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

Names removed for blind review

Affiliation removed

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. Sharing a functional unit normally involves adding multiplexers to its inputs, which are costly to implement in FPGAs. Hence the prevailing is sentiment that sharing is rarely useful for FPGAs, except when the shared resource is either very large, or is relatively scarce. In this paper, we re-visit the resource sharing question in the FPGA context and show that the utility of sharing depends on the underlying FPGA logic element architecture. Specifically, we show that different sharing trade-offs exist when 4-LUTs vs. 6-LUTs are used. We further show that certain multi-operator patterns (e.g. add followed by subtract) occur multiple times in programs, thereby creating additional opportunities for sharing larger composite functional units – units comprised of patterns of interconnected operators. The 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 (uses 6-LUTs) and Cyclone II (uses 4-LUTs). Results show resource sharing in HLS can reduce area by up to 38% for Stratix IV designs (8-12%, on average), and 47% for Cyclone II designs (7-16%, on average).

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. HLS can be applied to both ASICs and FPGAs, though it is particularly relevant for FPGAs [7], owing to the growing popularity of FPGAs in high-performance computing and embedded applications, where they are used as accelerators to deliver improvements in computational throughput and energy-efficiency (e.g. [15]). While the use of FPGAs has historically required knowledge of hardware design techniques, HLS holds out the promise of making FPGA technology accessible to software engineers, who outnumber hardware engineers by 10-to-1 [19]. HLS, if successfully adopted, would allow FPGAs to be programmed in a manner similar to standard processors, greatly expanding their user base and growing the FPGA market. Indeed, the importance of HLS in the FPGA space was underscored recently by the acquisition of AutoESL (an HLS tool company) by Xilinx [16].

There are several traditional steps in high-level synthesis [9]. *Allocation* determines the number and types of functional units to be used in the hardware implementation (e.g. the number of multipliers that may be used). 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. The output of high-level synthesis is an RTL specification in VHDL or Verilog, synthesizeable using standard CAD tools.

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 clock cycle. 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. To see why, consider the example of implementing a 4-bit adder in a 4-LUT-based FPGA architecture. The 4-bit adder can normally be implemented using four 4-LUTs (and associated carry logic), while four 2-to-1 MUXes (needed to share an input port on the adder) also consume four 4-LUTs – the same number of LUTs as the adder itself! A key contribution of this paper is to show conclusively the cases for which resource sharing is advantageous for FPGAs.

In this paper, we study resource sharing in the FPGA context, and in particular, we examine the impact of the FPGA logic element architecture on the effectiveness of sharing. We conduct our analysis using two commercial Altera FPGA families: 1) Cyclone II, which contains 4-LUT-based logic elements, and 2) Stratix IV, which contains dual-output 6-LUT-based logic elements. Results show that certain operators (e.g. addition) that are not worth sharing in Cyclone II are indeed worth sharing in Stratix IV. We demonstrate that 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 move on to 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

FPGA '12 Monterey, California, USA.

analysis results to drive decisions made in the binding phase of an open source HLS tool [17]. Experimental results show resource sharing can reduce area by up to 38% for benchmark circuits implemented in Stratix IV FPGAs, and up to 47% for Cyclone II. To our knowledge, no prior work has studied the impact of FPGA architecture on high-level synthesis algorithms – a direction we believe to be fruitful for future research.

The remainder of the paper is organized as follows. Section 2 describes related work, including prior research on resource sharing in FPGA HLS. Section 3 presents our sharing analysis for individual operators, implemented in two commercial FPGA families with different logic architectures. In particular, we examine each operator type in isolation and the extent to which its area increases when it is made *shareable*, by adding MUXes to its inputs. Section 4 describes an algorithm for discovering commonly occuring composite operators in programs and presents an analysis of their sharing tradeoffs. Section 5 describes the modifications we made to the binding step of HLS to support resource sharing. Our experimental study is presented in Section 6. Conclusions and suggestions for future work are offered in Section 7.

2. RELATED WORK

In this section, we review prior work on resource sharing, describe the architecture of the two commercial FPGAs considered in this study, and give an overview of the HLS framework.

2.1 Resource Sharing in Binding

Resource sharing in binding has been considered since the earliest high-level synthesis efforts in the 1980s. Here, we review recent research that specifically targets FPGAs.

Cromar et al. considered resource sharing in FPGA HLS targeting both area and power reduction [10], and employed an early glitch estimation model. A 19% power and a 9% area reduction were reported relative to a baseline HLS system that performed considerable resource sharing – i.e. a baseline HLS where resource sharing may have contributed significantly to the overall area. In contrast, our work demonstrates the merits/costs of resource sharing relative to a *no sharing* scenario in an effort to show conclusively the cases where sharing is beneficial.

Other recent works include [8], which applied network flow techniques to the binding problem, and [5] which considered binding for multi-Vdd FPGAs. Sun et al. proposed an algorithm for combined module selection and resource sharing in the synthesis of pipelines implemented on FPGAs [18] – the work is applicable when multiple hardware implementations are available for a functional unit (e.g. an array multiplier and a booth multiplier). Casseau and Le Gal study integrated scheduling and binding in the synthesis of DSP-oriented FPGA circuits, and demonstrate an area reduction vs. separated synthesis steps [4].

Among prior work, that 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.

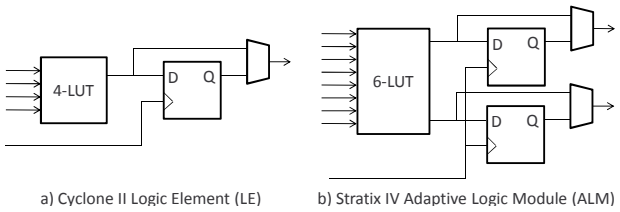

Figure 1: FPGA logic element architectures.

Our work is unique relative to the prior efforts mentioned above in that we consider the influence of the target FPGA architecture on HLS decisions.

2.2 FPGA Architecture

The two commercial FPGAs targeted in this study have considerably different logic element architectures. Fig. 1(a) shows a simplified version of the logic element in Cyclone II [2]. Combinational logic functions are implemented using 4-input LUTs, each of which is coupled with a flip-flop that may optionally be bypassed. For clarity, dedicated arithmetic circuitry is omitted from Fig. 1.

Fig. 1(b) depicts a logic element in Stratix IV [3] – referred to as an adaptive logic module (ALM). The 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. Specifically, the ALM can implement two functions of 4 variables, two functions with 5 and 3 variables, respectively, as well as several other combinations. A bypassable flip-flop is present for each LUT output.

2.3 High-Level Synthesis Tool

We investigate resource sharing using an open source high-level synthesis framework [17]. The HLS tool is implemented as new back-end passes of the LLVM compiler framework [12] – the same compiler used within Apple's SDK for iPod and iPhone development. In our HLS tool, an input `C` program is first parsed by LLVM and transformed into an intermediate representation (IR). The LLVM IR is essentially machine-independent assembly code, with instructions for implementing computations (e.g. add, shift, multiply) and control flow (e.g. branch). HLS operates directly on the LLVM IR: it schedules the instructions in the IR into clock cycles, and generates the corresponding finite state machine for control.

3. RESOURCE SHARING ANALYSIS FOR INDIVIDUAL OPERATORS

As a first step, we investigate the value of sharing of individual operators *outside* of the context of a larger circuit. For each type of operator, be it arithmetic or logical, we wish to know whether any area savings may arise from sharing that operator's hardware implementation vs. the case of instantiating additional instances of the operator.

An example of resource sharing is illustrated in Fig. 2. The top of the figure shows two `C` statements, each implementing addition. Fig. 2(a) shows an implementation without sharing, using two adders; Fig. 2(b) depicts the shareable

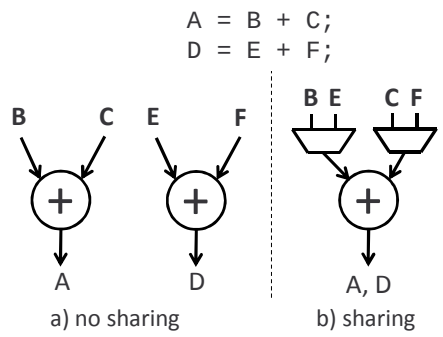

Figure 2: Illustration of sharing.

Table 1: Operators for which sharing is beneficial in Cyclone II and Stratix IV.

Cyclone II	Stratix IV
Mod	Mod
Div	Div
Mult	Mult
Shift	Shift
	Add/Subtract
	Compare
	Bitwise (AND, OR, XOR)

case. In the shareable case, 2-to-1 MUXes are present on the operator's inputs. We wish to compare the area consumed by both possible implementations, shareable vs. unshareable, for various types of operators, in two different FPGA architectures. As is apparent in the figure, the utility of sharing depends on how multiplexers are implemented, and their area, relative to the area of the operator being shared.

We created two different Verilog implementations for each operator type in the LLVM intermediate representation (IR). The first Verilog implementation contains a *single* instance of the operator. The second Verilog implementation, for evaluating the shareable case, contains the single operator instance with 2-to-1 multiplexers on its inputs. We implemented these two subcircuits in FPGAs to measure their delay and area. The subcircuits were synthesized to Cyclone II and Stratix IV 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, as suggested in [1]. Aside from this, Quartus II was run with default settings.

Table 1 lists the operators for which sharing is advantageous in Cyclone II and Stratix IV. In both architectures, sharing is useful 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 with non-constant inputs.

Aside from the 4 operations in which sharing proved useful for both target architectures, there are additional cases for which sharing is useful *only* for Stratix IV. In Stratix IV, sharing is beneficial for addition, subtraction, comparison, as well as all of the bitwise operations: `AND`, `OR`, exclusive-`OR`. Note that addition, subtraction and compare occur very frequently in programs and consequently, it is likely the choices made by HLS regarding sharing such operators would impact area/performance results considerably.

We investigated why sharing is beneficial for certain operators in Stratix IV, while not beneficial for the same operators in Cyclone II. Taking a 4-bit adder as an example, we found that the adder consumed 4 LEs in Cyclone II in the unshareable case, and 12 LEs in the shareable case. In the shareable case, 4 of the 12 LEs are used to implement the adder, and the remaining 8 LEs are required to implement the 8 2-to-1 MUXes on the adder inputs (4 on each 4-bit input port). Recall that Cyclone II LEs contain a 4-LUT and as such, it was not possible for Quartus II to collapse the adder and MUX functionality together to yield fewer overall LUTs. A 2-to-1 MUX is a 3-input function and each full adder receives 2 data inputs (and one carry input via the carry chain). Since Cyclone II LEs contain 4-input LUTs, MUXes and full adders cannot be combined into a single LE. Thus we conclude that for Cyclone II, it is cheaper from the area perspective to instantiate additional adders rather than share them.

In Stratix IV, a 4-bit adder consumes two and three ALMs in the unshareable and shareable cases, respectively. That is, two unshareable 4-bit adders would consume four ALMs – a 33% increase over the shareable case. The reason for this area difference is shown in Fig. 3, which is a snapshot of a shareable 4-bit adder in the Quartus II technology mapped viewer. Labels show the function implemented by each ALUT. Two of the six ALUTs implement 2-to-1 MUXes; two ALUTs implement a full adder with 2-to-1 MUXes on its inputs; and, two ALUTs implement a 2-to-1 MUX and an adder. With the larger LUTs in Stratix IV, Quartus II synthesis is able to *combine* portions of adder and MUX functionality together into ALUTs, and then pair ALUTs together into ALMs, producing savings in total ALM count vs. the unshareable case. Similar results were observed for other operators in Stratix IV – larger LUTs in the target fabric allow MUX functionality to collapse in with operator functionality, reducing area, and making sharing advantageous.

To gain further insight into why sharing adders offers an advantage in Stratix IV, we examined the LUT input usage distribution in the unshareable and shareable scenarios. When a 32-bit unshareable adder is implemented, Quartus reports that the design contains 32 LUTs with ≤ 3 used inputs. Such LUTs are under-utilized and can accommodate additional logic. When the shareable 32-bit adder is implemented, Quartus reports 16 LUTs with 5 used inputs (each LUT implements two MUXes and an adder), 16 LUTs with 4 inputs (one MUX and an adder), and 16 LUTs with ≤ 3 inputs (adder). The shareable adder contains more highly utilized LUTs, as the MUX and addition circuitry have been combined together. In the shareable case, each 5-input function can be paired with a 3-input function and implemented in a single Stratix IV ALM (which can implement two functions where one uses 5 inputs and the second uses 3 inputs); the 4-input functions can be grouped into pairs, with each pair likewise implemented in a single ALM.

Table 2 provides detailed area and speed (in MHz) results for sharing individual operators in both Cyclone II and Stratix IV. Each row of the table corresponds to an operator type (32 bit wide input). The row labeled **bitwise** represents the `AND`, `OR`, and `XOR` operators, all of which con-

Table 2: Area and speed data for individual operators in the unshareable and shareable scenarios (ratios in parentheses represent the differences between the shararable vs. unshareable scenarios).

	Cyclone II				Stratix IV			
	Unshareable		Shareable		Unshareable		Shareable	
	LEs	Fmax	LEs	Fmax	ALMs	Fmax	ALMs	Fmax
Add/Sub	32	234.2	96 (3.00)	213.3 (0.91)	16	617.7	25 (1.56)	459.4 (0.74)
Bitwise	32	420.2	64 (2.00)	420.2 (1.00)	32	800	32 (1.00)	800 (1.00)
Compare	32	213.6	96 (3.00)	184.5 (0.86)	24	500	46 (1.92)	423 (0.85)
Div	1118	8.1	1182 (1.06)	8.1 (1.00)	568	18.6	599 (1.05)	18.5 (0.99)
Mod	1119	8.3	1183 (1.06)	8.1 (0.98)	581	17.9	613 (1.06)	19.1 (1.07)
Mult	689	82.5	747 (1.08)	76.9 (0.93)	221	168.3	362 (1.64)	163.8 (0.97)
Shift	173	191.1	215 (1.24)	186.8 (0.98)	75	443.7	94 (1.25)	266.4 (0.60)

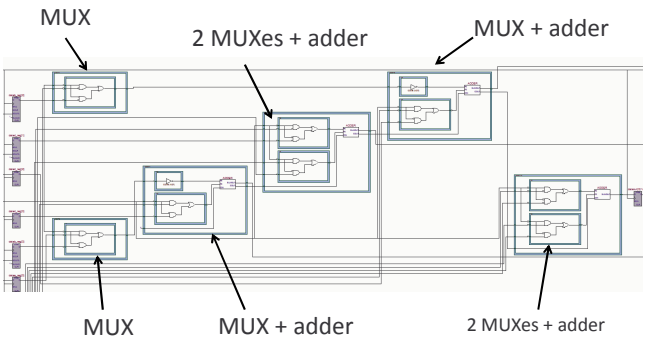

Figure 3: 4-bit shareable adder in Stratix IV (from Quartus II technology mapped viewer).

sume the same area. The left side of the table gives results for Cyclone II; the right side gives results for Stratix IV. Area and speed 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/speed vs. the unshareable case. For area, 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.

The right side of Table 2 (Stratix IV) shows that sharing is particularly advantageous in the case of the larger operators (i.e. divide, multiply, mod and shift), as well as for the bitwise operators. With regard to speed performance, we see in the right-most column of the table that there is a considerable slow-down when operators are made shareable: the shareable operators are, on average, 13% slower than in the unshareable case.

It is worthwhile to consider the bitwise operator data for Stratix IV in Table 2. 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. Despite this, Table 2 presents the results actually achieved by Quartus II, and we believe the sub-optimal packing of the unshareable bitwise operator is perhaps due to algorithmic noise within the tool.

The results in this section suggest that HLS binding algorithms should be tailored to the logic element architecture of the target FPGA. Table 2 confirms conventional wisdom that there are relatively few cases wherein sharing holds value in 4-LUT-based FPGA architectures; however, surprisingly, we observe important cases where sharing is beneficial in 6-LUT-based architectures – add/subtract/compare/bitwise. In the next section, we move on to consider the value of sharing patterns of multiple interconnected operators.

4. RESOURCE SHARING ANALYSIS FOR COMPOSITE OPERATORS – PATTERNS

Having studied the value of sharing individual operators, we now move on to consider composite operators, which are groups of individual operators that connect to one another in specific ways. We first describe our approach to analyzing the LLVM IR of a program to discover the patterns of operators that are candidates for sharing. We then give results showing the most popular computational patterns in a set of benchmark circuits and an analysis of whether such patterns are worth sharing. We begin by defining the key concepts used in our pattern analysis algorithm:

Pattern graph: A pattern graph is a directed dataflow graph representing a computational pattern. Each node in the graph is a binary operation (from the LLVM IR), i.e. nodes are 2-operand operations. Each pattern graph has a single root (output) node. In this way, a graph resembles a directed binary tree, except that re-converging paths are possible. An example pattern graph is shown in Fig. 4(a). The number of nodes in a pattern graph is referred to as its *size*. Note that we require the nodes in a pattern graph to reside in the same *basic block* of the program being analyzed, where a basic block is a contiguous set of instructions with a single entry point at the beginning of the set, and a single exit point at the end[1].

PatternMap: The PatternMap is a container for pattern graphs. It organizes patterns of operations by their size and functionality. A key operation performed by the PatternMap is the testing of two patterns for equivalence. Because patterns are directed graphs, this requires solving the graph isomorphism problem – known to be NP-hard. However, in our case, the pattern graphs are relatively small (up to size 10) and we found that using breadth-first search for comparison of two pattern graphs was computationally

[1]Patterns straddling basic block boundaries are not viable for sharing as a branch instruction may result in the pattern being "entered" at an intermediate rather than a leaf node.

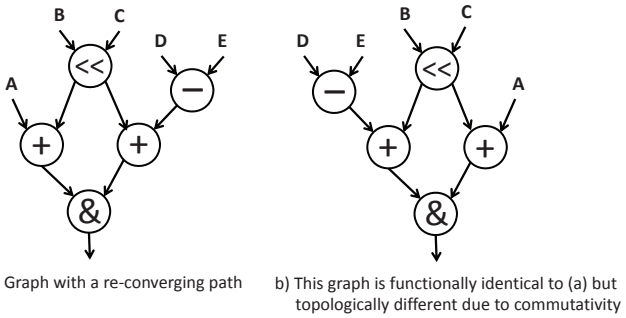

Figure 4: Directed Graphs representing a computational pattern of size 5.

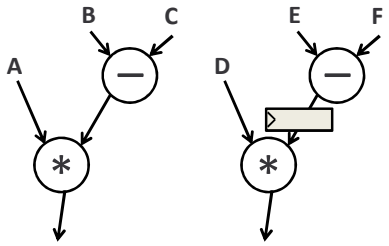

Figure 5: Two patterns that are not functionally equivalent as a result of scheduling.

tractable. Our matching approach is similar to structural graph matching as applied to standard cell technology mapping (e.g. as in [13]).

The pattern graph equivalence checking in the PatternMap object accounts for patterns which are functionally but not topologically equivalent due to the order of operands in commutative operations. For example, the pattern graph in Fig. 4(b) is functionally equivalent to that in Fig. 4(a), yet they contain topology differences that reflect the commutativity of inputs to operators (addition in this case). Furthermore, note that pattern graphs with different schedules are not considered functionality equivalent. Fig. 5 shows two patterns that contain the same sequence of operations. However, in the pattern on the left, the two operations are chained together in a single cycle, whereas on the right, the two operations happen in successive clock cycles – a register is present on the edge between operations. The graph equivalence checking uses the scheduling results to detect such cases. A direction for future work is to influence the scheduling step itself to produce identical schedules for popular operation combinations.

Finally, note that the LLVM IR also represents the bit widths of operators. It is thus possible that two pattern graphs may contain the same set of operators (nodes), 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. For example, consider a variable x that is logical `AND`'ed with a constant `60`. We deduce that the result will require at most `6` bits. As another example, the width of a variable produced by an `add` instruction is `1` + the maximum width of either operand. Right shift instructions may also produce variables with smaller widths than their operands.

Following bit width analysis, we handle bit width differences using a straightforward approach: 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), or if the inputs to the pattern graph leaf nodes differ in bit width. 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. For example, an adder with a constant-`1` input has a lower cost than an adder whose inputs both are 32-bit non-constants, and the former would be optimized by standard synthesis tools. But if these two additions are bound to the same adder functional unit, then optimizations cannot be performed on the adder with the constant input. Therefore, operations with constant inputs are disqualified from pattern inclusion. 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 isolated operators and shared as much as possible (i.e. by wide multipliexers on their inputs).

4.1 Pattern Discovery Approach

An iterative algorithm is applied to discover the pattern graphs in programs, whereby we first find small patterns, record them in the PatternMap, then repeat the process for successively larger patterns. In this work, we find all patterns up to size 10.

We identify the patterns of size S as follows: We iterate over all instructions in the program, skipping over instructions that are not valid for pattern inclusion (see above). Once a valid instruction is found, this becomes the root instruction, r, of a new pattern graph. We then perform a breadth-first search (BFS) to find all pattern graphs of size S rooted at r. All nodes in each pattern graph identified must be valid. Each pattern graph is then added to the PatternMap object as a pattern graph. The PatternMap object keeps track of which pattern graphs are functionally equivalent – such graphs are candidates for sharing during binding (described below). Having found all pattern graphs of size S, we increment S and we *grow* the patterns of size S to form the patterns of size $S + 1$.

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 13 programs consist of the 12 CHStone HLS benchmarks [11], as well as `dhrystone`. Table 3 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, and a second having 2-to-1 MUXes on each input, representing the shareable case. The left side of Table 3 gives analysis results for Cyclone II; the right side for Stratix IV. Ratios in parentheses compare the shareable case to unshareable case. 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 (meaning that the pattern implementation with MUX'ed inputs is cheaper than instantiating two unshareable pattern instances). Note that the data in Table 3 is for combinational patterns of interconnected operators (there are no registers on the edges between operators). Table 4, described below, gives analysis data for the case of sequential patterns, with registers on each edge between operators.

Looking first at the results for Cyclone II in Table 3, we see that all ratios in the LEs column are greater than or equal to 2. None of these patterns are worth sharing in Cyclone II, and moreover, speed drops by up to 18% in the shareable scenario. Turning to the Stratix IV results, 3 of the 6 patterns appear to be worth sharing. `Add_Add_Add_Add`, `Add_XOR` and `XOR_XOR` all have area ratios less than 2. Observe that the shareable `XOR_XOR` consumes the same number of ALMs as the unshareable case – a favorable sharing result. In the unshareable `XOR_XOR`, each output of the pattern is a 3-input logic function of the primary inputs, implementable in a 3-LUT. Since Stratix IV contains larger LUTs, the 3-input functions can be combined with MUX circuitry to realize a compact implementation in the shareable case.

The Stratix IV data in Table 3 shows that making the `Add_Sub` pattern shareable is a particularly disastrous choice from the area angle – the shareable version of this pattern consumes nearly 4× as many ALMs as the unshareable version. We studied this case and found that in the unshareable version, the addition and subtraction were collapsed together into LUTs by Quartus, leading to a very compact implementation. Conversely, in the shareable case, it was not possible to collapse the MUX circuitry into the same LUTs as the operators (as had happened in Fig. 3), as the operator LUTs were already "filled up" with the add/subtract. The MUX circuitry was implemented separately from the operators, leading to a significant area hit.

Table 4 presents sharing analysis data for the case of sequential patterns, where a register is present between each operator in the pattern graph. The registers between pairs of interconnected operators make it impossible for Quartus II to collapse the functionality of operators together into LUTs. In the unshareable case, the LUTs are therefore less heavily utilized – they have free inputs. The free inputs permit MUXes to be collapsed in with the operators in the shareable scenario, reducing the area cost of sharing. Indeed, for Cyclone II, we observe that sharing is beneficial in 2 of the 6 patterns (`OR_OR_OR` and `XOR_XOR`). 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 likewise see several cases for Stratix IV where the shareable scenario had a slightly higher speed (Fmax) than the unshareable scenario, which we also believe to be noise in the tool.

From the data in Tables 3 and 4, 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 that prevent an efficient mapping of operators into LUTs, thereby leaving LUTs with free inputs and have unused "space" to accommodate MUX circuitry. Despite these challenges, in the next section we describe a general approach to resource sharing during binding that we found works well for a broad suite of benchmark circuits.

5. BINDING

Our approach to resource sharing in binding proceeds in a manner similar to the pattern discovery approach described in the previous section, except in this case we first consider large pattern graphs for binding, then move on to smaller graphs. For each pattern graph size, we choose pairs of functionally equivalent pattern graphs to be implemented by (bound to) a single shareable composite operator in the hardware. Any two pattern graphs whose operations happen in non-overlapping clock cycles are candidates for sharing. However, two additional optimizations are possible that provide further area reductions.

5.1 Variable Lifetime Analysis

Consider two functionally equivalent pattern graphs that are candidates for binding to the same hardware, i.e. their operations happen in different clock cycles. If the values computed by the pattern graphs have non-overlapping lifetimes, then a further optimization is possible, described below. To illustrate the lifetime concept, Fig. 6 shows two pattern graphs, $P1$ and $P2$, that are sharing candidates. In Fig. 6(a), the value computed by $P1$ is produced in cycle #2 and used in cycle #6; the value computed by $P2$ is produced in cycle #5 and used in cycle #6. The lifetimes of the two values overlap in the schedule. Fig. 6(b), shows the case where the values produced by the two patterns do not have overlapping lifetimes. $P1$'s value is produced and consumed in cycles 2 and 3, respectively. $P2$'s value is produced and consumed in cycles 5 and 6, respectively.

Consider now, the impact of the variable lifetimes on the synthesized hardware. In the case of overlapping lifetimes, such as Fig. 6(a), two separate registers must be used to store the values produced by the two patterns. However, in the non-overlapping case (Fig. 6(b)), a single register can be used to store the value produced by $P1$, and the same register can be re-used later to store the value produced by $P2$. In fact, registers on intermediate edges of the pattern graph are also re-used in the non-overlapping lifetimes case. Binding pattern graphs together that produce variables with non-overlapping lifetimes is therefore desireable, as it reduces the overall register count. The LLVM com-

Table 3: Area and speed results for combinational patterns of operators in the unshareable and shareable scenarios (ratios in parentheses represent differences between the shareable vs. unshareable scenarios).

	Cyclone II				Stratix IV			
	Unshareable		Shareable		Unshareable		Shareable	
Pattern	LEs	Fmax	LEs	Fmax	ALMs	Fmax	ALMs	Fmax
Add_Add_Add_Add	128	141.1	288 (2.25)	125.2 (0.89)	48	249.09	90 (1.88)	251.13 (1.01)
Add_Sub	64	182.5	160 (2.50)	161.8 (0.89)	17	548.8	67 (3.94)	462.3 (0.84)
Add_XOR	64	223.6	128 (2.00)	184.3 (0.82)	16	584.4	41 (2.56)	366.4 (0.63)
Add_XOR_Add	96	157.8	192 (2.00)	135.9 (0.86)	32	363.5	59 (1.84)	257.1 (0.71)
OR_OR_OR	32	420.2	96 (3.00)	420.2 (1.00)	32	800	64 (2.00)	800 (1.00)
XOR_XOR	32	420.2	96 (3.00)	420.2 (1.00)	32	800	32 (1.00)	800 (1.00)

Table 4: Area and speed results for sequential patterns of operators in the unshareable and shareable scenarios (ratios in parentheses represent differences between the shareable vs.unshareable scenarios)

	Cyclone II				Stratix IV			
	Unshareable		Shareable		Unshareable		Shareable	
Pattern	LEs	Fmax	LEs	Fmax	ALMs	Fmax	ALMs	Fmax
Add_Add_Add_Add	128	224.1	288 (2.25)	212.9 (0.95)	64	410.2	73 (1.14)	457.3 (1.11)
Add_Sub	64	244.4	160 (2.50)	215.3 (0.88)	32	431.2	42 (1.31)	455.8 (1.06)
Add_XOR	64	248.5	128 (2.00)	216 (0.87)	33	605.7	42 (1.27)	490.9 (0.81)
Add_XOR_Add	96	208.7	192 (2.00)	217.2 (1.04)	48	539.1	58 (1.21)	448 (0.83)
OR_OR_OR	96	420.2	128 (1.33)	420.2 (1.00)	64	800	51 (0.80)	800 (1.00)
XOR_XOR	64	420.2	96 (1.50)	420.2 (1.00)	32	800	32 (1.00)	800 (1.00)

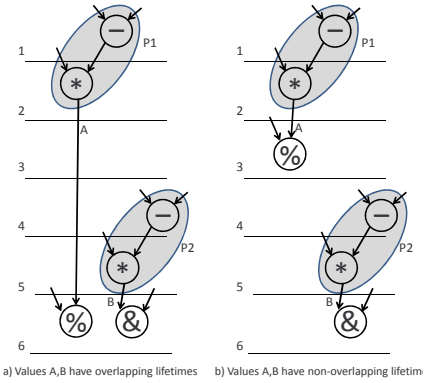

Figure 6: Illustration of variable lifetimes.

piler 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. We use the variable lifetimes in binding to ensure that variables with overlapping lifetimes are *not* bound to the same hardware unit.

5.2 Shared Input Variables

An additional optimization pertains to patterns that share input variables. Fig. 7 shows three addition operations that are candidates for being bound to a single adder functional unit in hardware. The first and second additions receive a common variable as input, A, whereas the third addition has no variables in common with the others. If the first and second additions are bound together, one input multiplexer can be saved, as shown on the right side of the figure. Hence, binding additions 1 and 2 together is the preferred choice.

5.3 Binding Implementation

We proceed with binding after having constructed the PatternMap and determined the lifetimes of all variables. For every pattern graph, we thus have a list of all other pattern graphs that may be bound to the same hardware unit as it. Resource sharing has the most positive impact when the resource being shared is large, and consequently, we first bind the largest pattern graphs, and then move on to bind smaller pattern graphs (containing operators that have not already been bound as part of a larger pattern graph).

Any two pattern graphs that are equivalent and produce values with non-overlapping lifetimes are termed *sharing candidates*. Consider two sharing candidates, $P1$ 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 if operation widths between pattern graphs that share resources are as closely aligned as possible. Finally, we adjust the cost of each pair of sharing candidates using the concept described in Section 5.2. We count the number of shared input variables that feed into the two patterns, and we reduce the sharing cost for each such shared input variable (cost determined empirically).

Using the sharing candidate costs, 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 we are solving is essentially that of finding a

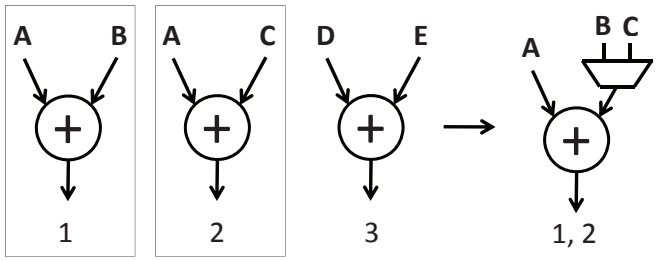

Figure 7: Adders 1 and 2 share the common input A which makes them a preferred pair because only one input multiplexer is needed. Adder 3 would not be shared.

minimum cost graph matching, and though we found that a greedy approach suffices, more sophisticated algorithms can certainly be applied (e.g. [14]).

6. EXPERIMENTAL STUDY

We now present results for resource sharing in HLS binding for a set of 13 benchmark `C` programs. We target both Cyclone II and Stratix IV and use the 12 CHStone benchmarks [11], as well as `dhrystone`. For both target FPGA families, we evaluated several sharing scenarios that successively represent greater amounts of resource sharing:

- **Scenario #1:** No sharing.
- **Scenario #2:** Sharing dividers and remainders (modulus).
- **Scenario #3:** Scenario #2 + sharing multipliers.
- **Scenario #4:** All of the above + 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 above applies only to the case of multipliers implemented with LUTs.

Table 5 presents resource sharing area results for Cyclone II. The first column lists the benchmarks. The next three columns give sharing results for the case of multipliers implemented using embedded multipliers, and correspond to scenarios #1, #2, and #4 above, respectively. The last four columns give results for the case of multipliers implemented using LUTs. These columns correspond to scenarios #1-4 above. The fourth row from the bottom gives the geometric mean results for each column. The last three rows contain ratios of the geometric means to allow us to assess the impact of more resource sharing being enabled. Each area datapoint in the table has a ratio, in parentheses, comparing the circuit area with respect to the same circuit without any resource sharing.

When embedded multipliers are used to implement multiplication (left side of Table 5), sharing dividers/remainders provides a 3% reduction in LEs. Naturally, only benchmarks that contain multiple division/remainder operations are affecting by enabling sharing for such operators. An additional 4% area reduction is observed when resource sharing

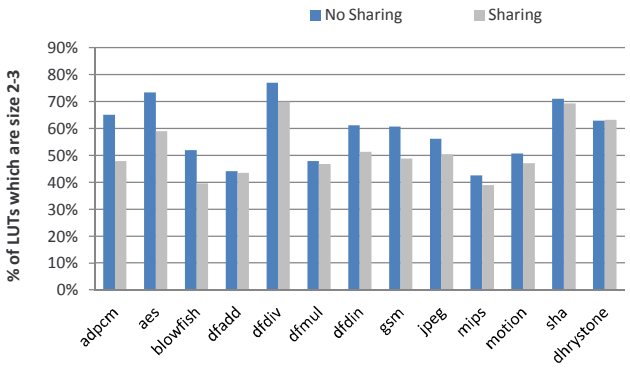

Figure 8: Distribution of small LUTs without and with resource sharing (Stratix IV).

for composite operator patterns is enabled. Overall, resource sharing provides a 7% LE reduction vs. the no sharing case, with the best results observed for circuit `blowfish` – a 20% reduction in LEs. Resource sharing of composite patterns was advantageous for all but 2 of the benchmark circuits: `gsm` and `sha`. As noted in Section 4.2, the impact of resource sharing for composite operators is difficult to predict given that it depends on specific optimization decisions made by Quartus II.

When multipliers are implemented using LUTs (right side of Table 5), resource sharing provides larger area reductions, as expected. In this case, sharing dividers/remainers provides a 3% reduction in LEs vs. the no sharing case. When multipliers are also shared, an additional 11% reduction in LEs is observed. Sharing composite operator patterns provides a further 2% reduction. Overall, resource sharing offers a 16% reduction in LEs. The best area-reduction results, 47%, were achieved for the circuit `adpcm`, which contains a large number of multiplications.

Table 6 presents resource sharing area results for Stratix IV. Beginning with the left side of the table, corresponding to the use of hard DSP blocks, we observe that sharing dividers/remainders provides a 4% ALM reduction vs. the no sharing case. An additional 5% ALM reduction is achieved when the resource sharing of composite operator patterns is enabled. Overall, we see that resource sharing provides an 8% area reduction, on average, vs. the no sharing case. The best results were achieved for the circuit `aes`, for which sharing composite operators provided a 17% reduction in ALM usage.

When LUT-based multipliers are used in Stratix IV, sharing dividers/remainders provides a 3% area reduction. When multipliers are also shared, an additional 6% reduction is observed. Sharing composite operator patterns provides an additional 4% reduction. Overall, sharing offers a 12% area reduction, on average. The multiplier-heavy circuit `adpcm` showed the best results with resource sharing enabled: a 38% reduction in ALM usage.

Observe that for Stratix IV, sharing composite operator patterns provides a benefit for *all* circuits in both styles of multiplier implementation, with the exception of `dhrystone`. This echoes the results in Section 4.2, wherein sharing

Table 5: Cyclone II resource sharing area results. Values in the table are LEs. Values in parentheses represent ratios relative to the no sharing case.

	Multiplication Using Embedded Mults			Multiplication Using LUT-Based Multipliers			
Benchmark	No Sharing	Sharing Div/Mod	Sharing Div/Mod + Patterns	No Sharing	Sharing Div/Mod	Sharing Div/Mod + Mult	Sharing Div/Mod + Mult + Patterns
adpcm	22541	21476 (0.95)	19049 (0.85)	46702	45696 (0.98)	23802 (0.51)	24933 (0.53)
aes	18923	15418 (0.81)	15477 (0.82)	18923	15418 (0.81)	15418 (0.81)	15342 (0.81)
blowfish	11571	11571 (1.00)	9306 (0.80)	11571	11571 (1.00)	11571 (1.00)	9306 (0.80)
dfadd	7012	7012 (1.00)	6364 (0.91)	7012	7012 (1.00)	7012 (1.00)	6258 (0.89)
dfdiv	15286	13267 (0.87)	13195 (0.86)	22404	20421 (0.91)	19217 (0.86)	19151 (0.85)
dfmul	3903	3903 (1.00)	3797 (0.97)	8669	8669 (1.00)	8669 (1.00)	8613 (0.99)
dfsin	27860	27982 (1.00)	26996 (0.97)	40353	38449 (0.95)	37277 (0.92)	36407 (0.90)
gsm	10479	10479 (1.00)	10659 (1.02)	18203	18203 (1.00)	13584 (0.75)	13762 (0.76)
jpeg	35792	34981 (0.98)	34316 (0.96)	49218	48388 (0.98)	38755 (0.79)	38273 (0.78)
mips	3103	3103 (1.00)	2986 (0.96)	5732	5732 (1.00)	4377 (0.76)	4114 (0.72)
motion	4049	4049 (1.00)	3897 (0.96)	4049	4049 (1.00)	4036 (1.00)	4228 (1.04)
sha	11932	11932 (1.00)	12307 (1.03)	11932	11932 (1.00)	12069 (1.01)	12449 (1.04)
dhrystone	5277	5277 (1.00)	5277 (1.00)	5291	5291 (1.00)	5351 (1.01)	5351 (1.01)
Geomean:	**10419.82**	**10093.65**	**9677.25**	**13921.41**	**13515.54**	**12034.45**	**11752.99**
Ratio:	**1.00**	**0.97**	**0.93**	**1.00**	**0.97**	**0.86**	**0.84**
Ratio:		**1.00**	**0.96**		**1.00**	**0.89**	**0.87**
Ratio:						**1.00**	**0.98**

Table 6: Stratix IV resource sharing area results. Values in the table are ALMs. Values in parentheses represent ratios relative to the no sharing case.

	Multiplication Using DSP Blocks			Multiplication Using LUT-Based Multipliers			
Benchmark	No Sharing	Sharing Div/Mod	Sharing Div/Mod + Patterns	No Sharing	Sharing Div/Mod	Sharing Div/Mod + Mult	Sharing Div/Mod + Mult + Patterns
adpcm	8585	8064 (0.94)	7943 (0.93)	18951	18438 (0.97)	11909 (0.63)	11722 (0.62)
aes	9582	8136 (0.85)	7929 (0.83)	9582	8136 (0.85)	8136 (0.85)	7929 (0.83)
blowfish	6082	6082 (1.00)	5215 (0.86)	6082	6082 (1.00)	6082 (1.00)	5215 (0.86)
dfadd	3327	3327 (1.00)	2966 (0.89)	3327	3327 (1.00)	3327 (1.00)	2966 (0.89)
dfdiv	7043	5949 (0.84)	5915 (0.84)	9352	8277 (0.89)	8203 (0.88)	8204 (0.88)
dfmul	1893	1893 (1.00)	1824 (0.96)	3170	3170 (1.00)	3170 (1.00)	3105 (0.98)
dfsin	12630	11529 (0.91)	11094 (0.88)	16631	15418 (0.93)	15523 (0.93)	15129 (0.91)
gsm	4914	4914 (1.00)	4537 (0.92)	7630	7630 (1.00)	6252 (0.82)	6043 (0.79)
jpeg	17148	16703 (0.97)	16246 (0.95)	22349	21853 (0.98)	19592 (0.88)	19127 (0.86)
mips	1610	1610 (1.00)	1493 (0.93)	2471	2471 (1.00)	2299 (0.93)	2210 (0.89)
motion	1988	1988 (1.00)	1878 (0.94)	1988	1988 (1.00)	1982 (1.00)	1930 (0.97)
sha	5909	5909 (1.00)	5856 (0.99)	5909	5909 (1.00)	5947 (1.01)	5917 (1.00)
dhrystone	2598	2598 (1.00)	2598 (1.00)	2602	2602 (1.00)	2607 (1.00)	2607 (1.00)
Geomean:	**4980.59**	**4788.06**	**4558.11**	**6273.87**	**6078.47**	**5709.30**	**5499.92**
Ratio:	**1.00**	**0.96**	**0.92**	**1.00**	**0.97**	**0.91**	**0.88**
Ratio:		**1.00**	**0.95**		**1.00**	**0.94**	**0.90**
Ratio:						**1.00**	**0.96**

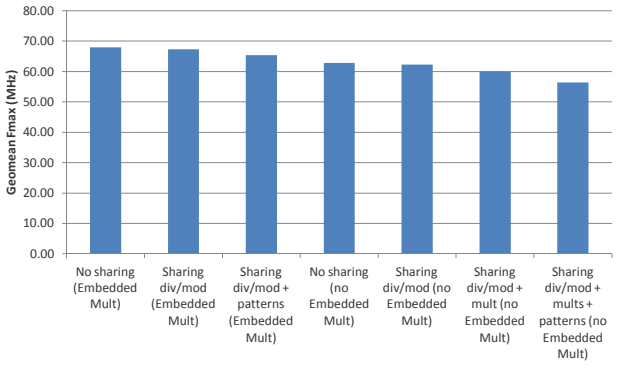

Figure 9: Cyclone II speed performance results.

composite operators provided larger benefits in Stratix IV vs. Cyclone II.

Fig. 8 gives further evidence that resource sharing provides a benefit when the MUXes that facilitate sharing can be combined into the LUTs that perform computation. The figure shows the percentage of small 2-3 input LUTs in each benchmark without resource sharing and with resource sharing enabled. Without any sharing, the designs contain a large number of small LUTs (nearly 80% in the case of `df-div`). When resource sharing is turned on, the fraction of small LUTs decreases for each design, suggesting that the MUXes are being implemented by using up free inputs on small LUTs.

Finally, we consider the impact of resource sharing on circuit speed performance. Geometric mean performance results for Cyclone II appear in Fig. 9. Detailed circuit-by-circuit results are omitted due to space restrictions. As one might expect, circuit speed decreases as increasing amounts

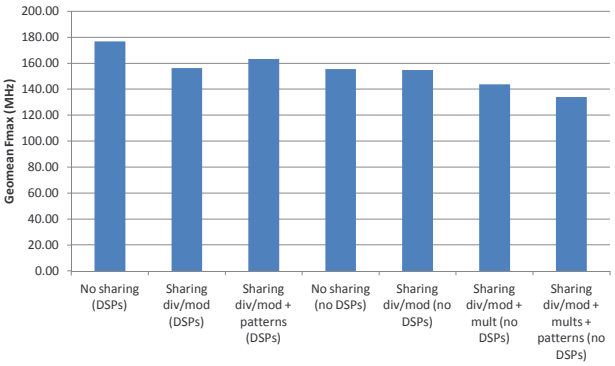

Figure 10: Stratix IV speed performance results.

of resource sharing are enabled, due to the presence of the MUXes on the operator inputs. When embedded multipliers are used, sharing divide/remainder reduces circuit speed by about 1%; sharing composite operator patterns lowers speed by 4%. When LUT-based multipliers are used, circuit speed is reduced by 11% when all forms of sharing are enabled.

Speed performance data for Stratix IV appears in Fig. 10. The changes in speed performance due to resource sharing match quite closely with those for Cyclone II. When hard DSP blocks are used, resource sharing impacts speed by 1% and 4% respectively, for sharing divide/remainder and sharing composite operators. When LUT-based multipliers are used and all forms of resource sharing are enabled, circuit speed drops by 11%, on average. It is worthwhile to mention that in many applications, circuits do not need to be run at the maximum possible speed, and resource sharing provides significant area reductions that we believe will be of value in area-sensitive and/or embedded settings.

7. CONCLUSIONS AND FUTURE WORK

Resource sharing is a classical approach to area reduction in high-level synthesis. In this paper, we considered resource sharing for FPGAs and demonstrated that different resource sharing tradeoffs exist depending on the logic element architecture of the target FPGA. We evaluated operators in isolation and showed that certain operators that are not worth sharing in Cyclone II (4-LUT-based logic elements) are indeed worth sharing in Stratix IV (dual-output 6-LUT-based logic elements). We also found that programs frequently contain multiple instances of composite operators, which are patterns of interconnected operators. Resource sharing of (large) composite operators provides additional area reduction opportunities. 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. In addition, our HLS system currently synthesizes `C` programs to circuits on a function-by-function basis and as such, the work in this study did not consider resource sharing across functions (aside from those functions that were automatically inlined). Resource sharing between functions may therefore offer additional chances for area reduction.

8. REFERENCES

[1] Quartus-II university interface program. *http://www.altera.com/education/univ/research/unv-quip.html*, 2009.

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

[3] Altera, Corp., San Jose, CA. *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, J. Cong, Y. Fan, and J. Xu. Optimality study of resource binding with multi-Vdds. In *IEEE/ACM DAC*, pages 580 – 585, 2006.

[6] J. Cong and W. Jiang. Pattern-based behavior synthesis for FPGA resource reduction. In *ACM/SIGDA Int'l Symp. on FPGAs*, pages 107–116, 2008.

[7] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Zhiru Zhang. High-level synthesis for FPGAs: From prototyping to deployment. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 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] P. Coussy, D. Gajski, M. Meredith, and A. Takach. An introduction to high-level synthesis. *IEEE Design Test of Computers*, 26(4):8 – 17, jul. 2009.

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

[11] Y. Hara, H. Tomiyama, S. Honda, and H. Takada. Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis. *Journal of Information Processing*, 17:242–254, 2009.

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

[13] K. Keutzer. Dagon: technology binding and local optimization by dag matching. In *ACM/IEEE DAC*, pages 341–347, 1987.

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

[15] J. Luu, K. Redmond, W. Lo, P. Chow, L. Lilge, and J. Rose. FPGA-based monte carlo computation of light absorption for photodynamic cancer therapy. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 157–164, 2009.

[16] D. McGrath. Xilinx buys high-level synthesis EDA vendor. In *EETimes*, 2011.

[17] Removed for blind review.

[18] W. Sun, M. Wirthlin, and S. Neuendorffer. FPGA pipeline synthesis design exploration using module selection and resource sharing. *IEEE Tran. on CAD of Integrated Circuits and Sytems*, 26(2):254 – 265, 2007.

[19] United States Bureau of Labor Statistics. *Occupational Outlook Handbook 2010-2011 Edition*, 2010.