MIRROR: MaxImizing the Re-usability of RTL thrOugh RTL to C CompileR

Md Imtiaz Rashid and Benjamin Carrion Schafer
The University of Texas at Dallas
Department of Electrical and Computer Engineering
mdimtiaz.rashid@utdallas.edu, schaferb@utdallas.edu

***Abstract*—This work presents a RTL to C compiler called MIRROR that maximizes the re-usability of the generated C code for High-Level Synthesis (HLS). The uniqueness of the compiler is that it generates C code by using libraries of pre-characterized RTL micro-structures that are uniquely identifiable through perceptual hashes. This allows to quickly generate C descriptions that include arrays and loops. These are important because HLS tools extensively use synthesis directives in the form of pragmas to control how to synthesize these constructs. E.g., arrays can be synthesized as registers or RAM, and loops fully unrolled, partially unrolled, not unrolled, or pipelined. Setting different pragma combinations lead to designs with unique area vs. performance and power trade-offs.**

Based on this, the main goal of our compiler is to parse synthesizable RTL descriptions specified in Verilog which have a fixed micro-architecture with specific area, performance and power profile and generate C code for HLS that can then be re-synthesized with different pragma combinations generating a variety of new micro-architectures with different area vs. performance trade-offs. We call this 'maximizing the re-usability of the RTL code because it enables a path to re-target any legacy RTL description to applications with different constraints. In particular we deal with pipelined descriptions in this work due to their uniqueness. Experimental results show that our proposed compiler is very effective, opening the door to automating the re-optimization of legacy hardware designs previously manually optimized using low level Hardware Description Languages (HDLs). We aim at making this compiler framework open source and available to the research community.

I. Introduction

More and more companies are now relying on High-Level Synthesis (HLS) to design their hardware circuits, and in particular the hardware accelerators that most heterogeneous System-on-Chips (SoC) now include. These are typically Digital Signal Processing (DSP) accelerators, image processing or encryption engines. One of the main advantages of raising the level of design abstraction from the RT-level (RTL) to the behavioral level is that a variety of different designs, each with different area vs. performance and power trade-offs can be easily generated from the same behavioral description (ANSI-C, C++ or SystemC) by simply setting different synthesis options. These synthesis options typically have the form of synthesis directives in the form of pragmas inserted at the behavioral description to force the HLS tool to synthesize arrays as either RAM/ROM or registers, and unroll loops fully, partially, not unroll or pipeline. One of the main problems that often prevents companies from embracing HLS is having large amounts of legacy RTL code. This implies that they often need to manually re-optimize it when used in different projects that require different area vs. performance, and power trade-offs including having to completely re-timed any pipelines to make use of smaller technology nodes or newer FPGAs.

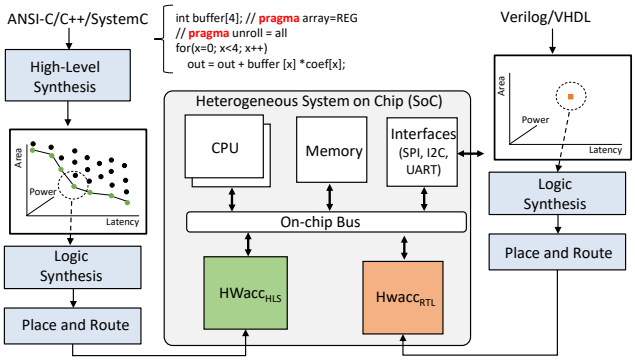

Fig. 1: Target architecture with two HW accelerators. One generated through HLS, the other through Verilog/VHDL.

Fig. 1 shows an overview of a typical heterogeneous SoC composed of multiple embedded processors, on-chip memory, diverse interfaces (i.e., SPI, I2C and UART) and two hardware accelerators. Fig. 1 also shows the VLSI design flow to generate those accelerators. In particular, $HWacc_{HLS}$ is generated using HLS, while $HWacc_{RTL}$ is generated using Verilog or VHDL. As also shown in the figure, $HWacc_{HLS}$ allows to generate a variety of designs (in the given dot-product snippet by inserting a pragma to synthesize an array and loop) and then choose the design that meets the product's constraints best. In contrast, when designing these accelerators at the RT-level, the user manually fixes the micro-architecture of the design, leading to a design with specific area, performance, and power constraints. The designer would have to manually re-write and re-verify the RTL code if a new design with different area vs. performance trade-offs is needed. This is obviously time consuming and error prone.

To address these issues, in this work we present an RTL to C compiler that can convert synthesizable Verilog code into ANSI-C code optimized for HLS, and in particular to maximize its re-usability. This involves generating C code that includes arrays and loops in the new C description that can in turn be explored setting different synthesis directives.

Due to their uniqueness, in this work we mainly target pipelined architectures which are often the main way that the hardware accelerators (DSP and image processing) are synthesized. We will continue extending the compiler framework in order to support any synthesizable RTL description, but leave that for future work. In summary, the main contributions of

978-3-9819263-7-8/DATE23/© 2023 EDAA

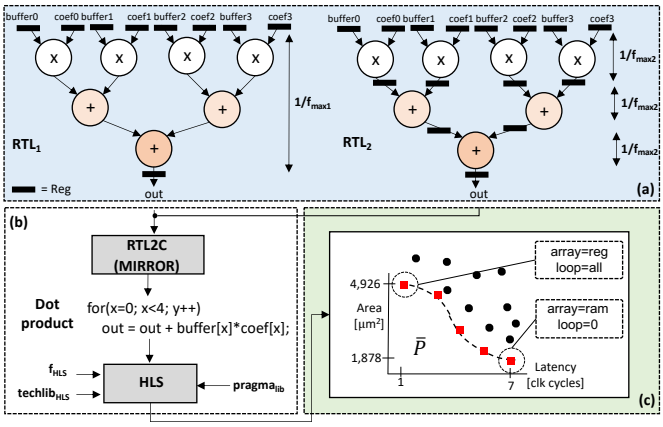

Fig. 2: Motivational example.(a)Two different dot product RTL implementations (RTL1 and RTL2); (b) Proposed flow overview; (c) Result of HLS design space exploration of converted C code.

this work are:

- Introduce and RTL to C compiler framework called MIRROR that takes as input synthesizable Verilog code and generates ANSI-C code optimized for HLS.
- Present a smart RTL structure detection method based on perceptual hashing to convert these structures into efficient ANSI-C code.
- Present extensive experimental results highlighting the effectiveness of our proposed compiler framework.

II. Motivational Example

Fig. 2(a) shows two circuits that implement the dot-product. In both cases a tree-height reduction optimization is performed to fully parallelize the computation and reduce the delay. The main difference between both circuits is that in the first case (RTL_1) the entire dot-product is executed in a single clock cycle, while in the second case (RTL_2) the circuit is pipelined by inserting registers after each functional unit (FUs). In RTL_1 the maximum frequency is limited by the chained delay of the multiplier and adders (f_{max1}), while in RTL_2 the maximum frequency is limited in this case by the multipliers to f_{max2}, where here $f_{max2} > f_{max1}$.

Fig. 2 also shows an overview of our proposed flow. In particular the RTL2C compiler that reads in the Verilog descriptions of RTL_1 and RTL_2 and outputs a behavioral description optimized for HLS. As shown, the description should be exactly the same for both cases composed of a for loop that does dot product on the arrays that contain the data. Outputting this code as a for-loop instead of individual expressions enables to set different synthesis directives to control how to synthesize the loop and array, and thus, generating designs with different area vs. performance trade-offs. Out of all the synthesis directive combinations, the user is typically only interested in the combinations that lead to the Pareto-optimal designs, highlighted as red squares in the figure that form the Pareto front ($\bar{P}$). In this particular examples, two of the Pareto-optimal designs are highlighted, which are obtained by fully unrolling the loop and synthesizing the array as registers. This leads to the largest but fastest designs, while synthesizing the array as RAM and not unrolling the loop, leads to the smallest, but slowest designs.

The proposed RTL to C compiler thus, enables to re-optimize legacy RTL code that was manually optimized for a particular set of constraints and technology node or even FPGA and automatically generate new designs with different design trade-offs. Based on this, we can formally define the problem in this work as follows:

Problem Definition: Given a synthesizable Verilog description (RTL_{in}), compile it to an untimed behavioral description (C_{HLS})that is functional equivalent $f(RTL_{in}) = f(C_{HLS})$, optimized for HLS such that the generated behavioral description $C_{HLS} = \{array, loops\}$, contains arrays, and loops, such that setting different synthesis directives on C_{HLS} leads to a variety of different micro-architectures with unique area vs. performance and power constraints ($\bar{P}_{new}$).

III. Related Work

So far, the problem of generating C/C++ descriptions from RTL (VHDL or Verilog) has primarily had the objective of creating simulation models to either verify the RTL code with newly generated C/C++ code or to accelerate the RTL simulation by abstracting the RTL code, but not to optimize the C/C++ description for re-optimization. In [1] a translation tool that generates a C++ model from VHDL achieved faster simulation time compared to traditional RTL simulations. In [2] synthesizable Verilog is translated into C++ to reduce the number of delta cycles by merging processes together. V2C is a Verilog to C compiler used for hardware property verification [3]. Some commercial tools like Carbon Design Systems (acquired by ARM) [4] can convert RTL models into cycle-accurate and register-accurate C++ models targeted mainly for the creation of virtual platform from legacy RTL code.

Closer to this work, Bombieri et al. generate C++ models by abstracting many architectural details of the original RTL code for HLS [5], [6]. This previous work, directly converts the RTL into C looking at the structure of the RTL description and hence has limited capabilities. Another recent work includes Veriintel2C [7] which converts Verilog descriptions into synthesizable C descriptions, but is limited to detecting loops by looking at back edges in the CDFG generated from the RTL code. Thus, it cannot convert tree structures nor pipelined circuits as shown in the motivational example. Finally, he authors in [8] showed with a rudimentary RTL to C compiler how legacy RTL code can be modernized.

To the best of our knowledge this is the first work that is able to locate and convert into optimized C code for HLS, pipelined circuits and typical structures encountered in most DSP and image processing applications like trees a using an extensible database of pre-characterized micro-structures.

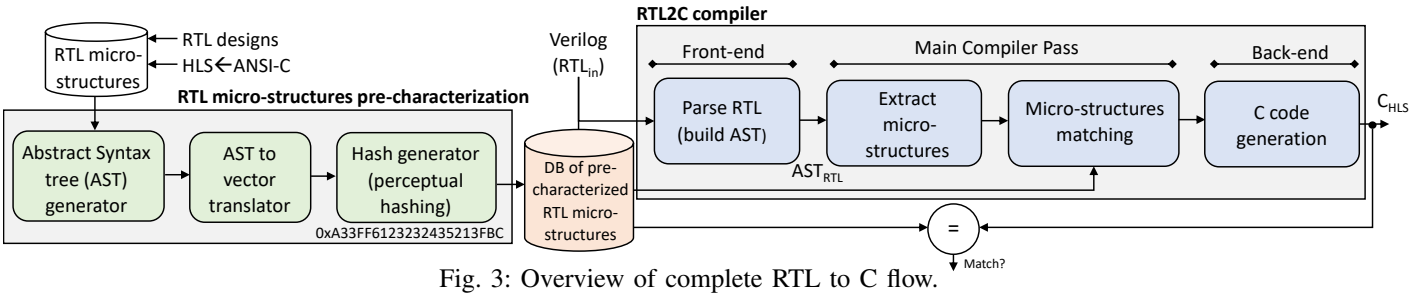

Fig. 3: Overview of complete RTL to C flow.

IV. Proposed RTL to C Compiler Framework

Fig.3 shows an overview of the entire proposed compiler framework. The complete framework is written in Python and is composed of two main stages. The first stage, pre-characterizes different RTL micro-structures identifying each structure with a unique hash value, while the second stage does the actual RTL to C compilation using this data. The inputs to the compiler are thus, the synthesizable Verilog description to be converted (RTL_{in})and a database of pre-characterized RTL micro-structures. The output is the synthesizable C code optimized for HLS (C_{HLS}). This C code can then be passed to an automated HLS design space explorer in order to automatically generate designs with different trade-offs or manually re-synthesized with different constraints. It is needless to say that the original RTL description and the newly generated C code must be functional identical: $f(RTL_{in}) = f(C_{HLS})$. The next subsections describe the entire compilation process in detail.

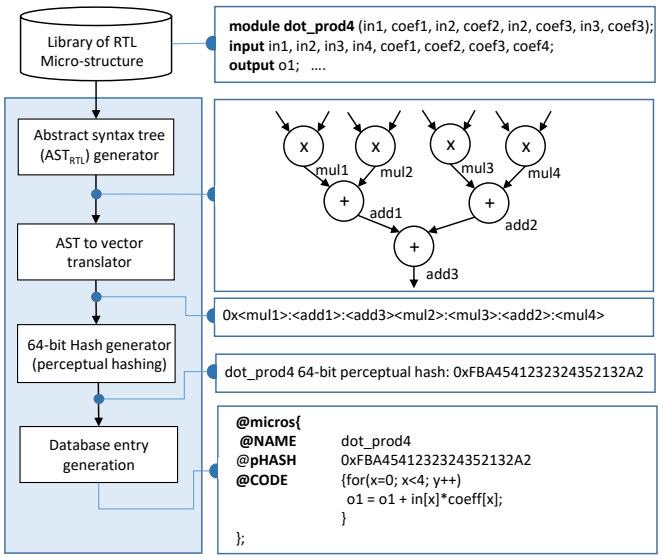

Fig. 4: RTL Micro-Structure pre-characterization flow.

Stage 1: RTL Micro-structures Pre-characterization: This first stage takes as input a library of different RTL micro-structures ($MicroS$) like the ones shown in the motivational example (Fig. 2) and pre-characterizes them. Fig. 4 shows how the database of pre-characterized micro-structures are generated. These micro-structures are typical structures found in most DSP and image processing applications. For this, we used the open source OpenCore [9] RTL designs and isolate these micro-structures, and also generate other micro-structures from behavioral descriptions for HLS. In this case we isolate individual loops in the behavioral description and synthesize them with different pipelining Data Initiation Intervals (DIIs). These behavioral benchmarks come from the main HLS benchmarks (CHStone [10] and S2CBench [11]). In the behavioral benchmark case, a micro-structure is generated by isolating individual loops or nested loops, while for the manually written RTL designs, these are manually extracted from the benchmarks. In total over 70+ micro-structures were identified and pre-characterized.

Each micro-architecture ($MicroS_i$) is fully characterized by a unique hash value that easily identifies it and the optimized C code that translates this RTL micro-structure into optimized C code for HLS as shown in Fig. 4. A unique hash value is needed to easily identify these micro-structures in new unseen RTL descriptions. This hash value should detect when two micro-structures are structural different, but also allow to identify if they are similar, thus, a method to fingerprint each micro-structure in a robust way is needed.

To address this, in this work we make use of perceptual hashing [12], which is an algorithm commonly used in digital forensics. The key advantage of perceptual hashing is that it is robust enough to consider dissimilarities, but flexible enough to distinguish between different micro-kernel structures. Traditional cryptographic hashing methods, e.g., MD5 and SHA result in completely different hashes if a single bit changes, thus, making them not useful for this application. Perceptual hashing, in contrast, can be used to compare two micro-structures by calculating their Hamming distance (H).

Thus, as shown in Fig. 4, to generate the hash value, we build an abstract syntax tree of the micro-structure, where each node corresponds to an RTL component (e.g., adder, multiplier, mux, etc.), and create a vector based on each node-type following a depth-first tree traversal method. Basically, each node in the AST is assigned a unique value which is then concatenated to the other nodes' values. This vector is in turn passed to the hash generator that generates a unique 64-bit perceptual hash. This perceptual hash value is considered the micro-structure signature. The output of this stage is database with an individual entry for each pre-characterize RTL micro-structure $MicroS_i$ that contains for each micro-structure $MicroS_i = \{pHash_i, C_i\}$, here $pHash_i$ is the perceptual hash value and C_i the ANSI-C code snippet optimized for HLS for that particular micro-structure, as shown in Fig. 4.

Stage 2: MIRROR RTL2C Compiler: The compiler itself can be sub-divided into three main phases, as shown in Fig.3. The next sub-sections describe these in detail.

Step 1: Compiler Front-end: This first step parses the Verilog descriptions and creates an abstract syntax tree (AST_{RLT}) from it. For this it uses a Python-based Hardware design parser called Pyverilog [13]. The parser checks for syntax errors in the Verilog code, performs basic technology independent optimization like constant propagations and outputs an AST representation (AST_{RTL}).

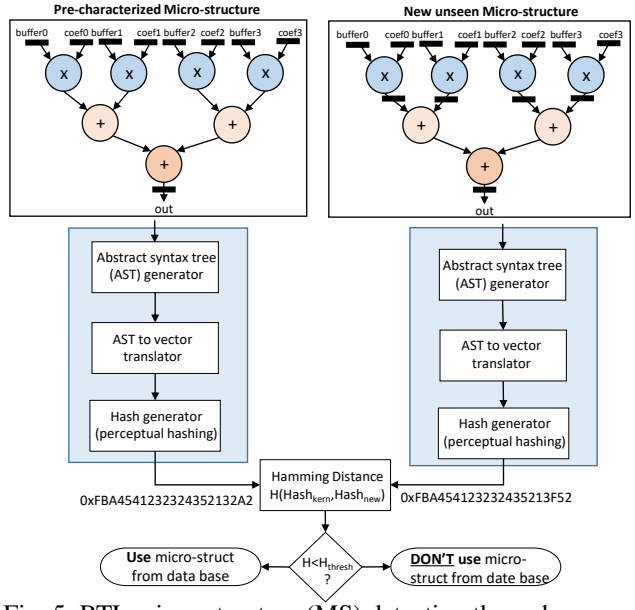

Fig. 5: RTL micro-structure (MS) detection through perceptual hashing of RTL abstract syntax tree (AST_{RTL}).

Step 2: Main Compiler Pass: This second stage is the main compiler pass that analyzes the AST_{RTL} to generate the optimized C code for HLS. This pass is based on a library with different rules that are applied to the AST. These rules are expressed in a modular and extensible way in the compiler framework such that new rules can be easily added. The first rule parses the AST_{RTL} and automatically extracts any micro-structures in it. It then checks in the micro-structures database generated in the first stage if any of the micro-structures extracted match by comparing their perceptual hash.Fig.5 shows an example. if there is a match, then the portions of the AST_{RTL} is *substituted* by the corresponding behavioral description of that micro-structure stored in the DB. In the example shown the structures are similar, but the new one is pipelined, while the original in the database is not.

The similarity/difference between two micro-structures is estimated by computing the hamming distances between their hash values, e.g., $Hash_p$ and $Hash_q$, $Hash_{diff}(p,q) = \{(Hash_p; Hash_q)\}$, where a large H represents a large difference in the micro-structures being compared. In the example given in Fig.5, the only changes between the two micro-structures are the flip-flops are inserted between the multipliers and the adder tree. Thus, the hash values are almost identical, and hence, the result is that the pre-characterized micro-structure can be used to translate this new unseen micro-structure into ANSI-C. We tried different hamming distances during the experimental setup and noticed that 10% threshold lead to good results.

The compiler also leverages that most commercial HLS tools perform automatic internal variables bitwidth optimization. Basically, designers only must specify the desired bitwidth at the primary inputs and outputs and the HLS tool will automatically adjust the bitwidth of the internal variables. Thus, this stage also creates a database of additional features required by the compiler. These include the name and bitwidth of the primary IOs, and internal signals as well as the functional units (FUs) and assignment expressions.

This information is used in the following compiler passes that use additional rules stored in the extensible rule database to continue the conversion process. E.g., detecting constant coefficient multiplication. Many DSP applications contain these types of operations, but these multiplications can be re-written as either a shift in the case of powers of 2 multiplications or shifts and addition/subtractions. Moreover, shift can be simplified using simple bitwise extraction and concatenation. E.g., expressions like the following:

```
assign odata = {3'h0, data[7:3]};
```

needs to be converted to $odata = data/8$;

The compiler continues with the rule-based transformation adding to the internal database all the of the transformations found and by generating a new internal data structure of the AST_{RTL} pointing to the optimizations identified in the internal database.

Step 3: Compiler Back-end: This last compiler stage takes as input the analyzed AST with the different conversion optimizations and outputs the ANSI-C code for HLS (C_{HLS}). As mentioned previously the only exact bitwidths that will be set are the ones from the primary IOs, but because ANSI-C does not allow to set custom data types we use the custom data types that commercial HLS tools always include. E.g., Xilinx Vivado and Siemens Catapult both use the ac_type data types, and Cadence Stratus the SystemC sc_int data type that allows to specify any bitwidths. This step is extremely important as it affects the bitwdith of the FUs that will be generated when synthesizing the behavioral description.

Finally, as shown in Fig.3, the user can decide to run an automatic functional verification check to make sure that the converted C code functionally equivalent to the original RTL code. This process is automated if the RTL code given includes a testbench with separate test vectors. In this case our proposed framework automatically generates a main function wrapper on top of C_{HLS} passing the test vectors to the newly generated C code that is instantiated as a function in it. gcc is then used to compile the entire program, which is then executed and the outputs compared against the golden outputs from the RTL simulation.

V. Experimental Results

Table I shows an overview of the experimental setup used to test our proposed flow. NEC CyberWorkBench v.6.1.1 is used

TABLE I: Experimental Setup

HLS Tool	NEC CyberWorkBench 6.1.1
Logic Synthesis	Synopsys Design Compiler v.0-2018.06-SP1
HLS Frequency	200 MHz
Synthesis Technology	Nangate 45nm Opencell
Compiler front-end	Pyverilog [13]
Compiler	Python 3.0

as HLS tool to convert the behavioral description back into RTL. Synopsys Design Compiler v.0-2018.06-SP1 is used to synthesize manually written RTL code (Verilog) and compare it with our converted C code that is in turn synthesized back to RTL using Cadence Stratus. In all cases the target technology chosen is Nangate 45nm OpenCell and the target synthesis frequency 200MHz. The compiler itself was written in Python (4,000 lines of code) and we use Pyverilog [13] as front-end to parse the Verilog code.

Two set of independent experiments were conducted to measure the effectiveness of our proposed flow. First, to quantify numerically the quality of the conversion, we take several benchmarks from the open source S2CBench benchmark suite written in SystemC for HLS and perform a HLS DSE on them obtaining the Pareto-optimal designs. We then choose the Pareto-optimal design (RTL) with highest performance (throughput) and convert it back to ANSI-C through our proposed RTL to C compiler and re-explore it. Intuitively, if the compiler is successful, the original DSE results and the DSE results from the compiled behavioral description should match. To measure the actual quality of the conversion we use Average Distance to Reference Set (ADRS) which is typically used to compare multi-objective optimization problems like these ones [14].

ADRS indicates how close a Pareto-front is to the reference front. In this particular case we run an exhaustive enumeration of all possible pragma combinations to obtain the optimal designs. Although time consuming, this guarantees to find the optimal designs in all cases. The smaller the ADRS value is, the closer the obtained approximated front is to the reference front. Given a reference Pareto-front $\Gamma = \gamma_1 = (a_1, l_1), ..., \gamma_n = (a_n, l_n)$ and an approximate Pareto front $\Omega = \omega_1 = (a_1, l_1), ..., \omega_n = (a_n, l_n)$ with a $\in A$ and l $\in L$, where A is the design area and l its correspondent latency. ADRS can thus, be defined as follows:

$$ADRS(\Gamma,\Omega) = \frac{1}{|\Gamma|} \sum_{\gamma \in \Gamma} \min_{\omega \in \Omega} f(\gamma, \omega)$$

$$f(\gamma = (a_\gamma, l_\gamma), \omega = (a_\omega, l_\omega)) = max\{|\frac{a_\omega - a_\gamma}{a_\gamma}|, |\frac{l_\omega - l_\gamma}{l_\gamma}|\}$$

The lower the distance value (ADRS) is, the more similar two Pareto sets are. Based on this the exploration results of the original S2CBench description and our converted C code should both have an ADRS of 0%. We also compare our method vs. the state of the art following the instructions from Veriintel2C [7] to compare our compiler with this previous work.

The second set of results uses manually written RTL code obtained from OpenCores [9] and shows how our proposed compiler can generate a variety of different designs from a single fixed micro-architecture (RTL code), as this is the ultimate goal of this work.

TABLE II: Experimental results comparing quality of search space of original C code vs. converted through our compiler

Benchmark	**Original ADRS[%]**	**Previous** ([7]) **ADRS[%]**	**MIRROR (Proposed) ADRS[%]**
ave8	0.0	100	0.0
fir	0.0	90.3	0.0
sobel	0.0	97.1	12.4
interp	21.8	85.1	1.07
decim	0.0	100	13.79
dct	0.0	100	0.0
Avg.	**3.6**	**94.5**	**4.5**

It should noted that none of these behavioral nor RTL benchmarks were used to build the micro-structures pre-characterization library.

Experiment 1 : Design Space Comparison: Table II compares the quality of the Pareto-optimal designs obtained from the C code generated by our compiler (converted) compared to the Pareto-optimal designs obtained from the benchmarks (original) and previous work (Veriintel [7]). To reduce the effect of any stochastic effect of the design space exploration process, we use an exhaustive search to explore all of the benchmarks so that we can compare the optimal designs found in both of the behavioral descriptions.

From the results shown in the table we can make the following observations: **Observation 1:** Surprisingly for one of the benchmarks (interpolation filter - interp), the converted C code leads to better results than the original behavioral description (ADRS of 1.07% vs. 21.8%). Upon investigating this, we noticed that our compiler grouped all of the three FIR filters that form this interpolation filter into a single for loop, while the original behavioral description has each filter described into separate for loops. This is a more natural way of expressing the interpolation filter, but it seems that this leads to problems in the HLS tool, not being able to perform optimizations across loops. **Observation 2:** For three of the benchmarks our compiler generates an equivalent behavioral description that leads to the same ADRS (ave8, fir and dct). **Observation 3:** For two other benchmarks, the exploration lead to worse results. In particular decimation (decim) and sobel. Upon investigating this, we noticed that these benchmarks contain multi-dimension arrays, which can be explored by expanding and partitioning different dimensions of the array. This leads to a wider search space as compared to our compiled C code that does not have multi-dimension arrays. Unfortunately, the proposed compiler currently does not support this, but based on these results this has been added as a future extension. **Observation 4:** Our proposed work leads to much better results than the state-of-the art. This previous work uses the back-edges in the CDFG to build loops, but for these particular DSP and image processing applications that are pipelined and contain multiple tree like structures (no back-edges), this previous could not generate any loops, and hence, the generate C code could not be explored at all. The C code was generated we basically 'flatten' C code similar to previous work that compiled RTL to C code just to create a

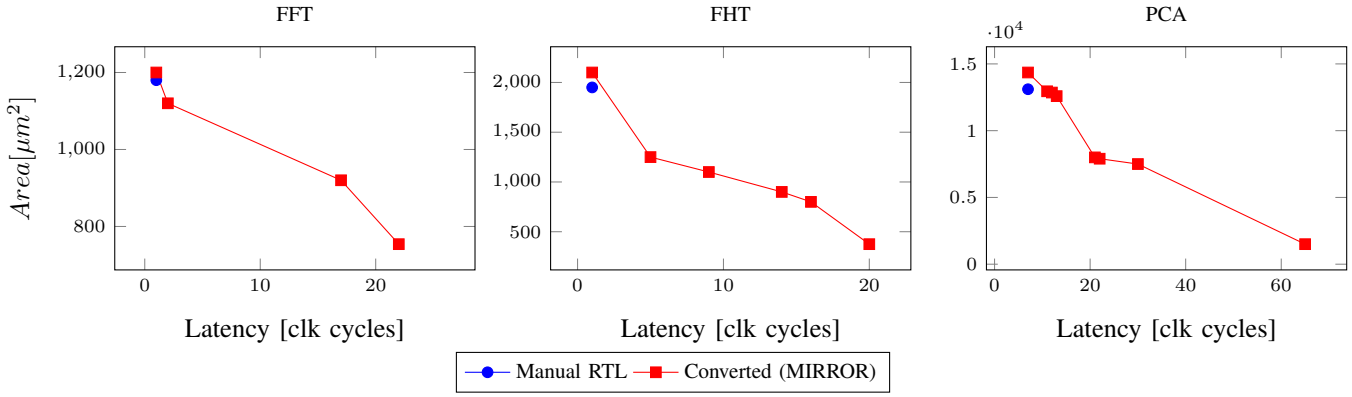

Fig. 6: Manually optimized RTL code (Manual) vs. trade-off curve of obtained through our RTL to C compiler (MIRROR).

functional verification model [1], [3] .

Experiment 2 : Manually Optimized RTL : Figures 6 show the results in terms of area and latency of three manually written RTL designs obtained from OpenCores [9] (manual RTL) and synthesized using Synopsys DC compiler vs. the results of the converted C description subsequently explored to obtain additional designs (converted). In particular a pipelined 128-point FFT, Fast Hadamard Transform (FHT) and Principal Component Analysis (PCA). From the results shown in these figures we can make the following observations: **Observation 1:** In all cases the converted behavioral description can generate a design that has the exact same performance as the manually optimized design (same latency). This is important because it shows that the compiler does not lead to any performance penalties. **Observation 2:** Although our proposed flow does not lead to any performance penalties, the designs have larger area in two out of the three benchmarks (FHT and PCA). In particular for the PCA benchmark the area for the design with same performance as the manual RTL increases by 9%. One of the reasons observed is that our compiler relies on the HLS tool to perform automatic bitwidth optimization. We noticed that in this particular case the manually optimized design did more aggressive optimizations than the tool. **Observation 3:** Our proposed flow is able to generate a variety of different designs with unique area vs. latency trade-offs from the original manually optimized RTL code. This is one of the most significant contributions of this work as it allows to re-target any legacy RTL description for newer applications and products.

Based on these results we can conclude that our proposed RTL to C compiler flow works well and that is able to expand the search space of hand coded RTL code efficiently. We have also shown that the compiler sometimes leads to better results than manually written C code for HLS as it does some optimizations like loop grouping that are not natural when writing C code manually.

VI. Conclusion

In this work, we have introduced an RTL to C compiler that has the objective to maximize the ability of maximize the re-usability of newly converted behavioral description by generating mainly array, loops. Experimental results show the effectiveness of our compiler. We believe that this flow has the potential to accelerate the adoption of HLS as well as to allow an easy and painless path to modernize legacy hardware designs written in Verilog or VHDL. Future work will extend the compiler to be able to generate functions and multi-dimensional arrays as we have seen that these leads to additional optimal designs when explored.

References

[1] W. Snyder, P. Wasson, and D. Galbi, "Verilator – convert verilog code to c++/systemc," Oct. 2016. [Online]. Available: http://www.veripool.org/wiki/verilator

[2] W. Stoye, D. Greaves, N. Richards, and J. Green, "Using rtl-to-c++ translation for large soc concurrent engineering: a case study," *Electronics Systems and Software*, vol. 1, no. 1, pp. 20–25, Feb 2003.

[3] R. Mukherjee, "v2c - a verilog to c translator tool," Oct. 2022. [Online]. Available: http://www.cprover.org/hardware/v2c/

[4] C. (now ARM), "Carbon model studio," Oct. 2017. [Online]. Available: http://carbondesignsystems.com/

[5] N. Bombieri, F. Fummi, and G. Pravadelli, "Abstraction of rtl ips into embedded software," in *DAC*, ser. DAC '10. New York, NY, USA: ACM, 2010, pp. 24–29.

[6] N. Bombieri, H. Y. Liu, F. Fummi, and L. Carloni, "A method to abstract rtl ip blocks into c++ code and enable high-level synthesis," in *DAC*, May 2013, pp. 1–9.

[7] A. Mahapatra and B. Carrion Schafer, "VeriIntel2C: Abstracting RTL to C to maximize High-Level Synthesis Design Space Exploration," *Integr.*, vol. 64, pp. 1–12, 2019.

[8] Md Imtiaz Rashid, Qilin Si and Benjamin Carrion Schafer, "Modernizing hardware circuits through high-level synthesis," in *ISCAS*, 2022, pp. 1739–1743.

[9] OpenCores, "opecores.org," Oct. 2022. [Online]. Available: https://opencores.org/

[10] Yuko Hara *et al.*, "Chstone: A benchmark program suite for practical c-based high-level synthesis," in *ISCAS*, May 2008, pp. 1192–1195.

[11] B. Carrion Schafer and A. Mahapatra, ""s2cbench: Synthesizable systemc benchmark suite for high-level synthesis"," *Embedded Systems Letters, IEEE*, vol. 6, no. 3, pp. 53–56, 2014.

[12] J. Fridrich and M. Goljan, "Robust hash functions for digital watermarking," in *Proceedings International Conference on Information Technology*, 2000, pp. 178–183.

[13] S. Takamaeda-Yamazaki, "Pyverilog: A python-based hardware design processing toolkit for verilog hdl," in *Applied Reconfigurable Computing*, vol. 9040, Apr 2015, pp. 451–460.

[14] E. Zitzler, L. Thiele, M. Laumanns, C. M. Fonseca, and V. G. da Fonseca, "Performance assessment of multiobjective optimizers: an analysis and review," *IEEE Transactions on Evolutionary Computation*, vol. 7, no. 2, pp. 117–132, April 2003.