Impact of FPGA Architecture on Resource Sharing in High-Level Synthesis

Stefan Hadjis[1], Andrew Canis[1], Jason Anderson[1], Jongsok Choi[1], Kevin Nam[1], Stephen Brown[1], and Tomasz Czajkowski[‡]

[1]ECE Department, University of Toronto, Toronto, ON, Canada
[‡]Altera Toronto Technology Centre, Toronto, ON, Canada

ABSTRACT

Resource sharing is a key area-reduction approach in high-level synthesis (HLS) in which a single hardware functional unit is used to implement multiple operations in the high-level circuit specification. We show that the utility of sharing depends on the underlying FPGA logic element architecture and that different sharing trade-offs exist when 4-LUTs vs. 6-LUTs are used. We further show that certain multi-operator patterns occur multiple times in programs, creating additional opportunities for sharing larger composite functional units comprised of patterns of interconnected operators. A sharing cost/benefit analysis is used to inform decisions made in the binding phase of an HLS tool, whose RTL output is targeted to Altera commercial FPGA families: Stratix IV (dual-output 6-LUTs) and Cyclone II (4-LUTs).

Categories and Subject Descriptors

B.7 [**Integrated Circuits**]: Design Aids

Keywords

Field-programmable gate arrays, FPGAs, high-level synthesis, resource sharing

1. INTRODUCTION

High-level synthesis (HLS) refers to the automatic compilation of a program specified in a high-level language (such as `C`) into a hardware circuit. There are several traditional steps in HLS. *Allocation* determines the number and types of functional units to be used in the hardware implementation. This is followed by *scheduling*, which assigns operations in the program specification to specific clock cycles and generates a corresponding finite state machine (FSM). *Binding* then assigns the operations in the program to specific functional units in a manner consistent with the allocation and scheduling results.

A well-studied area-reduction optimization in the binding step is called *resource sharing*, which involves assigning multiple operations to the same hardware unit. Consider, for example, two additions that are scheduled to execute in different clock cycles. Such additions may be implemented by the same adder in hardware – the additions *share* the hardware adder. Resource sharing is accomplished by adding multiplexers (MUXes) to the inputs of the shared functional unit, with the FSM controlling the MUXes to steer the correct data to the adder based on the state. Since MUXes are costly to implement in FPGAs, resource sharing has generally been thought to have little value for FPGAs, except in cases where the resource being shared is large or is scarce in the target device.

In this paper, we examine the impact of the FPGA logic element architecture on the effectiveness of resource sharing. We conduct our analysis using two commercial Altera FPGA families: 1) Cyclone II [2] (4-LUTs) and 2) Stratix IV [3] (dual-output 6-LUTs). One of the contributions of this paper is to show conclusively the cases for which resource sharing is advantageous for FPGAs. Results show that certain operators (e.g. addition) that are not worth sharing in Cyclone II are indeed worth sharing in Stratix IV. This is due to the larger LUT size, which permits portions of the sharing multiplexer circuitry to be combined into the *same* LUTs that implement the operators themselves. We then show that there exist patterns of operators that occur commonly in circuits and that such patterns can be considered as *composite operators* that can be shared to provide area reductions. We use the sharing analysis results to drive decisions made in the binding phase of the LegUp open source HLS tool [13] built within the LLVM compiler [11].

2. BACKGROUND AND RELATED WORK

Recent research on resource sharing that specifically targets FPGAs includes [9, 8, 5, 14, 4]. The work of Cong and Jiang [6] bears the most similarity to our own in that it applied graph-based techniques to identify commonly occurring patterns of operators in the HLS of FPGA circuits, and then shared such patterns in binding for resource reduction. Some of the area savings achieved, however, were through the sharing of multipliers implemented using LUTs instead of using hard IP blocks. Implementing multipliers using LUTs is very costly, and thus offers substantial sharing opportunities.

The two commercial FPGAs targeted in this study have considerably different logic element architectures. In Cyclone II, combinational logic functions are implemented using 4-input LUTs. Stratix IV logic elements are referred to as adaptive logic modules (ALMs). An ALM contains a dual-output 6-LUT, which receives **8** inputs. Each of the outputs corresponds to an adaptive LUT (ALUT). The ALM can implement any single logic function of 6 variables, or alternately, can be *fractured* to implement two separate logic functions (using both outputs) – i.e. two ALUTs. The ALM can implement two functions of 4 variables, two functions with 5 and 3 variables, respectively, as well as several other combinations. In both architectures, a bypassable flip-flop is present for each LUT output.

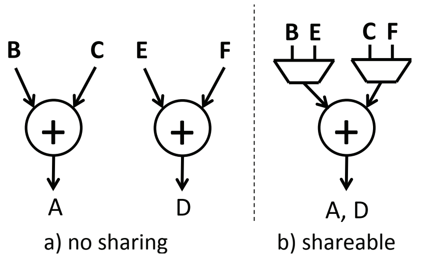

Figure 1: Illustration of sharing.

Table 1: Area data for individual 32-bit operators in the unshareable and shareable scenarios (ratios represent shareable/unshareable).

	Cyclone II		Stratix IV	
	Unshareable LEs	Shareable LEs	Unshareable ALMs	Shareable ALMs
Add/Sub	32	96 (3.00)	16	25 (1.56)
Bitwise	32	64 (2.00)	32	32 (1.00)
Compare	32	96 (3.00)	24	46 (1.92)
Div	1118	1182 (1.06)	568	599 (1.05)
Mod	1119	1183 (1.06)	581	613 (1.06)
Mult	689	747 (1.08)	221	362 (1.64)
Shift	173	215 (1.24)	75	94 (1.25)

3. SHARING INDIVIDUAL OPERATORS

We first investigate the value of sharing individual operators *outside* the context of a larger circuit. For each type of operator, we wish to know whether any area savings may arise from sharing its hardware implementation vs. the case of instantiating additional instances of the operator. An example of resource sharing is illustrated in Fig. 1 for two `C` statements: `A = B + C` and `D = E + F`. Fig. 1(a) shows an implementation without sharing, using two adders; Fig. 1(b) depicts the *shareable* case, in which the same two adders are now implemented using the same functional unit. We wish to compare the area consumed by both possible implementations, shareable vs. unshareable, for various types of operators. The utility of sharing clearly depends on how the multiplexers are implemented.

We created two different Verilog implementations for each operator type in the LLVM intermediate representation (IR). The first implementation contains a *single* instance of the operator. The second contains the operator instance with 2-to-1 multiplexers on its inputs. We implemented these two subcircuits in FPGAs to measure their area. The subcircuits were synthesized using Quartus II ver. 11.0. To measure area, we use logic element count (LE) for Cyclone II and the number of adaptive logic modules (ALMs) for Stratix IV. We use the Quartus II INI variable `fit_pack_for_density_light` to direct the tool to minimize circuit area when packing logic into ALMs/LEs and LABs [1].

Table 1 provides area results for sharing individual 32-bit-wide operators in Cyclone II and Stratix IV FPGAs. Each row of the table corresponds to an operator type. The row labeled **bitwise** represents the `AND`, `OR`, and `XOR` operators, all of which consume the same area. The left side of the table gives results for Cyclone II; the right side gives results for Stratix IV. Area is shown for both the unshareable (columns labeled **unshareable**) and shareable scenarios (columns labeled **shareable**). For the data corresponding to shareable operators, values in parentheses give the ratio in area vs. the unshareable case. Sharing provides a benefit when the ratio reported is less than 2; that is, less area is consumed by sharing the operator in hardware than by instantiating two instances of the operator.

Table 1 illustrates that in both FPGA architectures, sharing is useful (from the area-reduction perspective) for modulus, division, multiplication (implemented with LUTs) and bitwise shift. Modulus and division are implemented with LUTs in both architectures and consume considerable area in comparison with multiplexers. The shift represents a barrel shift. In Stratix IV, sharing is *also* beneficial for addition, subtraction, comparison, as well as all of the bitwise operations: `AND`, `OR`, `XOR`. The larger LUTs in Stratix IV allow some (or all) of the sharing MUXes to be combined into the same LUTs that implement the operators, and thus, for Stratix IV, sharing is useful for a broader set of operators.

Regarding the bitwise operator data for Stratix IV in Table 1, in the unshareable case, a 32-bit bitwise logical operator uses 32 ALMs; in the shareable case, 32 ALMs are also consumed. In the unshareable case, however, each output is a function of just 2 primary inputs. Since ALMs are dual-output and can implement *any* two functions of up to 4 inputs, the unshareable case *should* have consumed just 16 ALMs. Quartus did not produce an area-minimal implementation for this case.

4. SHARING COMPOSITE OPERATORS

We now consider composite operators (patterns), which are groups of individual operators that connect to one another in specific ways. We begin by defining the key concepts used in our pattern analysis algorithm:

Pattern graph: A directed dataflow graph representing a computational pattern. Each node in the graph is a two-operand operation from the LLVM IR. Each pattern graph has a single root (output) node. The number of nodes in a pattern graph is referred to as its *size*. We require the nodes in a pattern graph to reside in the same *basic block*, where a basic block is a contiguous set of instructions with a single entry point and a single exit point.

PatternMap: A container for pattern graphs that organizes pattern graphs based on size and functionality. A key operation performed by the PatternMap is the testing of two patterns for equivalence. The equivalence checking accounts for patterns which are functionally but not topologically equivalent due to the order of operands in commutative operations. Note that pattern graphs with different schedules are not considered functionality equivalent. That is, the corresponding nodes in two equivalent pattern graphs *must* have corresponding cycle assignments in the schedule (e.g. if two operators are chained together in a single cycle in one pattern graph, the corresponding operators must be chained in the equivalent pattern graph). HLS scheduling results are used to detect such cases.

Finally, note that two pattern graphs may contain the same set of operators connected in the same way, yet corresponding operators in the graphs have different bit widths. It is undesirable to consider the two pattern graphs as equivalent if there is a large "gap" in their operator bit widths. For example, it would not be advantageous to *force* an 8-bit addition to be realized with a 32-bit adder in hardware. We developed a simple bit width analysis pass within LLVM that computes the required bit widths of operators. Two pattern graphs are not considered as equivalent to one another if their corresponding operators differ in bit width by more than 10 (determined empirically). We also consider operator bit widths in our binding phase, described below.

Valid operations for patterns: We do not allow all operations to be included in pattern graphs. We exclude operators with constant inputs, as certain area-reducing synthesis optimizations are already possible for such cases. In addition, we do not allow division and modulus to be included in pattern graphs. The FPGA implementation of such operators is so large that, where possible, they should be left as

Table 2: Area for sequential patterns of operators in the unshareable and shareable scenarios (ratios represent shareable/unshareable).

	Cyclone II		Stratix IV	
	Unshared	Shareable	Unshared	Shareable
Pattern	LEs	LEs	ALMs	ALMs
Add_Add_Add_Add	128	288 (2.25)	64	73 (1.14)
Add_Sub	64	160 (2.50)	32	42 (1.31)
Add_XOR	64	128 (2.00)	33	42 (1.27)
Add_XOR_Add	96	192 (2.00)	48	58 (1.21)
OR_OR_OR	96	128 (1.33)	64	51 (0.80)
XOR_XOR	64	96 (1.50)	32	32 (1.00)

isolated operators and shared as much as possible (by wide multipliexers on their inputs).

4.1 Pattern Discovery Approach

All pattern graphs up to a maximum size S are discovered as follows: We iterate over all instructions in the program. Once a valid instruction is found, this becomes the root instruction, r, of a new pattern graph of size 1. We then perform a breadth-first search of the predecessors of r, adding all combinations of predecessors one at a time, to discover all graphs rooted at r. Each new graph is added to the PatternMap object and we stop once graph sizes exceed S or all graphs have been discovered. We then continue to the next instruction. In this work, we find all patterns up to size 10.

4.2 Pattern Sharing Analysis

We applied the pattern discovery approach described above to identify commonly occurring patterns in a suite of 13 `C` benchmark programs – the 12 CHStone HLS benchmarks [10], as well as `dhrystone`. Table 2 presents a sharing analysis for 6 patterns we found to be common. Each pattern listed occurs multiple times in at least one of the benchmarks. Our purpose here is not to exhaustively list *all* patterns that occur more than once in any benchmark; rather, our aim is to provide an illustrative analysis for the most commonly occurring patterns in these particular 13 benchmarks. The left column lists the pattern names, where each name defines the operators involved. For example, `Add_Add_Add_Add` is a pattern with 4 addition operators connected serially.

We follow the same analysis approach as described in Section 3. We created two Verilog modules for each pattern: one representing the unshareable case, a second having 2-to-1 MUXes on each input, representing the shareable case. The left side of Table 2 gives results for Cyclone II; the right side for Stratix IV. All operators in patterns are 32 bits wide. For the columns of the table representing area, resource sharing provides a "win" if the ratio in parentheses is less than 2 (see Section 3). The results in Table 2 are for sequential patterns, with registers on edges between operators.

For Cyclone II, we observe that sharing is beneficial in 2 of the 6 patterns (`OR_OR_OR` and `XOR_XOR`) from the area perspective. For Stratix IV, sharing is beneficial for all 6 patterns. For one of the patterns, `OR_OR_OR`, the shareable implementation consumed less area than the unshareable implementation. We investigated this and found that Quartus did not produce an area-minimal implementation for this pattern in the unshareable scenario, which we attribute to algorithmic noise. We also analyzed the impact of resource sharing for *combinational* patterns (i.e. patterns without registers on edges between operators) and found sharing to be less beneficial, owing to the ability of Quartus II to collapse chained operators together into LUTs, thereby reducing the opportunities for collapsing the sharing MUXes into the same LUTs as the operators.

We conclude that it is quite challenging to predict up-front when sharing will provide an area benefit, as it depends on the specific technology mapping and packing decisions made by Quartus, which appear to depend on the specific pattern implemented. However, we observe two general trends: 1) Sharing is more likely to be beneficial for composite operators that consume significant area, particularly when the MUXes that facilitate sharing can be rolled into the same LUTs as those implementing portions of the operator functionality. 2) Sharing is more advantageous when registers are present in patterns – registers prevent an efficient mapping of operators into LUTs, thereby leaving LUTs underutilized, with free inputs to accommodate MUX circuitry.

5. BINDING

For each pattern size (in descending order) we choose pairs of functionally-equivalent pattern graphs to be implemented by (bound to) a single shareable composite operator in the hardware. Any two graphs whose operations happen in non-overlapping clock cycles are candidates for sharing.

Consider two sharing candidates, patterns $P1$ and $P2$. We compute a *sharing cost* for the pair by summing the bit width differences in their corresponding operators:

$$SharingCost = \sum_{n1 \in P1, n2 \in P2} |width(n1) - width(n2)| \quad (1)$$

where $n1$ and $n2$ are corresponding operators in pattern graphs $P1$ and $P2$, respectively. The intuition behind (1) is that it is desirable for operation widths between pattern graphs sharing resources to be as closely aligned as possible.

However, two additional optimizations are possible that provide further area reductions:

1. Variable Lifetime Analysis: Our binding approach only pairs pattern graphs whose output values have non-overlapping lifetimes. Otherwise, separate output registers are required to store the values produced by each pattern. If pattern graph output value lifetimes do not overlap, a single register can be used. The LLVM compiler already has a pass to determine a variable's lifetime in terms of basic blocks spanned. We combine the results of this pass with the output of scheduling to determine the cycle-by-cycle lifetimes of each variable.

2. Shared Input Variables: If two patterns share an input variable, then adding a MUX on the input is unnecessary if the patterns are bound to the same hardware, saving MUX area. Hence, our binding algorithm prefers to pair patterns with shared input variables.

After computing the sharing cost using (1) for a pair of candidate patterns (based on their operator bitwidths), we adjust the computed cost to account for shared input variables between the patterns. Specifically, we count the number of shared input variables that feed into the two patterns and reduce the sharing cost for each such shared input variable (cost determined empirically).

Finally, we apply a greedy algorithm to bind pairs of pattern graphs to shared hardware units. Sharing candidates with the lowest cost are selected and bound to a single hardware unit. Note that owing to the costs of implementing MUXes in FPGAs, we allow a composite operator hardware unit to be shared at most twice. Once we have exhausted binding pattern graphs of a given size, we proceed to binding pattern graphs of the next smaller size. The problem is that of finding a minimum cost graph matching, and though we found that a greedy approach suffices, more sophisticated algorithms can be applied (e.g. [12]).

6. EXPERIMENTAL STUDY

We now present results for resource sharing in HLS binding for a set of 13 benchmark `C` programs – the 12 CHStone

Table 3: Area results for resource sharing using hard multipliers/DSP blocks.

	Cyclone II			Stratix IV		
Benchmark	No Sharing	Sharing Div/Mod	Sharing Div/Mod + Patterns	No Sharing	Sharing Div/Mod	Sharing Div/Mod + Patterns
adpcm	22541	21476 (0.95)	19049 (0.85)	8585	8064 (0.94)	7943 (0.93)
aes	18923	15418 (0.81)	15477 (0.82)	9582	8136 (0.85)	7929 (0.83)
blowfish	11571	11571 (1.00)	9306 (0.80)	6082	6082 (1.00)	5215 (0.86)
dfadd	7012	7012 (1.00)	6364 (0.91)	3327	3327 (1.00)	2966 (0.89)
dfdiv	15286	13267 (0.87)	13195 (0.86)	7043	5949 (0.84)	5915 (0.84)
dfmul	3903	3903 (1.00)	3797 (0.97)	1893	1893 (1.00)	1824 (0.96)
dfsin	27860	27982 (1.00)	26996 (0.97)	12630	11529 (0.91)	11094 (0.88)
gsm	10479	10479 (1.00)	10659 (1.02)	4914	4914 (1.00)	4537 (0.92)
jpeg	35792	34981 (0.98)	34316 (0.96)	17148	16703 (0.97)	16246 (0.95)
mips	3103	3103 (1.00)	2986 (0.96)	1610	1610 (1.00)	1493 (0.93)
motion	4049	4049 (1.00)	3897 (0.96)	1988	1988 (1.00)	1878 (0.94)
sha	11932	11932 (1.00)	12307 (1.03)	5909	5909 (1.00)	5856 (0.99)
dhrystone	5277	5277 (1.00)	5277 (1.00)	2598	2598 (1.00)	2598 (1.00)
Geomean:	**10419.82**	**10093.65**	**9677.25**	**4980.59**	**4788.06**	**4558.11**
Ratio:	**1.00**	**0.97**	**0.93**	**1.00**	**0.96**	**0.92**
Ratio:		**1.00**	**0.96**		**1.00**	**0.95**

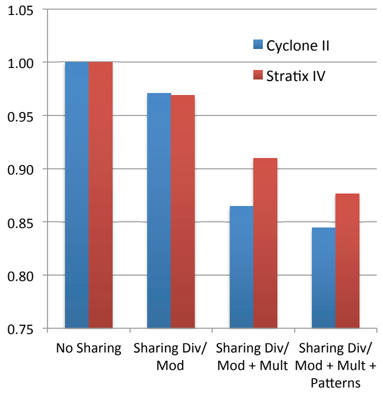

Figure 2: Normalized area results with soft (LUT-based) multipliers.

benchmarks, as well as `dhrystone`. For both target FPGA families, we evaluated several sharing scenarios that successively represent greater amounts of resource sharing: 1) No sharing; 2) sharing dividers and remainders (mod); 3) scenario #2 + sharing multipliers; and 4) Scenarios #2 + #3 + sharing composite operator patterns. The work in [6] implemented multipliers with LUTs instead of hard IP blocks, i.e. DSP blocks in Stratix IV and embedded multipliers in Cyclone II. To permit comparison with [6], we implemented the benchmarks in two ways: 1) with LUT-based multipliers, and 2) using hard multipliers. Scenario #3 applies only to the case of multipliers implemented with LUTs.

Table 3 gives area results for Cyclone II (left) and Stratix IV (right) when multipliers are implemented using hard IP blocks. Ratios in parentheses show the area reduction vs. the no sharing case. Observe that sharing division/modulus alone provides 3% and 4% average area reduction for Cyclone II and Stratix IV, respectively. Sharing patterns provides an additional 4% and 5% area reduction, on average, for Cyclone II and Stratix IV respectively.

Fig. 2 summarizes the average area results across all circuits for the case when multipliers are implemented with LUTs. Larger area reductions are observed, as expected, owing to the significant amount of area needed to realize multipliers with LUTs. For Cyclone II, a 16% reduction in LEs is observed when all forms of sharing are turned on (left bars for each scenario); for Stratix IV, a 12% reduction in ALMs is seen (right bars).

The pattern sharing approach introduced in this work provides a larger benefit in Stratix IV (4-5%) vs. Cyclone II (2-4%), due to the ability to exploit ALM under-utilization by combining MUX and operator functionality together into LUTs. While speed performance results are omitted for space reasons, we found that resource sharing reduced speed by 11%, on average, in both Cyclone II and Stratix IV, when all forms of sharing were turned on.

7. CONCLUSIONS AND FUTURE WORK

We investigated resource sharing for FPGAs and demonstrated that different resource sharing tradeoffs exist depending on the logic element architecture of the target FPGA. On average, resource sharing provides area reductions of 7-16% for Cyclone II, and 8-12% for Stratix IV, depending on whether multipliers are implemented using hard IP blocks or LUTs. Directions for future work include modifying the scheduling phase of HLS to encourage the generation of composite operator patterns with registers at specific points, in order to allow MUXes to be more easily combined together in LUTs with portions of the operator functionality.

8. REFERENCES

[1] Altera QUIP. *http://www.altera.com/education/univ/research/unv-quip.html*, 2009.

[2] Altera, Corp., *Cyclone II FPGA Family Data Sheet*, 2011.

[3] Altera, Corp., *Stratix IV FPGA Family Data Sheet*, 2011.

[4] E. Casseau and B. Le Gal. High-level synthesis for the design of FPGA-based signal processing systems. In *IEEE Int'l Symp. on Systems, Architectures, Modeling, and Simulation*, pages 25 – 32, 2009.

[5] D. Chen, et al. Optimality study of resource binding with multi-Vdds. In *IEEE/ACM DAC*, pages 580 – 585, 2006.

[6] J. Cong, et al. Pattern-based behavior synthesis for FPGA resource reduction. In *ACM FPGA*, pages 107–116, 2008.

[7] J. Cong, et al. High-level synthesis for FPGAs: From prototyping to deployment. *IEEE Trans. on CAD*, 30(4):473–491, 2011.

[8] J. Cong and J. Xu. Simultaneous FU and register binding based on network flow method. In *ACM/IEEE DATE*, pages 1057 – 1062, 2008.

[9] S. Cromar, et al. FPGA-targeted high-level binding algorithm for power and area reduction with glitch-estimation. In *ACM/IEEE DAC*, pages 838 – 843, 2009.

[10] Y. Hara, et al. Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis. *J. of Information Processing*, 17:242–254, 2009.

[11] http://www.llvm.org. *The LLVM Compiler Infrastructure Project*, 2010.

[12] V. Kolmogorov. Blossom v: A new implementation of a minimum cost perfect matching algorithm. *Mathematical Programming Computation 1*, 1(1):43–67, 2009.

[13] A. Canis, et al. LegUp: high-level synthesis for FPGA-based processor/accelerator systems. *ACM FPGA*, pages 33–37, 2011.

[14] W. Sun, et al. FPGA pipeline synthesis design exploration using module selection and resource sharing. *IEEE Tran. on CAD*, 26(2):254 – 265, 2007.