five clock cycles. All four internal pipeline stages are enabled to achieve high throughput and one extra register is added at the output to hold the output value of the configuration when the DSP block is used for other computations. Resource sharing implementation can be divided into a data path and control path. The data path includes DSP blocks with multiplexers at inputs, add/sub blocks, and pipeline balancing registers. The design of the control path for TRS and IRS varies.

TRS: For a given constraint on the number of DSP blocks (n), a maximum of n DSP blocks are used in each schedule time. The control path includes a microcoded read-only memory (ROM) initialized with control signals depending on the schedule generated. The tool analyses the nodes scheduled in each schedule time, and correspondingly generates ROM contents for initialisation. The ROM output controls the input multiplexers and configurations for the DSP blocks in each clock cycle. The II for the TRS implementation is determined by the maximum number of configurations required for any DSP block.

IRS: IRS uses multiple sets of separately shared DSP blocks. Thus, the control path consists of multiple state machines, one for each set of DSP blocks. For each state machine, a microcoded ROM is initialized with the correct control signals. Due to the pipeline depth of five for DSP blocks, the supported constraints on II are increments of five, i.e. 1, 6, 11, 16, and so on. An II of 1 implies a fully pipelined resource-unconstrained implementation, where each DSP block is used for only one operation.

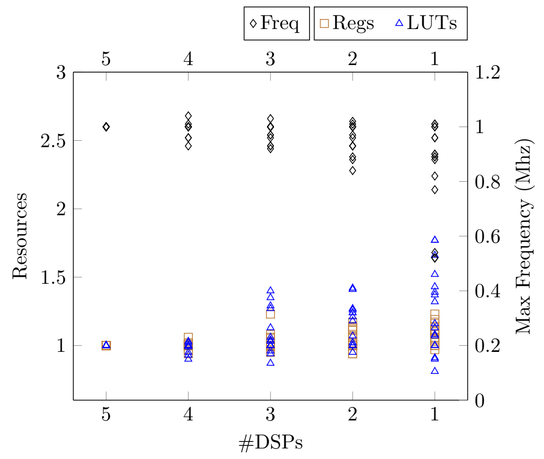

Fig. 2: Frequency/area tradeoff for constraints on number of DSPs varying from 5 to 1, normalized against 5 DSPs.

HLS: For HLS, we run the Vivado HLS project generated in the previous stage, which translates the high-level C++ implementation into synthesisable RTL.

After generating RTL for all the techniques, the vendor tool flow is executed to generate final results using a fully automated set of scripts.

V. Experiments and Analysis

All implementations target the Virtex 6 XC6VLX240T-1 FPGA as found on the ML605 development board, and use the Xilinx ISE 14.6 and Xilinx Vivado HLS 2013.4 tools.

To explore the effectiveness of the proposed technique, we implemented a number of benchmark multiply-add flow graphs. These include the Mibench2 filter, quadratic spline, and Savitzky-Golay filter from [12]; the ARF, EWF, Horner Bézier, motion vector, and smooth triangle extracted from MediaBench [13]; and 7 polynomials of varied complexity from the Polynomial Test Suite [14]. Note that these represent computational kernels within larger applications. As we are analysing the low-level effects of this resource sharing technique, we feel this is more appropriate than analysing large applications where the optimised logic is only a very small fraction.

We present the trade-off between post-place-and-route area (LUTs and registers) and throughput in Fig. 2. All values are normalized to the resources used and maximum frequency achieved for implementation with 5 DSPs since TRS does not utilise more than 5 DSP blocks for any of the benchmarks. We can see that with fewer DSP blocks, as schedule length increases resulting in more balancing registers being required, and as more operations are mapped onto the same DSP blocks, the complexity of the state machines increases, contributing to increased usage of LUTs and registers.

Fig. 3 shows the throughput gain as more DSP blocks are added for all benchmarks in our set. The increase is with reference to the number of DSP blocks used for the highest throughput TRS implementation. The traditional approach achieves a best II of 11 for more than half of the designs

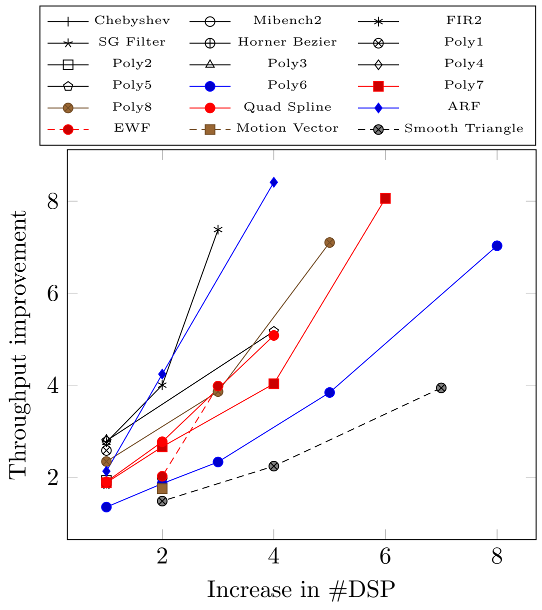

Fig. 3: Tradeoff between increase in DSP blocks usage and throughput improvement. Throughput values are normalised with maximum throughput achieved using TRS.

but cannot achieve 6 for any. For an II of 11, the improved approach offers an average throughput improvement of 1.8× (0.92×-4×) at a cost of 1.4× DSP blocks. For an II of 6, throughput improvements are up to 8× (Poly6) at a cost of a 3× increase in DSP blocks. Our proposed approach hence enables possible design points between resource unconstrained implementations and the best throughput achievable using the traditional approach (design points shown in Fig. 3), allowing designers more flexibility in the area-throughput trade-off. Within the context of a high-level synthesis tool, this means computational sections of code can be optimised to minimise resource usage given the throughput constraints imposed by the rest of the design, rather than over-using DSP blocks but clocking them at reduced rates.

Compared to resource unconstrained implementations (II=1), our approach achieves up to a 50% reduction in DSP blocks for an II of 6, and up to 67% for II of 11. Recall that these configurations are not achievable with TRS for many benchmarks. For fairness, both TRS and IRS implementations explored use the DSP block's dynamic programmability, such that different operations can be mapped to the DSP48E1 primitives in different cycles.

We used Vivado HLS with II constraints of 1, 6, and 11 and compared with IRS implementations. The Vivado HLS implementations do not exploit DSP block dynamic programmability which is crucial in our work. For an II of 1, equivalent to an unconstrained implementation, DSP block utilisation for HLS is similar to that of IRS. However, for higher II constraints, the tool is not able to optimise designs for relaxed IIs, resulting in designs without reduction in DSP block utilisation.

VI. Conclusions

Traditional resource sharing of DSP blocks results in high II due to their long pipeline depths. As a result, any sort of sharing results in very low throughputs that are insufficient for many applications. Meanwhile, resource unconstrained implementations often achieve higher frequencies and throughputs than needed, at a cost of excessive DSP block usage. In this paper, we have presented an SDC based scheduling technique that allows for lower IIs than are achievable using traditional approaches. This results in throughput improvements of up to 8× at a cost of up to 3× the number of DSP blocks, with resulting designs better balancing LUT and DSP block usage. The proposed technique has been integrated into an automated tool flow, which can generate synthesisable RTL from an input C description, allowing exploration of an otherwise infeasible space between unconstrained and traditional resource sharing methods. We also showed that Vivado HLS does not offer the same DSP block savings when opting for lower IIs.

References

[1] H. Y. Cheah, F. Brosser, S. A. Fahmy, and D. L. Maskell, "The iDEA DSP block-based soft processor for FPGAs," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 7, no. 3, pp. 19:1–19:23, 2014.

[2] A. K. Jain, D. L. Maskell, and S. A. Fahmy, "Throughput oriented FPGA overlays using DSP blocks," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, 2016, pp. 1628–1633.

[3] J. Cardoso, "Novel algorithm combining temporal partitioning and sharing of functional units," in *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines*, March 2001.

[4] S. O. Memik, G. Memik, R. Jafari, and E. Kursun, "Global resource sharing for synthesis of control data flow graphs on FPGAs," in *Proceedings of the Design Automation Conference*, June 2003.

[5] W. Sun, M. Wirthlin, and S. Neuendorffer, "FPGA pipeline synthesis design exploration using module selection and resource sharing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 254–265, 2007.

[6] M. Alle, A. Morvan, and S. Derrien, "Runtime dependency analysis for loop pipelining in high-level synthesis," in *Proceedings of the Design Automation Conference*, May 2013.

[7] A. Canis, J. H. Anderson, and S. D. Brown, "Multi-pumping for resource reduction in FPGA high-level synthesis," in *Proceedings of Design, Automation Test in Europe Conference and Exhibition (DATE)*, March 2013, pp. 194–197.

[8] B. Ronak and S. A. Fahmy, "Minimizing DSP block usage through multi-pumping," in *Proceedings of the International Conference on Field Programmable Technology (FPT)*, Dec 2015, pp. 184–187.

[9] P. G. Paulin and J. P. Knight, "Force-directed scheduling in automatic data path synthesis," in *Proceedings of the Design Automation Conference*, 1987, pp. 195–202.

[10] J. Cong and Z. Zhang, "An efficient and versatile scheduling algorithm based on SDC formulation," in *Proceedings of the Design Automation Conference*, 2006, pp. 433–438.

[11] B. Ronak and S. A. Fahmy, "Mapping for maximum performance on FPGA DSP blocks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 4, pp. 573–585, April 2016.

[12] 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 *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, Nov 2007, pp. 143–148.

[13] C. Lee, M. Potkonjak, and W. Mangione-Smith, "MediaBench: a tool for evaluating and synthesizing multimedia and communications systems," in *Proceedings of the International Symposium on Microarchitecture*, Dec 1997, pp. 330–335.

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