2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines

A Grammar Induction Method for Clustering of Operations in Complex FPGA Designs

Muhsen Owaida, Christos D. Antonopoulos, Nikolaos Bellas
Department of Electrical and Computer Engineering,
University of Thessaly, Volos, Greece
Email: {mowaida, cda, nbellas}@inf.uth.gr

***Abstract*—In large-scale datapaths, complex interconnection requirements limit resource utilization and often dominate critical path delay. A variety of scheduling and binding algorithms have been proposed to reduce routing requirements by clustering frequently-used set of operations to avoid longer, inter-operational interconnects. In this paper we introduce a grammar induction approach for datapath synthesis. The proposed approach deals with the problem of routing using information at a higher level of abstraction, even before resource scheduling and binding. It is applied on a given data flow graph (DFG) and builds a compact form of DFG by identifying and exploiting repetitive operations patterns with one or more outputs. Fully placed and routed circuits were successfully generated for complex designs that failed to be placed and routed by the standard manufacturer toolchain without applying our method. Moreover, placement and routing time was accelerated by 16% on average. Our grammar-based approach achieved 12% reduction in area on average, mostly as a result of reducing multiplexer sizes and the number of flip-flops, without noticeable adverse effect on clock frequency. Our comparison with a state of the art algorithm described in [8] shows that our approach outperforms it in both reduction in FPGA area and time to place and route the design.**

Keywords-Automatic synthesis; FPGAs; Grammar-based compression; Routing optimizations.

I. INTRODUCTION

One of the most challenging tasks of FPGA design is achieving fully routed circuits, especially in datapath-dominated designs. Routing resources, in the form of multiplexers and interconnects can dominate both the area and the signal delay for the implementation of computationally intensive algorithms. Moreover, placement and routing (P&R) in modern FPGAs is a very computationally intensive process, potentially taking hours or days, even with the use of state-of-the-art routing algorithms.

Given the routing complexity for large designs, the pressure is growing for techniques that address the placement and routing problem at a higher abstraction level. In a typical high level synthesis approach, the tasks of resource allocation, scheduling and binding are applied on a set of primitive operations such as basic arithmetic and logic operations. The cost of routing resources per primitive functional unit is increasing rapidly in modern FPGAs. For example, the area cost of a 32-bit adder with a 4-input multiplexer on each input port is dominated by multiplexers (67% of the FPGA slices).

By clustering primitive operations (such as additions, shifts, bit selects, etc.) into *macro-instructions,* and implementing those as *Macro FUs (MFUs*), we can effectively reduce the amount of interconnect per operation. Generation of application specific macro-instructions is a common practice among instruction-set extensions designers [1, 2, 3, 7]. Such macro-instructions can substitute a set of primitive operations and consume fewer resources. Regular computation patterns that appear repetitively in a program DFG are strong candidates to be implemented as macro-instructions. As an example, macro-instruction K in Fig. 1.b, which consists of two successive additions, results to a more compact and efficient circuit, requiring fewer resources (i.e. multiplexers) than the individual primitive ADD operations. A macro-instruction can be designed to optimize a set of different criteria, such as silicon real-estate or latency, compared with the set of corresponding primitive operations.

The generation of application specific macro-instructions is a two steps process: a) candidate instructions identification, after a space exploration of the application DFG and b) candidate instructions selection, based on a number of optimality criteria, like latency and area.

In this work we propose the use of a grammar-induction approach for macro-instruction generation and selection. Our target is to exploit the characteristics of macro-instructions, which are in turn implemented as MFUs to reduce datapath complexity, and hence, reduce routing complexity and improve performance. Grammar induction is an established technique used in string and tree compression algorithms [13]. It is a very efficient approach to extract repetitive patterns from a data sequence and to create hierarchical models of such patterns that can be readily understood, analyzed and applied in other domains. In this paper we extend a grammar induction technique called *Sequitur* [13], to identify and generate a set of candidate macro-instructions. Our generated grammar has a regular hierarchal structure with few *non-terminals*, each serving as a potential macro-instruction.

The contributions of our work are the following:

- We introduce an efficient and simple grammar-based technique to identify highly repetitive computational patterns in a DFG. The hierarchal structure of the grammar resembles a multi-granularity computational representation of candidate macro-instructions. Our algorithm is not limited to macro-instructions with just one output, unlike approaches based on PBQP.
- We introduce a set of metrics and cost parameters to

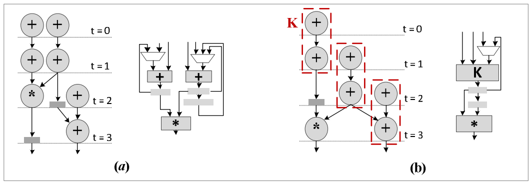


Figure 1: Scheduling and binding of a DFG with: (*a*) Primitive instructions. (b) Mixture of primitive and macro instructions. Macro instruction K is scheduled on the macro FU K which is a pipelined 3-input adder.

978-1-4799-5111-6/14 $31.00 © 2014 IEEE
DOI 10.1109/FCCM.2014.62

estimate the gains expected by the generation of macro-instructions.

- We discuss a systematic approach to pipeline macro functional units and optimize their implementation.

The experimental evaluation proves the efficiency of our approach in significantly reducing the amount of multiplexers in the designs, and hence, reducing routing overhead and area. More importantly, benchmarks with notoriously demanding placement and routing requirements, such as the DVB-S2 telecommunications decoder, were successfully placed and routed only after applying our proposed approach. On average, our proposed techniques achieved 12% area reduction for a series of benchmarks implemented on a Virtex-6 FPGA.

Moreover, we have performed an extensive comparison between our methodology and a state of the art algorithm presented in [8] by Cong *et al.* Our methodology results to both reduced area (by 6%) and lower time to place and route the design (by 7%). These numbers are higher for larger designs that are difficult to implement (Section IV).

The remainder of the paper is organized as follows: Section II introduces the grammar-based macro-instruction identification algorithm. Section III presents the macro-instruction selection and implementation algorithms. In Section IV we discuss the experimental evaluation of the proposed approach. Section V outlines related work, followed by conclusions in Section VI.

II. GRAMMAR GENERATION

In this section we introduce a grammar generation algorithm for systematically discovering repetitive computation patterns inside a DFG, or equivalently identifying candidate sets of operations to be implemented as macro-instructions.

A. *Grammar Representation*

Fig. 2.a depicts a motivating example. A grammar representation consists of a set of statements called *rules* or *non-terminals*. Each rule is a sequence of symbols that contains other rules and/or data symbols called *terminals*. In Fig. 2.a, rule *B* includes both non-terminal *A* and terminals *a* and *d*. The original statement *S* can be restored by substituting each non-terminal with its production, namely the right-hand side of the rule, until all non-terminals are eliminated.

In this paper we extend grammar inductions to also represent DFGs. Fig. 2.b depicts a subgraph of a DFG represented as a compound statement *S*. A simple grammar can be deduced by introducing rule *A*. We treat each primitive instruction as a terminal symbol. A new concern when using grammar representations for DFGs is the operand order for non-commutative operations, such as subtraction or division. We use clock-wise numbering of inputs to state their order. In a DFG that consists merely of primitive instructions, each rule can be considered as a potential compound macro-instruction.

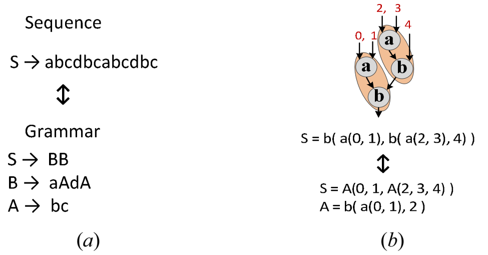


Figure 2: Grammar representation applied on (*a*) a sequence of data symbols, and (b) a data flow graph (DFG).

A convenient property of grammar representations is their hierarchical structure, which inherently integrates multiple levels of granularity. This proves very handy when it comes to hardware implementation of computationally intensive algorithms. For example, assume the DFG subgraph *S* in Fig. 2.b is part of a larger DFG, populated by multiple subgraphs of type *S*. In this case, *S* can function as a non-terminal in the larger DFG. Therefore, the synthesizer has the choice to implement either the macro-instruction *A* that represents a fine granularity computation, or the macro-instruction *S* which represents a coarser granularity computation. An FU that implements a macro-instruction with coarser granularity requires lower routing overhead because most interconnects tend to be within the FU, rather than across the FUs. By reducing inter-FU routing, final datapath implementation tends to suffer less from routing congestion and to require lower P&R overhead. However, a coarser granularity macro-instruction like *S* is not necessarily fitter for implementation. This is, for example, the case when the implementation of *S* requires many resources and at the same time there are just a few occurrences of *S* in the program to exploit these resources. In section III.B we will introduce a systematic method for selecting between different granularity levels.

B. *Generation of Grammar-based DFG Representation*

The grammar generation algorithm traverses the DFG and discovers repetitive patterns by matching pairs of instructions. A pair of instructions *b(a)* denotes that the output of instruction *a* is an operand to instruction *b* as shown in Fig. 2.b. We call instruction *b* destination node and instruction *a* source node.

The rules of a grammar generated according to Sequitur share two properties:

(1) *Digram uniqueness*: A digram is a pair of adjacent symbols, each being a terminal or non-terminal e.g. *aA* in Fig. 2.a. Each digram should appear exactly once in the productions (right-hand side) of the grammar rules.

(2) *Rule utility*: Each rule in the grammar should appear at least twice in the productions of other, higher-level rules. This property ensures that all rules are useful.

In addition to the above constraints we introduce the following constraints, specifically for data flow graphs:

(1) *Output ports number*: The number of outputs of a compound statement described by a rule should not exceed an upper limit N_{out}. This constraint helps reduce the complexity of the selection process by reducing the number of potential patterns.

(2) *Convexity*: A rule is a representation of a convex subgraph in the DFG. A subgraph *S* is convex if there is no path from a node $u \in S$ to a node $v \in S$ through a node $w \notin S$. Convexity ensures that a selected rule can be implemented entirely within an MFU.

(3) *Data computation instructions only*: Load, store, and control instruction nodes are not considered for inclusion in the grammar rules.

Fig. 3 shows the steps of the grammar generation algorithm using an example. The algorithm starts by enumerating the DFG nodes in a reverse topological order (Fig. 3.a). Given the sorted DFG, the algorithm selects the first node, *n0* (destination node) in our example, and builds the template pairs for each operand of the node (*n0(n2)* and *n0(n3)*). If a template pair satisfies the output ports number and convexity tests, the algorithm searches for additional instances of the template in the DFG. The search returns a list of pairs of instructions

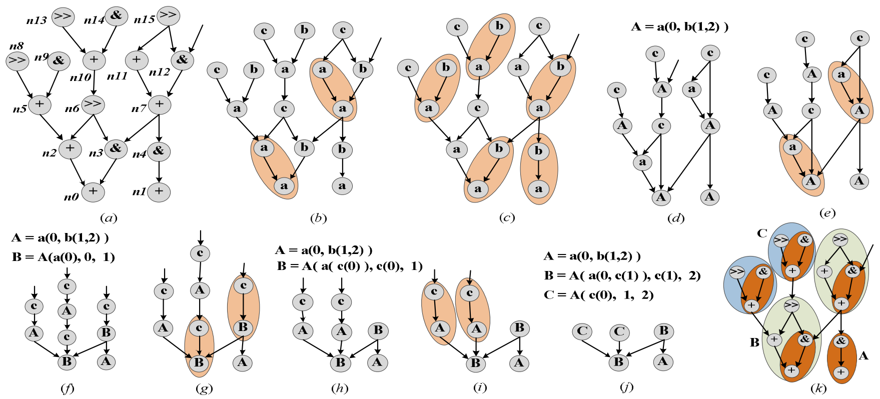


Figure 3: Motivational example showing the steps of the algorithm. In this case output ports number constraint is set to one ($N_{out} = 1$). The final generated grammar is depicted in (*k*).

matching the template pair. A matching instance should have the same operations as the template pair and, generally, the same order of operands. The order of operands is ignored in case the destination node is a commutative operation such as addition. From all the template pairs derived from *n0*, namely *n0(n2)* (Fig. 3.b) and *n0(n3)* (Fig. 3.c), we greedily choose to consider the template pair with the maximum number of instances for implementation as a macro-instruction. The experimental analysis proved that routing complexity and area reduction correlates closely with the number of rule instances. In Fig. 3.d we chose the template pair *a(b)* (corresponding to *n0(n3)*) which has 5 occurrences rather than the template pair *a(a)* (corresponding to *n0(n2)*) which appears only twice.

When a template pair is chosen, the algorithm will update the grammar in one of two ways:

(1) If the destination node in the pair is a *terminal,* i.e. a primitive instruction, the algorithm generates a new rule. In Fig. 3.d we create a new rule *A* for the pair *a(b)* because *a* is a primitive operation.

(2) If the destination node in the pair is *non-terminal* (e.g. node *A* in Fig. 3.e), then;

a. If all its occurrences in the DFG have a matching pair (e.g. *A(a)* in Fig. 3.e), we extend the *non-terminal rule* of the destination node.

b. Otherwise, we create a new rule.

In Fig 3.e, not all the occurrences of the destination node *A* have a matching pair *A(a)* (only 2 of the 5 occurrences of *A* do), so we create the new rule *B*. However, in Fig 3.g, all occurrences of the destination node *B* have a matching pair *B(c)*, so we extend rule *B* to include *c*.

After updating the grammar, the algorithm updates the destination node in each matching pair as follows:

(1) Substitute the destination node of each matching pair by a *non-terminal* node. E.g. node *a* in the pair *a(b)* of Fig. 3.c becomes non-terminal node *A* in Fig. 3.d.

(2) Add the source node in the pair (*b* in the pair *a(b)* of Fig. 3.c) to the internal subgraph of the destination node. Each node marked as *non-terminal* has an internal subgraph which is a cut of the original DFG. In Fig 3.d, non-terminal node *A* corresponds to subgraph *a(0, b(1, 2))*.

(3) Update the operands list of the newly created non-terminal node to include the operands of the source node in the pair, and empties the operands list of the source node.

The process is repeated on the new state of the DFG, searching for templates (pairs of nodes) having the newly inserted non-terminal as destination. In Fig. 3.e, after merging terminal node *a* to non-terminal node *A*, the algorithm repeats the process of building template pairs and searching for matches using destination node *A* which now has two more operands: *c* and *A*, to node *b*. If the algorithm fails to find matching pairs having the newly inserted non-terminal as destination node, it continues with the next node in the sorted DFG list. The iterative process continues until there are no more nodes to consider as destination nodes. It should be noted that our approach is not limited to producing nodes with 1 output, like existing techniques for code generation (PBQP).

III. GRAMMAR-DRIVEN DATAPATH SYNTHESIS FLOW

The hierarchical grammar representation of a DFG can be exploited in many practical problems, such as DFG compression. Since each FU in a datapath can be typically reused for multiple DFG operations, a multiplexer tree is needed at the input ports of each FU to select among a multitude of inputs. Multiplexer trees may cost more in terms of area than the FU itself, especially for simpler FUs. If all instances of a grammar rule are implemented as a macro functional unit (MFU), where the internal data flows are free of multiplexers, the area gain may be significant; furthermore, reducing routing complexity leads to reducing routing latency, and time it takes to place and route the design.

Fig. 4 shows the complete grammar-driven datapath synthesis flow. For each input DFG we generate the datapath RTL that implements the DFG functionality. Given the original input DFG, the synthesis flow starts by slicing the DFG into one or more smaller subgraphs. Then, the grammar generation engine processes each DFG slice separately. A subset of the non-terminal rules is selected to generate macro-instructions. Given the selected set of rules, the algorithm will produce a new DFG, incorporating primitive and macro-instructions.

A preliminary step before grammar generation in our tool is the slicing of the given DFG into smaller DFGs. In some cases, for example when the DFG expresses computation of an unrolled, data-parallel loop, the graph consists of multiple

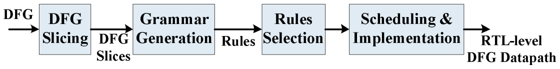


Figure 4: Grammar-based datapath synthesis flow.

strongly connected subgraphs (slices). Different slices can be treated independently in grammar generation, scheduling and binding. For grammar generation, the search space for matching pairs is smaller, which speeds up grammar generation. Another important benefit is the implicit creation of isolated islands of resources (FUs, registers) for each DFG slice, making the task of the placement & routing much easier.

A. Grammar Generation & Selection

Following DFG slicing, the flow continues with the grammar generation algorithm described in Section II, which is applied independently on each slice. Hence, each DFG slice will end up with its own grammar representation.

Grammar-driven data compression algorithms normally use all the grammar rules to compress a sequence of data symbols. In our case, only a subset of rules will be used to implement MFUs. The purpose of this step is to identify a subset of grammar rules that minimizes routing density and reduces total area. The greedy selection algorithm we introduce in this paper uses a fitness function to assign weights to each rule in the generated grammar. At each step, the rule with the highest fitness value is selected to be implemented as an MFU and all its instances are removed from the grammar. When a rule is selected, all grammar rules using this rule as a non-terminal in their productions are essentially also removed from the grammar and they are no longer considered for implementation as MFUs. Otherwise, multiple different MFUs would be generated, executing the same primitive operations. After each step, the fitness function updates the fitness of the remaining rules. The process is repeated until the grammar is empty.

The fitness function (1) uses a set of metrics to estimate the gain from implementing rule *i* as an MFU:

$$W_i = CG_i * \left(LG_i + MUXG_i\right) \quad (1)$$

The following paragraphs explain the parameters of (1).

Coverage Gain (CG): The coverage gain for rule *i* is a normalized value of the total number of primitive instructions in the DFG covered by the specific rule. The metric is computed in (2). Higher coverage means fewer primitive FUs will be implemented individually, hence, smaller multiplexer trees. To compute a fair metric value, we compute the total number of primitive instructions that can be covered by a given rule, instead of relying only on the count of rule instances (*occurrences*) or the number of primitive instructions (*operations*) per rule instance.

$$\begin{aligned} Coverage_i &= \left(Occurrences_i * \#Operations_i\right), \\ CG_i &= \frac{Coverage_i}{\max\limits_{0 \le i \le RulesCount}\left(Coverage_i\right)} \end{aligned} \quad (2)$$

The coverage gain factor functions as a multiplier for two metrics, *LG* and *MUXG* that correspond to area gains. The value of the coverage gain metric will change each time we select a rule to be implemented as an MFU. This happens because some rule instances are removed from the grammar if they appear as non-terminals in the production of a rule selected earlier. The maximum coverage value will also change, and hence, the normalized values of CG.

Multiplexers Gain (MUXG): This metric quantifies area gains due to reduction of the number of multiplexers per instance of each *rule*. The metric is computed using (3). The numerator is the difference between the total number of inputs of all primitive FUs of an MFU (Σ#*Operands*) and the number of the MFU inputs (#*RuleOperands*). To normalize, we divide by "Σ#*Operands*".

$$\begin{aligned} Ratio_i &= \frac{\sum\limits_{p=0}^{RuleOps} Operands_p - RuleOperands}{\sum\limits_{p=0}^{RuleOps} Operands_p} \quad (3) \\ MUXG_i &= \frac{Ratio_i}{\max\limits_{0 \le p \le RulesCount}\left(Ratio_p\right)} \end{aligned}$$

Based on formula (3), we observe that the value of *MUXG* tends to increase when the number of primitive instructions in a rule increases. In other words, larger rules will have higher multiplexer gain. However, the algorithm does not always favor larger rules over smaller ones. A smaller rule with lower multiplexers gain per instance may be associated with a much higher coverage gain, which makes it fitter for implementation.

Logic Gain (LG$_i$): This metric quantifies the potential for reduction of logic cells through packing of primitive instructions within an MFU (or equivalently a grammar rule). The metric is computed using equation (4). The numerator quantifies the efficacy of fusing the logic cells of all the primitive FUs of the MFU. LUTs in FPGAs have a limited number of inputs. The more the number of MFU inputs increases the more difficult it becomes to map its function on fewer LUTs, and therefore, we divide by the number of the MFU input signals (#*RuleOperands*) in equation (4).

$$\begin{aligned} LogicGain_i &= \frac{\sum\limits_{l=0}^{RuleOps}\left(1 - A_l\right)}{RuleOperands} \quad (4) \\ LG_i &= \frac{LogicGain_i}{\max\limits_{0 \le p \le RulesCount}\left(LogicGain_p\right)} \end{aligned}$$

The value of the parameter A_l in (4) is normalized in the range [0, 1] and is characteristic for each primitive instruction type *l*. It quantifies the difficulty to fuse this instruction with additional ones, in the same set of logic cells. A_l is dependent on the nature of the instruction in the FPGA architecture, and the synthesis, placement and routing toolchain. The following paragraphs present a brief explanation of the estimation of A_l.

A_l Parameter Estimation: We developed a set of representative subgraphs, with various primitive instruction types and configurations, to be used as micro-benchmarks for systematically off-line estimation of A_l on each target platform. Although the methodology is the same, the values of the parameters are FPGA-device specific. A_l quantifies an estimation of the percentage of the implementation capacity of the LUT taken by the primitive instruction *l*. Similarly, if two primitive instructions *l* and *l′* are fused on the same LUT, the summation of the corresponding area estimation parameters A_l *and* $A_{l'}$ provides a good estimation of the consumption of the LUT implementation capacity by both instructions.

An initial estimate of A_l is computed by finding how many primitive FUs of the same type *l* can be packed in one level of logic of the same LUTs. We perform synthesis, placement and routing on the given FU (or MFU) to determine the number of consumed LUTs. The process of adding more FUs of the same type continues, until the resulting subgraph requires more LUTs for its implementation. For example two adders can be packed in the same number of LUTs required for the implementation of one adder of the same bitwidth; if a third adder is added, it will occupy a different set of LUTs. Therefore, the initial estimate of A_{add} takes the value 0.5. If

	CG	*MUXG*	*LG*	*W*
A	1	0.5	0.95	**1.45**
B	**0.8**	**1**	**1**	**1.6**
C	0.6	0.67	0.72	**0.83**

	CG	*MUXG*	*LG*	*W*
A	1	0.75	1	**1.75**
C	**1**	**1**	**0.75**	**1.75**

Select Rule **B** → Update Metrics → Select Rule **C** →

Figure 5: The selection process of rules for the grammar of Fig 3. The selected set of rules is: {B, C}. Parameter A_l is 0.2 for logic operations, 0.5 for add/sub, and 1.0 for multiplications, divisions and FP operations in a Virtex-6 architecture. These values are higher for the older Virtex-4 architecture.

packing a third adder on the same set of LUTs succeeded, the estimate would be 0.33.

Given the initial estimates of parameters A_l, the heuristic performs a refinement step which tries to reduce the error in the initial estimate. The second step refines the parameter A_l for primitive operation of type l by computing how many primitive operations of type k, with $A_k < A_l$, can be packed in the same LUTs already occupied by operation l.

Rule Selection Example: Fig. 5 shows how we apply rule selection on the grammar of the example of Fig. 3. The left table of Fig. 5 contains the normalized metric parameters and the corresponding fitness for each rule according to (1). After selecting the rule with the maximum fitness (*B* in Fig. 3), we update the metric parameters, and normalize their values again. Note that after removing rule *B* from the grammar, we also removed two instances of rule *A*, which appears now in only 3 instances. Rules *A* and *C* now have the same coverage since they both cover 6 instructions. After updating the metrics, rules *A* and *C* have both the same weight. Since rule *C* is using rule *A*, *OrderRules* subroutine prioritizes rule *C* over rule *A*, and hence the algorithm selects rule *C* for implementation and removes 2 more instances of the rule *A*. Since rule *A* now appears in only one instance, we can no longer consider it for MFU implementation, because of the *rule utility* constraint: rules must appear in the grammar with at least two instances.

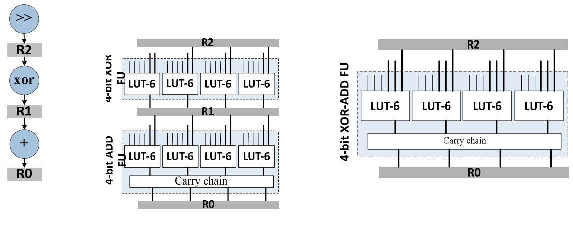
B. Macro Functional Unit Pipelining

Once the set of rules have been selected for MFU implementation, we have to determine the pipeline depth of each MFU and therefore its latency.

The algorithm uses a default pipelining scheme for inserting pipeline registers in MFU as a reference. The default scheme greedily adds a pipeline register after each primitive FU (Fig 6.a). In this reference pipeline scheme, the combinational path of a single primitive FU (4-bit XOR and 4-bit ADD FUs in Fig 6.b) is considered as one level of logic. Hence, using the default pipelining scheme, only one level of logic exists between two successive pipeline registers.

The algorithm traverses the MFU subgraph and removes a pipeline register if its removal does not increase the levels of logic cells between two other pipeline registers. For example, in Fig 6.a, pipeline register **R1** will be removed if it does not increase the levels of logic between registers **R0** and **R2**. In Fig 6.c, the removal of register **R1** allowed fusing the logic cells of the XOR FU with the logic cells of the ADD FU. The removal of **R1** produces a new Boolean expression that may be implementable using one level of logic cells (LUTs).

To decide if the removal of a pipeline register will increase the number of logic levels – in the form of LUTs – or not, the algorithm uses the same set of A_l parameters used in (4) to compute the logic gain metric LG_i. In general, if the summation of area estimation parameters A_l in a DFG sub-path, is less than or equal to 1.0, we estimate that the corresponding primitive instructions can be fused and implemented on a single LUT, or equivalently, they require the same levels of logic as one primitive instruction. As a result, intermediate registers in the sub-path can be safely removed without affecting the timing characteristics of the circuit.



(*a*) (*b*) (*c*)

Figure 6: (*a*) Reference pipeline scheme used as template for the pipelining algorithm. (b) Logic level of pipelined Xor and Add operators. (c) Fused Xor and Add operations in a single logic level.

C. Scheduling and Implementation

Once a set of rules is selected for MFU implementation, each instance of a rule is converted to a macro instruction of the specific type and the resulting DFG is scheduled using modulo scheduling. A macro instruction is scheduled only when all input data are available. We use *Swing Modulo Scheduling (SMS)* [12] to generate a schedule of the DFG nodes on the allocated resources. The scheduler identifies an iterative pattern of instructions and their assignment to functional units (FUs), so that each iteration can be initiated before the previous ones terminate. SMS creates software pipelines under the criterion of minimizing the Initiation Interval (II). The latter is the main factor affecting computational throughput. The value of II represents the rate at which we initiate successive loop iterations. By increasing the amount of allocated resources we lower the value of II, unless a recurrence circuit in the loop prevents that. For II = 1, a separate FU is allocated per instruction. Target IIs > 1 are particularly useful in certain domains, where designs are subject to strict resource constraints.

IV. EXPERIMENTAL EVALUATION

A. Methodology

The tool flow of Fig. 4 is used to produce macro-instructions, and the synthesis engine performs resource allocation, modulo scheduling and binding on the new DFGs. Finally, the tool generates synthesizable Verilog of the DFG datapath. We used Xilinx ISE 12.4 for synthesis, placement & routing, targeting the Virtex-6 LX760 FPGA. Our benchmark base consists of the 8 kernels outlined in Table I. The benchmarks are computation-intensive and their DFG sizes vary from tens up to thousands of primitive instructions.

TABLE I: APPLICATIONS USED FOR EVALUATION

Application	***Description***
CMC	AVS Video Decoder Chroma motion interpolation
Luma	AVS Video Decoder Luma motion interpolation
DCT	H.264 Video Encoder 8x8 Integer DCT
SEAL	Seal cryptography kernel
CN	Forward Error Correction (FEC) decoder CheckNode Kernel
BN	Forward Error Correction (FEC) decoder BitNode Kernel
Deblocking	AVS Video Decoder Deblocking Filter
LUD	LU Decomposition Perimeter (Floating Point)

TABLE II: GRAMMAR BASED DFG REPRESENTATION RESULTS. "*INSTS(G)*" REFERS TO THE NUMBER OF DFG NODES AFTER GRAMMAR BASED REPRESENTATION.

App.	*#Rules*	*#Selected Rules*	*Rule Size*	*#Instances per Rule*	*#Insts*	*#Insts(g)*	*Reduct.*
CMC	6	3	[2-9]	7	136	86	37%
LUMA	18	11	[2-4]	6	299	219	27%
DCT	10	8	[2-3]	9	307	197	36%
SEAL	8	5	[2-3]	6	143	107	25%
CN	18	7	[2-5]	127	3962	2500	37%
BN	8	5	[2-7]	64	2917	1677	43%
Deblocking	9	5	[2-4]	3	176	150	15%
LUD	1	1	[2]	2	20	18	10%

To explore the efficacy of the proposed grammar driven synthesis approach, we experimented with a variety of datapath configurations. A datapath configuration depends on the number, type and bitwidth of allocated functional units, and memory I/O bandwidth. Each configuration is characterized by the Initiation Interval (II), as explained in Section III.C. We experimented with three configurations: C_A targeting II = 1 in the *original* case, C_B with II = 8, and C_C with II = 16. The only exception was *SEAL*, which contains complex cross-iteration loop dependences that create recurrence scheduling constraints. This constraint limits modulo scheduling ability to a minimum II value of 60 cycles. The value of II relates with the size of the multiplexer tree at FU input ports as well. Schedules with higher II result into larger multiplexer trees due to FU reuse.

B. Grammar Representation Results

Table II summarizes some low-level characteristics of the DFGs after grammar generation and rule selection. Column "*#Rules*" lists the grammar size in numbers of rules generated for each application. Column "*#Used Rules*" lists the number of selected rules from each grammar to be implemented as MFUs in the final representation of the DFG. Column "*Rule Size*" shows the range of number of instructions per rule for the selected rules subset. "*#Instance per Rule*" shows the average number of instances per rule. Columns "*#Insts*" and "*#Insts(g)*" list the DFG size before and after grammar-based representation, respectively. Finally, "*Reduction*" shows the percentage of reduction in the number of primitive instructions.

Unlike pattern recognition and enumeration approaches, the generated set of subgraphs (i.e. *rules*) is much smaller in both the total number of subgraphs and subgraph size, yet it covers 40% – 53% of the program DFG.

C. FPGA Placement & Routing Results

Fig. 7 shows the area results for all 8 benchmarks, for the *original* and the *optimized* cases. On average, the grammar-based approach achieved very good results even for kernels with few instructions, delivering an average 12% area reduction in terms of number of slices and number of flip-flops (FFs are not shown in Fig. 7 for brevity). Reduced DFG size with grammar-based compression required around 16% less time on average to schedule and synthesize, which correlates with the reduction in DFG size (Fig. 8). A noticeable result appears for CN and BN kernels. These two kernels are very large (approximately 4000 and 3000 nodes, respectively) and are notoriously difficult to place and route. Without the grammar-driven synthesis approach, the vendor-provided synthesis tool failed to successfully finish placement and routing even in large FPGAs such as LX760. After the grammar-driven synthesis optimizations the tool generated a fully placed and routed design in less than 3 hours (Fig. 8).

To gain a better understanding of the effects of our methodology on lower-level FPGA components, we decided to take a closer look at the type and number of multiplexers, since this is one of the main causes of routing congestion. We considered the DCT benchmark for C_B configuration, for which our approach reduces number of slices by 32%. After applying our methodology the number of MUXF7 components, used to implement 8:1 (or wider) multiplexers in Virtex-6, dropped from 686 to 164 (4.18x), due to the reduction of the number of input multiplexers in each FU. On the other hand, the number of MUXCY components used in carry-propagation paths increased from 983 to 1605 (1.63x). By combining multiple operations (e.g. additions) in MFUs, we effectively increase the number of fast carry logic paths in the FPGA.

Xilinx toolset also instantiates DSP hard blocks to implement multiplications and in some cases, the number of DSP blocks in the optimized RTL increases. For example, the Luma benchmark requires 9 DSP blocks for C_B configuration in the optimized RTL up from 6 blocks in the unoptimized RTL. Such increases have marginal effect on the routability and performance of the final design, since it is always the LUTs and flip-flops that dominate area requirements and drive routing congestion.

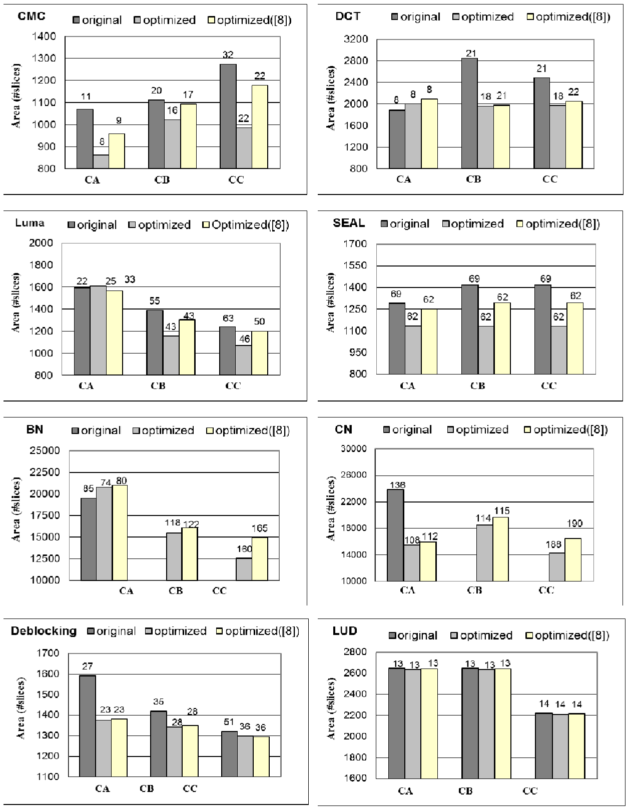


Figure 7: Number of slices (lower is better) for original and optimized configurations (with grammar-driven datapath synthesis). In CN and BN benchmarks the missing configurations for the original case are due to the fact that the standard manufacturer toolchain failed to fully place & route the generated RTL before applying our compression technique. The last bar shows the number of slices according to [8]. The numbers above the bars represent the schedule latency (in clock cycles) of a single loop iteration.

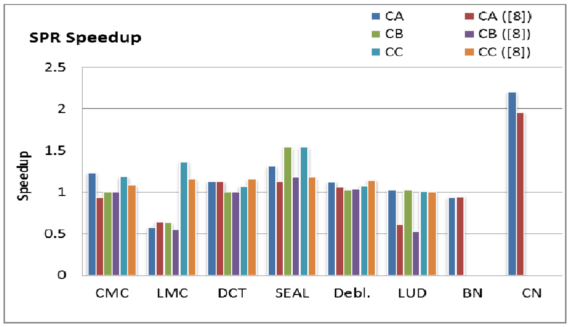


Figure 8: Synthesis, Placement & Routing (SPR) speedup of designs produced by applying our algorithm and the algorithm described in [8] with respect to unoptimized designs. C_B and C_C bars in BN and CN kernels are missing because the original designs failed to finish placement and routing successfully after 12 hours of runtime, while optimized designs succeeded within 3 hours.

Grammar-based designs typically involve more FU types than original designs in their datapath, due to the introduction of MFUs. The additional MFU types impose an area overhead. The issue manifests more clearly in the C_C configurations, where few FU instances (normally one or two) are allocated for each FU type. For DCT and Luma, our algorithm achieves 32% and 17% reduction in area respectively for the C_B configuration. For the C_C configuration area gains are limited to 20% and 13%. These two kernels use 8 and 11 MFU types respectively in their datapath. While using MFUs reduces multiplexer area, the area overhead from the large number of used rules limits the overall area reduction for configuration C_C. On the other hand, CMC and SEAL kernels use only 3 and 5 rules respectively, with limited area overhead, hence configuration C_C outperforms configuration C_B. Note also that MFU area overhead can be reduced whenever the pipeline algorithm identifies opportunities to produce compact and lightweight MFUs, which is the case for CMC and SEAL. On the contrary, MFUs in DCT and Luma datapaths consist of heavyweight primitive FUs, that could not be effectively fused.

DFGs characterized by patterns with a very low number of occurrences and low DFG coverage are also potentially susceptible to area overheads from the introduction of MFUs. In this case, the combination of MFUs overhead with the limited multiplexer reduction might produce designs with very little or no area reduction, which is the case for LUD and Deblocking kernels. However, during our experimental evaluation with a variety of kernels we observed that, even for DFGs with a small number of pattern repetitions (see Table II, #Instances per Rule), area reductions are achieved because these repetitions cover 45% to 53% of the DFG. Therefore, instruction clustering led to a significant reduction in the area spent for multiplexers, overweighing the MFUs area overhead.

It appears from the experimental evaluation that the grammar-based approach sometimes performs poorly at II = 1. This is expected, because in this case there are no multiplexers to optimize out. For some benchmarks (DCT and Luma) the consumed area is slightly more than that of the original configurations. For these benchmarks, the pipeline algorithm produced fully pipelined MFUs, because they contained heavyweight primitive FUs that could not be fused with others. Moreover, using macro-instructions in those benchmarks increased variable lifetimes, which led to allocating more registers. This is, for example, the case for the BN kernel (configuration C_A). The version produced after instruction clustering requires more area than the original one, despite the fact that the pipelining algorithm efficiently produced more compact MFUs. Most of the generated MFUs in BN kernel are not flat. They have latencies between 3 and 4 cycles (after being optimized down from 7 cycles by the pipelining algorithm). The large amount of MFUs with such latencies imposed an overhead on the scheduler, leading to increased variable lifetimes and registers requirements.

On the other hand, the proposed approach worked well even at II = 1 for benchmarks such as CMC, Deblocking and CN, in which the logic gain for generated macro-instructions was significant. The MFUs produced were compact and lightweight, which subsequently led to the area reduction. Compact MFUs generated after the pipelining algorithm have a positive impact on variables lifetime at II = 1, leading to reductions in registers requirements. Therefore, at II = 1, area reductions are obtained mainly by compressing and optimizing the generated MFUs.

We have implemented a state of the art algorithm described by Cong *et al.* [8] and have integrated it in our synthesis flow to compare the two methodologies (Fig. 7, third set of bars). Our approach produces designs with smaller number of slices than [8] by 6%, on average, and over 9% for higher II values. Our algorithm performs better than [8], especially in BN, CN, Luma, and SEAL kernels because we use fewer patterns to represent the DFG, despite covering nearly the same number of instructions. For example, we select 5 and 7 rules for BN and CN, while [8] selects 8 and 11 respectively. Both algorithms cover approximately the same number of primitive FUs (43% and 37% for BN and CN, respectively). In CMC, both algorithms used the same patterns to implement the datapath, however our MFU packing algorithm produces more compact MFUs with smaller area. For DCT, LUD and Deblocking kernels, our algorithm and Cong's approach produced almost the same patterns which were also implemented in a similar manner, hence the area reduction by both approaches is similar.

We should note that our approach performs better than [8] at higher II. This correlates with the fact that multiplexers reduction increases in proportion to II, meaning that our approach is more successful in reducing the number of multiplexers.

Fig. 8 shows the synthesis, placement & routing (SPR) speedup achieved on the standard Xilinx toolset for the optimized versus the original DFGs. SPR for grammar-based designs is on average faster than for the original designs, achieving an average speedup 1.16x for C_C configuration without taking into account CN and BN kernels. Original CN and BN kernels in C_B and C_C configurations were processed for over 12 hours before eventually failing to produce a bitstream due to routing congestion. The DFGs produced for the same benchmarks and configurations by the grammar-based approach succeeded in less than 3 hours. CN kernel achieves the highest speedup (2.2x) among the other benchmarks, mainly because of the significant area reduction attained by the optimized design. Our algorithm also resulted into faster SPR time over [8] by 7% on average for all configurations. In other words, our algorithm creates designs that occupy less area at a smaller amount of SPR overhead compared to [8].

The achieved clock frequency for optimized configurations has a deviation between +8% (frequency increase) to -1.2% from the original configurations with an average +1.6% (excluding benchmarks in which P&R on the original failed).

The proposed grammar-based algorithm is very fast; in all cases the grammar generation and rules selection took less than a second to finish and produce a new DFG.

V. RELATED WORK

Extracting regular computation patterns has been the focus of prior research in behavioral datapath synthesis [1-9]. In [10] the authors provide an extensive overview of the research in instruction-set extension. Regularity extraction has also been used for custom instruction generation [5,7,8]. The proposed approaches can be categorized based on how they solve the two sub-problems: candidate subgraph generation and selection.

Candidate subgraph generation. Early work used variations of enumeration techniques augmented with a set of constraints or a guide function to prune the search space [1,7]. Another set of early works used pattern recognition techniques to extract computations regularities in a DFG. A set of works [5,8,9] uses user-defined patterns libraries and patterns matching techniques to improve the quality of logical synthesis at the behavioral level.

Cong *et al*. proposed a pattern-recognition based approach for FPGA resources reduction [8], in which – contrary to our approach – pattern instances do not need to be totally identical. Therefore, MFUs can have extra multiplexers on intra-FU interconnects, increasing the area overhead of MFUs. ***Candidate subgraph selection***. All aforementioned papers approached the candidate subgraph selection problem in a similar manner: a cost function and a set of metrics have been used to weigh the performance gain and the feasibility of a candidate subgraph. In previous research that targeted application specific processors and instructions set extensions [1,2,7], where the concern is increasing processors performance, metrics that estimate latency, area, and inputs/outputs number have be used.

Cong [8] used metrics that estimate multiplexers cost reduction and latency. The latency metric gives higher priority to flat subgraphs. In our case latency is not the primary concern. The critical path latency is actually effectively reduced during MFU pipelining. However, using the flatness metric of Cong could reduce the variables lifetime overhead which appears in the BN kernel at configuration C_A (Fig. 7).

Prior work addressed the problem of multiplexer size reduction. The majority of works are based on resources binding techniques in datapath synthesis [4,11]. The drawback of previous binding algorithms is that they fail to exploit regular patterns and rely solely on iterative algorithms to minimize the multiplexers overhead during resources binding.

VI. CONCLUSION

In this paper we introduced a grammar-based datapath synthesis methodology. Our approach targets the reduction of the routing complexity and overhead in FPGA designs. The core of the methodology is the production of a hierarchical grammar representation of a DFG. The rules of the grammar correspond to cuts of the DFG which can be considered as candidate macro-instructions. The algorithm performs grammar generation, rules selection and implementation with very small computational complexity. Furthermore, we presented a simple yet systematic area estimation technique, which can be applied to characterize each target FPGA architecture and toolchain. The results of the area estimation are used to both guide the rules selection phase, and drive the insertion of pipeline registers in the produced macro FUs.

Experiments showed the efficiency of the proposed approach in reducing routing complexity and hence reducing area. Moreover, the pipelining algorithm typically produced schedules with smaller latency and no penalty on clock frequency. Most importantly the grammar-driven optimization allowed successful placement and routing on complex designs that were not deemed implementable before.

ACKNOWLEDGMENT

This work was supported by EU (European Social Fund ESF) and Greek funds through the operational program Education and Lifelong Learning of the National Strategic Reference Framework (NSRF) - Research Funding Program: THALIS.

REFERENCES

[1] ATASU, K. POZZI, L. AND IENNE, P.2003. Automatic application-specific instruction-set extensions under micro-architectural constraints. In *Proceedings of the 40th annual Design Automation Conference (DAC '03).* San Francisco, CA, 256–261.

[2] BRINGMANN, O. AND ROSENSTIEL, W. 1997. Resource Sharing in Hierarchal Synthesis. In *Proceedings of the IEEE/ACM Conference on Computer Aided Design (ICCAD'97)*. San Jose, CA, 318-325.

[3] BRISK, P. KAPLAN, A. KASTNER, R. AND SARRAFZADEH M. 2002. Instruction Generation and Regularity Extraction for Reconfigurable processors. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '02)*. Grenoble, France, 262 - 269.

[4] CHEN, D. AND CONG, J. 2003. Low-Power High-Level Synthesis for FPGA Architectures. In *Proceedings of the International Symposium On Low Power Electronics and Design (ISLPED'03)*. ACM, Seoul, Korea, 134 – 139.

[5] CHOWDHARY, A. KALE, S. SARIPELLA, P. SEHGAL, N. AND GUPTA, R.. Extraction of Functional Regularity in Datapath Circuits. *IEEE Transactions On Computer Aided Design of Integrated Circuits and Systems*. vol. 18, pg. 1279-1296, Sept. 1999

[6] CHRIS, L. AND VIKRAM, A. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '04)*. IEEE, Palo Alto, CA, 75-86.

[7] CLARK, N. ZHONG, H. AND MAHLKE, S. Automatic custom instruction generation for domain-specific processor acceleration. *IEEE Transactions on Computers*, vol. 54, issue 10, pg. 1258 – 1279, Oct. 2005

[8] CONG, J. AND JIANG, W. 2008. Pattern-based behavior synthesis for FPGA resources reduction. In *Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays (FPGA '08)*. ACM, Monterey, CA, 107 - 116.

[9] CORAZAO, M. KHALAF, M. GUERRA, L. POTKONJAK, M. AND RABAEY, M. 1996. Performance Optimization Using Template Mapping for Datapath-Intensive High Level Synthesis. *IEEE Transc. On Computer Aided Design of Integrated Circuits and Systems*. vol. 15, issue 8, pg. 877-888.

[10] GALUZZI, C. AND BERTELS, K. 2011. The instruction-set extension problem: A survey. In *ACM Transactions on Reconfigurable Technology and Systems*, vol. 4, issue 2, pg. 18:1-18:28

[11] HUANG, C. Y. CHEN, Y. S. LIN, Y. L. AND HSU, Y. C. 1990. Data Path Allocation Based on Bipartite Weighted Matching. In *Proceedings of the 27th annual ACM/IEEE Design Automation Conference (DAC '90)*. ACM, San Orlando, FL, 499 – 504.

[12] JOSEP, L. ANTONIO, G. EDUARD, A. AND MATEO, V. 1996. Swing Modulo Scheduling: A Lifetime-Sensitive Approach. In *Proceedings of the Conference on Parallel Architectures and Compilation Techniques (PACT '96)*. IEEE, Boston, MA, 80-86.

[13] MANNING, N. WITTEN, H. AND MAULSBY, L. 1994. Compression by Induction of Hierarchical Grammars. In *Proceedings of Data Compression Conference (DCC '94)*. Snowbird, UT, 244-253.