Towards Trainable Synthesis for Optimized Circuit Deployment on FPGA

Jean-Philippe Legault[1], Panagiotis Patros[1,2], and Kenneth B. Kent[1]

[1] Faculty of Computer Science, University of New Brunswick, Fredericton, New Brunswick, Canada
Email: {jlegault, patros.panos, ken}@usc.edu
[2] Department of Computer Science, University of Waikato, Hamilton, Waikato, New Zealand
Email: ppatros@waikato.ac.nz

***Abstract*—Field Programmable Gate Arrays (FPGAs) utilize multiple programmable elements and non-programmable blocks. After synthesizing an input Hardware Design Language (HDL) design into a circuit, optimizations are used to discover a satisfactory deployment on a target FPGA. HDLs' compound operations, such as addition, can be implemented in various ways and thus, multiple but functionally equivalent circuits can be synthesized. To leverage this, we propose a methodology that first enables configurable synthesis of compound operations. Second, it trains the system using a set of HDL files and architectures to optimize target performance objectives, such as critical path length and power. We prototyped our technique in the open source Verilog-To-Routing (VTR) tool. We subsequently produced two configuration files targeting different deployment objectives; experimental results with the VTR Verilog benchmarks revealed significant improvements.**

***Index Terms*—FPGA, HDL, compound arithmetic operators, reconfigurable synthesis, Verilog-To-Routing**

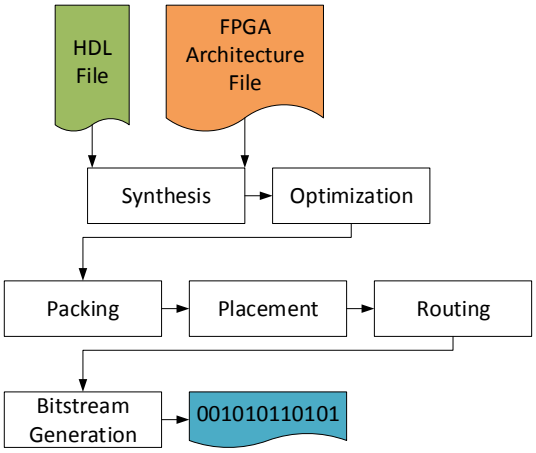

Fig. 1. FPGA CAD Flow

I. Introduction

An FPGA is an integrated circuit that can be programmed to emulate other circuits. FPGAs interface through programmable interconnect pins and maintain a large number of programmable logic blocks and programmable routing elements. Hard blocks that execute specific functions, such as multipliers and memories, are also embedded within the FPGA fabric for improved performance.

FPGAs are programmed using an HDL. Afterwards, an FPGA Computer Aided Design (CAD) flow is used to synthesize and optimize the design before packing, placing and routing it. A bitstream containing the instructions of the programmable elements can be delivered to be flashed onto the FPGA (Figure 1).

FPGA CAD synthesis modules can map an HDL operation to a hard-block or synthesize it into soft-logic. Furthermore, CAD tools have to select a design when synthesizing compound HDL operations, such as addition, multiplication and division. For example, the addition operation can be implemented with a ripple carry adder, or a carry select adder, or even, a hard adder available on the target FPGA architecture. However, a blanket, "fits-all" solution might not be ideal for specific circuits and/or architectures. Furthermore, it might be more beneficial, depending on the target design goals (e.g., space time tradeoffs), to split the implementation of a compound operation into a variety of sub-implementations and/or map it to one or multiple hard blocks. Overall, in this paper, we make the following contributions:

- We highlight the differences between simple and compound HDL operations.
- We propose a configurable and trainable synthesis technique that leverages the open-ended specifications of compound HDL operations.
- We formalize the problem of selecting a satisfactory compound synthesis configuration based on performance metrics (e.g., critical path length and FPGA utilization) as an optimization problem that can be efficiently solved by existing techniques, such as hill climbing.
- We implement a prototype of our technique on the open source Verilog-To-Routing (VTR) FPGA CAD toolchain.
- We evaluate our technique using five search sets targeting different performance metrics. Measurements revealed significant performance improvements.

II. Background

Verilog is an HDL used to design, model and validate electronic systems [1]. Verilog-to-Routing (VTR) is set of CAD tools that perform synthesis, optimization, verification,

978-1-5386-7557-1/18/$31.00

packing, placement and routing of circuits on customized FPGA architectures. They take as an input a Verilog file and an architecture description file; the final output is comprised of a routed, packed and placed netlist for the given circuit, in addition to a number of performance metrics, such as FPGA utilization, power requirements and critical path length [2].

The first tool in the VTR flow is Odin II, which is responsible for compiling/synthesizing the input Verilog file into a netlist. Odin II takes into consideration the target FPGA architecture and aims to map Verilog operations onto hard blocks before converting the rest as soft logic [3]. The netlist output of Odin II is then handed off to ABC, which is responsible for logic reduction, replacement and optimization. Finally, the resulting circuit is passed to the Versatile Placement Routing (VPR) tool, which is responsible for packing, placing and routing the circuit onto the target architecture as well as reporting the final performance metrics [4].

III. Related Work

Design space exploration is utilized in embedded systems to discover the best solution out of a number of alternatives such as the number of processors and the type of the interconnection network [5]. Instead, we propose an efficient methodology for first enabling design space configuration of compound HDL operators and second, trainable design space exploration per FPGA circuit and custom architecture based on user-defined weights for performance metrics.

Compound HDL operations could be replaced with simple operations to unambiguously designate a desired implementation per FPGA. However, defining each compound operation as a discrete design is error-prone, inconvenient and would require maintaining multiple iterations of the same circuit. Instead, we propose that HDL files remain intact and that configuration files instruct the layout of compound operations [6].

Netlist optimizers, such as ABC, perform various operations that can improve the sequential performance and size of a design via redundant logic removal and grouping [7]. Nevertheless, optimizations at this stage do not leverage the actual HDL code nor do they offer circuit parallelism.

HDL synthesizers, such as Odin II, have been used in various research projects that aimed in producing or facilitating optimized designs such as: netlist and hard-block reductions [8] and reset subcircuit elision [9]. We expand Odin II to synthesize heterogeneous sub-structures of compound HDL operations via targeted search.

In this work, we prototype our technique focusing on the adder compound operator. Luu et al. explored using hardened adders in VTR to improve efficiency of arithmetic operations as they are a rather small and prevalent circuit [10]. In that work, the authors measured an overall performance improvement of 15%. That study outlined that improved logic synthesis should result in higher gains and it is one of the driving factors of our work. The baseline the authors used is a soft-logic ripple carry adder and an optimized baseline should be an interesting comparison. Additionally, the authors used a "fits all" approach to place their adder hard-blocks. Thus, augmenting the technique to be tailored for specific circuits can improve the performance. However, due to the large degree of freedom in both the adder design and its context, embedding this information within the synthesizer is impractical.

TABLE I
Simple vs Compound Verilog Operators

Operators	Source	Type
Concatenation	`{a, b}`	Simple
Replication	`{a, {b}}`	Simple
Arithmetic	`a+b, a-b, a*b, a/b, a%b, a**b`	Compound
Relational	`a>b, a>=b, a<b, a<=b`	Simple
Logical	`!a, a&&b, a||b, a!=b, a==b, a!==b, a===b`	Simple
Bitwise	`~a, a&b, a|b, a^~b`	Simple
Reduction	`&a, ~&a, |a, ~|a, ^a, ^~a`	Simple
Shifts	`a<<b, a<<<b, a>>b, a>>>b`	Simple if `b` const Else, compound
Conditional	`a?b:c`	Simple
Vector	`a[b:c]`	Compound
Array	`[b:c]a`	Compound
Bit-select	`a[b]`	Simple

IV. Configurable Compound Synthesis

To improve the circuits performance on FPGAs as well as circumvent the limitations of traditional optimizers, we implement a configurable synthesis methodology per implicit logic subcircuit and bit width. The technique's guidelines are passed to the synthesizer in the form of a configuration file—alongside the HDL file describing the circuit and the architecture. Thus, the synthesis module can refer to the passed configuration files to tailor the implementation of the circuit's compound components.

Our technique does not alter components that are explicitly designed by the HDL file; instead, it utilizes the implementation freedom that implicitly declared compound components provide. For instance, consider an HDL file with multiplication ($*$). The synthesis module decides the best way to implement it: a series of additions organized in various types of trees; multiple shifters and adders; or even use a hard block.

Nevertheless, the best synthesis option for each compound operation varies on a number of parameters and goals as well as the bit length of the to-be-synthesized component. Consequently, a "fits-all", general solution is unlikely to consistently produce satisfactory results.

A. Simple vs Compound HDL Operations

Simple operations can be unambiguously synthesized into soft logic. For example, negating a bit can be directly synthesized with a NOT gate. Compound operations have no explicit logic implementation defined in the language specifications. For instance, an array can be synthesized as a set of various types of flip-flops or be mapped to an available hard register on the FPGA. Table I displays all Verilog operators and whether we consider them simple or compound.

B. Configuration File Format

A configuration file passed to the CAD flow is a 2D map describing implementation instructions in multiple levels. First, per different type of implicit operation (addition, multiplication, etc.) and second, per target bit-length of this operation. Constructions can be either implemented in soft logic or a hard block. Each construction is implemented for a specific bit width; if this width is less than the target, the complete construction is defined recursively using a dynamic programming scheme. For example, an instruction for a 32-bit adder could be creating a 24-bit ripple-carry adder and then, looking up the remaining 8 bits recursively.

More formally, a compound operations configuration file is defined as a set of lines, each formatted as follows:

<op> <m> soft | hard <constr> <n>

A valid operation "op" can be any of the defined compound HDL operators, such as addition ($+$) and subtraction ($-$). Each construction can be either soft logic (soft) or map to a hard block (hard). A valid construction "constr" can refer to any type of soft-logic implementation or hard block for that operation (e.g., "carrySelect" or "carryChain" for adders). Finally, two bit widths are also required: the target bit width "m" of the operation and the bit width "n" of the construction—a valid bit width is an integer greater than 0.

C. Configurable Compound Synthesis

Our compound-operator synthesis algorithm extends existing synthesis modules. A configuration file needs to be loaded and the operation synthesis step has to be instrumented as follows: The pair of the operation to be synthesized and its bit width are looked up as a key in the file-map and, if not found the default soft logic is synthesized. Otherwise, a triplet of a type (soft or hard), a construction name and a bit width are retrieved. First, if the mapping requires a specific soft-logic, it is constructed and attached to the netlist. Second, if the instruction requires a hard-block, it is attached to the netlist, if there are sufficient hard blocks of this type and length in the architecture. If not, again the default soft-logic is synthesized. In both cases, the process is repeated recursively until all outgoing wires have been connected to the netlist. The specifics of our solution are elaborated in Algorithm 1.

```
1  configurableCompoundSynth(
   Data: A configuration file map L
   Data: An operation op
   Data: A set of incoming wires in
   Data: A set of outgoing wires out with length m
   Data: An FPGA architecture A
2  )
3  begin
4      if m <= 0 then
5          return
6      end
7      if (op, m) ∉ L then
8          N ⟵ Default soft-logic for op, m
9          Attach N to in
10         Attach out[1 : m] to N
11         return
12     end
13     (logicType, constr, n) ⟵ L(op, m)
14     if logicType is soft then
15         N ⟵ Create soft logic for length n
16         Attach N to in
17         Attach out[1 : n] to N
18         Invoke configurableCompoundSynth for
             in[1 + n : m]
19         return
20     end
21     if A.availableHardBlock(op, n) then
22         N ⟵ A.useHardBlock(op, n)
23         Attach N to in
24         Attach out[1 : n] to N
25         Invoke configurableCompoundSynth for
             in[1 + n : m]
26         return
27     end
28     N ⟵ Default soft-logic for op, m
29     Attach N to in
30     Attach out[1 : m] to N
31     return
32 end
```

Algorithm 1: The Configurable Compound Synthesis Algorithm

D. Choosing Synthesis Configurations

The next step is selecting a proper configuration file, which can be done in a variety of ways. For instance, a specific file can be designed for a specific circuit (or circuit group), or a specific FPGA architecture (or group) or a combination thereof. In general, circuits synthesized with different configurations will also have varying properties, such as FPGA utilization and operating frequency; therefore, selecting an ideal synthesis configuration becomes an optimization problem with an objective function defined by the deployed circuit's metrics—which can be acquired via a CAD tool, such as VTR.

More formally, consider a set of FPGA architectures A, a set of circuits C and a set of configuration synthesis files F. A combination $(a, c, f) \in A \times C \times F$ corresponds to the circuit c synthesized with the configuration file f and deployed on the FPGA a. Since each deployment is associated with a set of performance metrics, an objective function $f : A \times C \times F \rightarrow \mathbb{R}$ can be defined. Therefore, an acceptable configuration file f_{acc} is one whose aggregate objective function value is less than or equal to a target threshold G:

$$\text{Aggregate}\{f(a, c, f_{acc}), \forall a \in A, \forall c \in F\} \leq G \qquad (1)$$

The aggregate function in Equation 1 can be selected by the user. For instance, it could be the arithmetic mean, the geometric mean or a weighted mean. Consequently, if the search space is small, exhaustive techniques suffice; otherwise, more elaborate optimization algorithms, such as hill climbing or linear programming, should be employed.

V. Prototype Implementation on VTR

To gain insight on the feasibility and efficacy of our model, we designed, implemented and open-sourced a prototype on VTR. For brevity, only the addition ($+$) compound operator was chosen since it is frequently used in increment-by-one operations in behavioral loops as well as for multiplication,

division and subtraction. In addition, a dynamic-programming compound-operation configuration search algorithm was designed and implemented on VTR that greedily finds the best synthesis instructions for a given set of goals, circuits and FPGA architectures. Using these, five prototype configuration files for adders were created by five search modes targeting different performance goals.

A. Configurable Compound Synthesis in Odin II

We first extended VTR's synthesizer, Odin II, to parse an input configuration file. A C++ map was used to store each combination of an operation and a target bit width to a struct containing the instructions for this component—soft vs. hard, design type and bit width.

Odin II preprocesses the input Verilog file performing search-and-replace operations. Next, it parses it into an Abstract Syntax Tree (AST), which is created by the context-free rules of Verilog and represents the syntactical relations of the Verilog file's various elements. Third, it traverses the AST and from it, builds a netlist, which is a graph representing the circuit's pins, wires and gates. Before Odin II exits, the netlist is traversed to export the final output to a file.

Our configurable synthesis algorithm could be implemented in any of the aforementioned stages of Odin II: The target operator could be unfolded in the Verilog file by the preprocessor, expanded at the AST level, or wired at the netlist level or even, synthesized at the netlist level via a post-processor. Nevertheless, Odin II perform various optimizations between these stages; using Verilog code unfolding was discarded as there would be loss of information preventing AST optimizations, such as arithmetic reduction.

Furthermore, AST node expansion was discarded because of its increased complexity and lower maintainability. Implementing configurable compound synthesis at the AST would require two steps: first, subdividing the compound operator using AST annotations according to the configuration file. Second, building the defined circuitry inside the netlist, by using these annotations. This would make expansions of our prototype harder since changes would be required in multiple locations. In addition, if an AST does not correlate one-to-one with the provided Verilog file, it reduces ease of debugging.

Expanding and rewiring the netlist was selected, as this enables every other optimization technique to work unimpeded and eases future development of other operators and designs. In particular, we traverse the netlist and each time we encounter an addition node (for future versions, more operators can be added here), we invoke Algorithm 1.

Regarding possible adder designs, we included Carry Select Adder (CSLA) [11], Carry Lookahead Adder (CLA) [12], Carry Select Adder with Binary in Excess (BEC_CSLA) [13] and the baseline Ripple Carry Adder (RCA). Consequently, any addition operation encountered is divided and wired to use any valid combination of the adder available, hard adder or soft adder, according to the instructions stored in the configuration map—in this prototype, we do not support mixed soft/hard block implementations.

B. Search Algorithm

To train our system and create our prototype configuration files, a dynamic-programming optimization script was developed and added in VTR. The script iteratively (and with greedy first selection) searches all possible splits of the target operations and design implementations for up to a maximum bit length. Functional verification through simulation is conducted and any invalid runs are discarded. For each valid combination, the operation that scores the best in the objective function is selected as part of the configuration file (Algorithm 2).

```
1  dynamicConfigurationSearch(
     Data: A maximum bit width W
     Data: A set of target operations OP
     Data: A map from operations to a set of implementations I
     Data: A set of circuits C
     Data: A set of FPGA architectures A
     Data: An objective function f
     Output: A synthesis configuration file F
2  )
3  begin
4      foreach op ∈ OP do
5          for w ← 1 to W do
6              foreach (i, w_s) ∈ I[op] × [1, w] do
7                  if Run[(op, i, w_s)] ≠ null then
8                      Run[(op, i, w_s)] = aggregate results of
                         running this configuration ∀(c, a) ∈ C × A
9                  end
10                 if Run[(op, i, w_s).isValid then
11                     if f(Run[(op, i, w_s)]) < bestScore then
12                         bestScore ← f(Run[(op, i, w_s)])
13                         best = (i, w_s)
14                     end
15                 end
16             end
17             F[op, w] ← (best.i, best.w_s)
18         end
19     end
20 end
```

Algorithm 2: The Dynamic Configuration Search Algorithm

Nevertheless, an exhaustive search was considered to be impractical and futile–we aim to show that we can gain better improvement via input and ouptut path optimization of circuit than predesigned or patterned circuit–instead, we tested all the designs minus the RCA baseline using an adder-tree circuit and found that in the vast majority of the cases, CSLA and BEC_CSLA were solutions that showed the most changes in the target characteristic. Consequently, we then restricted the optimization script to search all possible bit-width groupings that are implemented with either an RCA, a CSLA or BEC_CSLA.

VI. Experimental Evaluation

To experimentally evaluate our technique, we first trained the system to produce five configuration files, each targeting different metrics. Second, we ran the VTR flow with the five configuration files for two FPGA architectures using circuits from the VTR benchmarking suite. The experimental deployment results were then compared against Baseline, the unmodified version of VTR, which did not include config-

TABLE II
OBJECTIVE FUNCTION WEIGHTS OF THE FIVE EXPERIMENTAL SEARCH MODES

Search Mode / Variable	crit path	total dyn	total power	num clbs
CP	1	0	0	0
Dyn	0	1	0	0
Pow	0	0	1	0
Size	0	0	0	1
Mix	1	1	1	1

urable synthesis. It should be stressed that none of the circuits used during the search phase were also used for the evaluation.

A. Search Phase

The objective function of the search script was defined as a weighted average of the metrics acquired after deployment. Five search modes were tested by using the same circuits and FPGA architectures but different sets of optimization weights applied on the VTR metrics for each fully deployed circuit. The first four search modes – critical path (**CP**), dynamic power (**Dyn**), total power (**Pow**) and **Size** – targeted exactly one metric for optimization; the fifth (**Mix**) targeted each of the metrics of the other four with an equal weight (Table II).

After conducting the search, a compound operation synthesis configuration file was created for each of the five optimization modes. Visually examining the configuration files revealed that they were all different from each other; consequently, the prototyped search was able to produce different recommendations for different optimization targets.

B. Testing Phase

To evaluate our system, we used the configuration files we produced with the VTR micro and power benchmarks suite through the VTR flow. To sanitize our test suite, we first removed any circuit that did not report any adder creation. Second, we discarded Verilog circuits that were asynchronous because VTR's power estimation module can only evaluate synchronous circuitry. Third, we checked for benchmarks that were overfitting our search set, a 2-level and a 3-level adder tree of parameterized bit-width. There were 62 benchmarks in the micro regression suite and 18 in the full regression suite. Only eight from micro and nine from full fitted our requirements. We used the flagship 40-nanometer architecture offered by VTR with and without fracturable LUT: (K6_N10_mem32K_40nm) and (K6_frac_N10_mem32K_40nm), as these are well tested.

We conducted sanity tests on all the Verilog files to assure adders were being synthesized before evaluating their performance. The comparison was performed against Baseline: the unmodified VTR version we performed our changes on. We repeated each condition of search mode, benchmark and FPGA architecture on the VTR flow multiple times and recorded the following dependent performance variables produced by the tool: critical path length, total dynamic power, total power and number of CLBs. No variance was recorded.

TABLE III
AGGREGATE TESTING PERFORMANCE RELATIVE DIFFERENCES OVER BASELINE (SIGNIFICANT RESULTS WITH P-VALUE LESS THAN 0.05 ARE MARKED WITH A '*'; RESULTS TARGETED BY THE SEARCH MODE'S OBJECTIVE FUNCTION ARE SHADED)

Search Mode / Variable	crit path	total dyn	total power	num clbs
CP	-3.58%*	2.09%*	3.52%*	3.63%*
Dyn	0.55%	-2.29%*	0.58%	10.14%*
Pow	1.26%	-0.35%	-0.09%	4.44%
Size	3.29%*	-1.17%	-0.70%	3.16%
Mix	3.29%*	-1.17%	-0.70%	3.16%

C. Search Mode Significance

First, we conducted a paired t-test of each search mode against Baseline to decide the performance variables for which significant differences were produced via our technique. We set the significance level to the commonly used threshold of $p < 0.05$. Second, we calculated the geometric mean of the relative difference over Baseline across all combinations of FPGA-benchmark per dependent variable.

The aggregate significance/changes results are displayed in Table III and suggest that the **CP** and **Dyn** search modes were successful in significantly improving their target performance variables. On one hand, **CP** significantly reduced the critical path of the benchmarks by 3.58% and **Dyn** significantly reduced their dynamic power by 2.29%. However, these improvements came at a price: **CP** significantly worsened the other three metrics between 2.09% and 3.63% and **Dyn** significantly increased the number of CLBs by 10.14%. Nevertheless, these two search modes should be considered effective as their target was reached.

On the other hand, none of the **Pow**, **Size** or **Mix** search modes proved to be effective with a p value above our target threshold. Furthermore **Pow** did not record any significant differences; **Size** had no significant difference on its target number of CLBs variable (it also worsened the critical path); and **Mix** significantly worsened the benchmarks' critical path, which it was targeting to improve with a 25% weight. However, this is not surprising as the designed adder circuitry was better fit to improve critical path and as a tangent, also improved dynamic power via improved circuit parallelism.

The three ineffective modes were used to display the flexibility of the system. Regarding adder-only trainable and configurable synthesis, the ripple carry adder has the best performance in terms of size and overall power since it is the smallest circuit that implements the addition logic. Our search set is focused on improving inter adder delay, which skewed the performance metrics towards this direction.

D. Correlations with Benchmark Characteristics

To investigate the effect the characteristics of the benchmarks had on the results, we extracted adder information in Odin II after elaboration—at this stage, any compound operations that resolve to addition have already been transformed to a set of adders. For each benchmark, we calculated the number of adders, the average adder bit-width and standard deviation

Correlations	Adder Information of Benchmarks			
	Number of Adders	Avg of Adder Width	StdDev of Adder Width	Clocked Adder Ratio
CP `critical path`	0.19	-0.32	-0.08	-0.33
Dyn `total dyn`	-0.36	0.34	0.20	-0.24
Pow `total power`	0.21	0.18	-0.17	-0.29
Size `numb clbs`	-0.28	-0.07	0.04	-0.23
Mix `geomean`	-0.22	-0.02	-0.02	-0.27

Fig. 2. Color-Coded Correlations of Target Performance Metrics per Search Mode with General Adder Information of the Benchmarks (The smaller, the better)

Correlations	Distribution of Benchmark Adders per Bit Width			
	up to 7	8 to 15	16 to 31	32 and up
CP `critical path`	-0.18	0.10	0.05	-0.05
Dyn `total dyn`	0.29	-0.28	0.11	0.04
Pow `total power`	0.29	0.39	0.48	-0.78
Size `numb clbs`	-0.08	-0.41	-0.35	0.62
Mix `geomean`	-0.15	-0.32	-0.44	0.64

Fig. 3. Color-Coded Correlations of Target Performance Metrics per Search Mode with Width Distribution of the Adders in the Benchmarks (The smaller, the better)

as well as the proportion of clocked—timing-driven—adders. Furthermore, we also calculated the proportion of adders per bit-lengths: 1 to 7, 8 to 15, 16 to 31, 32 and above.

Figure 2 displays the correlations of the target metric per search mode against the general adder information of the benchmarks. In all five search modes, larger reductions were measured as the proportion of the clocked adders in the benchmarks increased. This is an indication that clocked versus unclocked adders—and thus, compound HDL operators in general—require different synthesis configurations for optimized performance since in our search set we only included clocked adders. In hindsight, enabling different synthesis of clocked and non-clocked operations of the same bit width, might have produced improved results.

Focusing on **CP** and **Dyn**, they performed well for opposite characteristics: **CP** was more effective with larger adders but less effective with more adders. For **Dyn** the correlations were inverse. The result supports that synthesizing compound operators should be customized.

Figure 3 displays the correlations of the metric per search mode against the distribution of adders bit-widths used. **Dyn** performed better when more medium-low-sized (8–15-bit) adders were present but worse with more small (up to 7 bits) adders. This result corroborates the combined correlation in Figure 2: **Dyn** performs better with short adders. Instead, **CP** tilted towards shorter adders in the circuit.

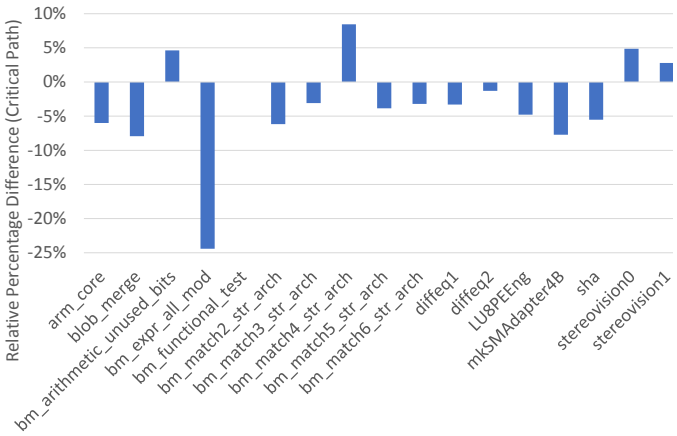

Fig. 4. **CP** Search Mode: Relative Percentage Changes of Critical Path (Objective) over Baseline per VTR Verilog Benchmark

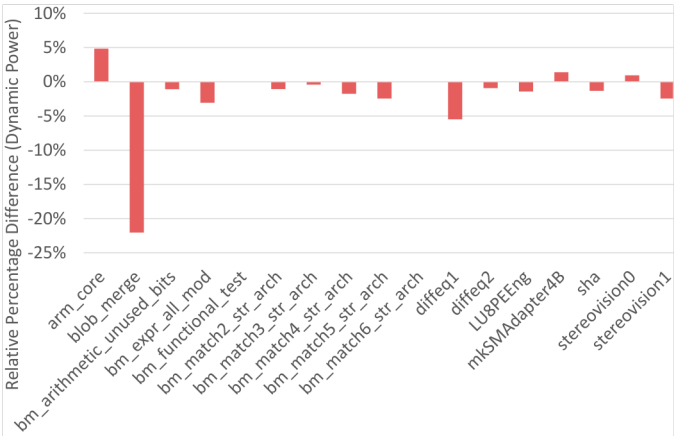

Fig. 5. **Dyn** Search Mode: Relative Percentage Changes of Dynamic Power (Objective) over Baseline per VTR Verilog Benchmark

E. Benchmark Improvements

Next, we investigated the performance changes of each Verilog benchmark per effective search mode (**CP** and **Dyn**). In all cases, we visually inspected the benchmarks' Verilog code to shed light on the recorded variations.

In Figure 4, we display the benchmarks' critical path changes for the **CP** search mode. Out of 17 total benchmarks, 12 registered reductions in their critical path length, four increases and one, no changes. The median improvement was -5.16% and the average, -6.44%. The difference between the average and median is mainly attributed to a specific Verilog file, *bm_expr_all_mod*, which registered a dramatic critical path change of -24.42%; this benchmark is rather small and contained operations organized in a way similar to our search set. Furthermore, four large VTR benchmarks (*arm_core*, *blob_merge*, *bm_match2_str_arch* and *mkSMAdapter4B*) scored critical path improvements around 7%, which we mainly attribute to the efficacy of leveraging BES_CSLA adders for simple increment by-one operations. Regarding benchmarks our technique underperformed,

bm_match4_str_arch relies on heavy usage of chained multiplication without a power of two bit-length.

Figure 5 displays the benchmarks' dynamic power changes over Baseline using **Dyn**. This time, 12 out of 17 benchmarks reduced their dynamic power requirements; whereas, 5 increased it. The median dynamic power change was -1.58% and the average -3.63%. One benchmark, *blob_merge* registered a dramatic change of -22.04%, attributed to its rather high proportion of clocked and short adders as diplayed with the previously discussed correlations. The *arm_core* benchmark performed the worst in this category with a 4.84% change, attributed to its low proportion of clocked adders.

VII. Conclusion

HDL compound operators–addition, multiplication–can be synthesized in a variety of ways, some better suited for a certain performance metric, architecture.

To facilitate customized implicit circuit synthesis, we propose adding reconfigurable compound operators to the synthesizer. Two ways are offered to configure the implicit circuit design: First, a computer architect can fine tune the configuration file to suite a project. Second, since finding the best synthesis configurations requires exponential time, a set of targeted circuits, FPGA architectures and performance metrics can be used to produce a satisfactory configuration file.

We prototyped, open sourced and made available our technique in the academic VTR FPGA toolchain enabling the configurable synthesis of addition and any other operation that reduces to it—e.g. multiplication and subtraction. Our experimental evaluation revealed significant performance improvements in two out of five search sets we explored. In particular, we measured 3.58% average critical path reduction with the search set that aimed in optimizing the critical path; and 2.29% average dynamic power with targeted search.

Two further outcomes of our analysis were: First, a subcircuit that adds the constant 1 can create significant improvements, if implemented as an increment rather than using a full adder; older HDL standards lack such operator. Second, we measured a consistent correlation between improved target metric and proportion of clocked adders. In our prototype design we did not differentiate between synchronous and asynchronous circuits. Since, we trained exclusively with timing-driven adders but experimented with mixed designs, it should be concluded that further improvements and increased fit should be attainable with timing-aware configurable synthesis.

Direct extensions of our work include providing further adder designs; expanding configurable synthesis to further compound operators, such as variable shifts; incorporate mixed soft/hard logic designs; and experimenting with larger search sets leveraging contemporary machine learning algorithms.

Acknowledgment

The authors would like to acknowledge the support of the Natural Science and Engineering Research Council and CMC Microsystems for their support of this project.

References

[1] D. Thomas and P. Moorby, *The Verilog® Hardware Description Language*. Springer Science & Business Media, 2008.

[2] J. Luu, J. Goeders, M. Wainberg, A. Somerville, T. Yu, K. Nasartschuk, M. Nasr, S. Wang, T. Liu, and N. Ahmed, "VTR 7.0: Next generation architecture and CAD system for FPGAs," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 7, no. 2, p. 6, 2014.

[3] P. Jamieson, K. B. Kent, F. Gharibian, and L. Shannon, "Odin II-an open-source verilog HDL synthesis tool for CAD research," in *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, IEEE, 2010, pp. 149–156.

[4] V. Betz and J. Rose, "VPR: A new packing, placement and routing tool for FPGA research," in *Field-Programmable Logic and Applications*, Springer, 1997, pp. 213–222.

[5] A. D. Pimentel, "Exploring exploration: A tutorial introduction to embedded systems design space exploration," *IEEE Design & Test*, vol. 34, no. 1, pp. 77–90, 2017.

[6] J.-P. Legault, "Experimental verilog synthesis features for odin & alternative multiplier hard-block for fpga," University of New Brunswick, Tech. Rep., 2017.

[7] R. Brayton and A. Mishchenko, "Abc: An academic industrial-strength verification tool," in *International Conference on Computer Aided Verification*, Springer, 2010, pp. 24–40.

[8] B. Yan and K. B. Kent, "Hard block reduction and synthesis improvements in Odin II," in *Rapid System Prototyping (RSP), 2015 International Symposium on*, IEEE, 2015, pp. 126–132.

[9] P. Patros and K. B. Kent, "Automatic detection and elision of reset sub-circuits," in *Rapid System Prototyping (RSP), 2016 International Symposium on*, IEEE, 2016, pp. 1–7.

[10] J. Luu, C. McCullough, S. Wang, S. Huda, B. Yan, C. Chiasson, K. B. Kent, J. Anderson, J. Rose, and V. Betz, "On hard adders and carry chains in FPGAs," in *Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*, IEEE, 2014, pp. 52–59.

[11] V. Viswam and S. S. Nair, "Vhdl architecture for delay efficient sqrt carry select adder," *International Journal*, vol. 6, no. 6, 2016.

[12] R. Zlatanovici, S. Kao, and B. Nikolic, "Energy–delay optimization of 64-bit carry-lookahead adders with a 240 ps 90 nm cmos design example," *IEEE Journal of Solid-State Circuits*, vol. 44, no. 2, pp. 569–583, 2009.

[13] P. S. Wasekar, "An area efficient carry select adder using binary excess converter," *Imperial Journal of Interdisciplinary Research*, vol. 2, no. 10, 2016.