Finding Bugs in RTL Descriptions: High-Level Synthesis to the Rescue

B. Parchamdar and B. Carrion Schaefer
{baharealsadat.parchamdar,schaferb}@utdallas.edu
Deptm. of Electrical and Computer Engineering, The University of Texas at Dallas
Richardson, Texas, USA

ABSTRACT

Most Register Transfer Level (RTL) designs originate from behavioral descriptions specified in C or C++ often written by Software (SW) designers. Hardware (HW) designers then manually describe an efficient hardware implementation of that application using a Hardware Description Language (HDL) like Verilog or VHDL. Although it has been shown that High-Level Synthesis (HLS) provides a direct path to synthesizing these behavioral descriptions into RTL, the quality of the generated RTL is often still unacceptable, hence, requiring to manually design the HW. This is nevertheless time consuming and error prone. In particular, finding bugs introduced in the manual design is very tedious as HW designers rely on long simulations that generate large waveforms that have to be thoroughly scrutinized.

To address this, in this work we present an automated method to accurately point to where in an RTL description a bug is located by using HLS. In particular we leverage the ability of HLS to generate a variety of different micro-architectures to automatically find a design architecturally 'similar' to the manually optimized one in order to help locate the bugs. This also involves automatically updating the HLS technology library in order to match the delay from floating-point functional units (FUs) such that these match the equivalent integer functional units used in the manually optimized RTL design as one of the main optimization that can lead to numerical stability issues in the manual translation is floating-point to fixed-point data translation.

KEYWORDS

RTL faults, High-Level Synthesis, Design Space Exploration, Similarity estimation.

ACM Reference Format:
B. Parchamdar and B. Carrion Schaefer. 2024. Finding Bugs in RTL Descriptions: High-Level Synthesis to the Rescue . In *61st ACM/IEEE Design Automation Conference (DAC '24), June 23–27, 2024, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3649329.3658258

1 INTRODUCTION

High-Level Synthesis (HLS) converts untimed behavioral descriptions (e.g., ANSI C or C++) into efficient hardware (HW) circuits specified in either Verilog or VHDL. It has been shown that raising the VLSI design abstraction from the Register Transfer level (RTL) to the behavioral level has many advantages like reducing the turn-around time, allowing faster simulations and the ability to generate different types of hardware circuits from the same behavioral descriptions by simply setting different synthesis options [1].

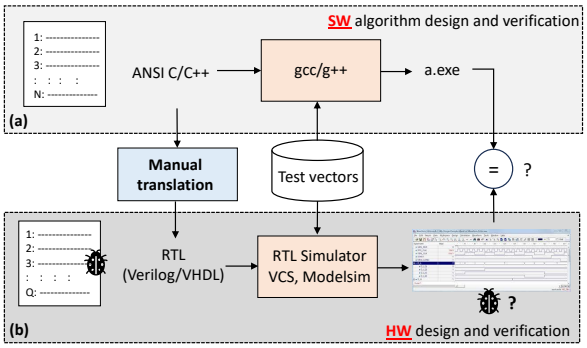

Figure 1: Overview of the problem addressed in this work. (a) SW development in C/C++. (b) Manual HW design of C/C++ application. How to find bugs (faults) introduced by a HW designer?

One of the main problems with HLS though, is that the quality of the generated circuit is often not as good as manually optimized RTL code written using a traditional Hardware Description Language (HDL) like Verilog or VHDL [1, 2]. This is specially true for control intensive applications. Thus, many VLSI design companies start their Integrated Circuits (ICs) design by first developing the algorithmic description of the application, e.g., image processing, digital signal processing or encryption algorithm and then manually implementing a hardware architecture that can efficiently execute this application. This is particularly true for the HW accelerators that most ICs now include for energy efficient reasons.

Fig. 1 shows an overview of this flow. The algorithmic design and verification is first done in software (SW) and then the HW designer manually creates an optimized architecture by usually parallelizing the SW description and performing a variety of optimizations like floating-point to fixed-point data conversion and pipelining in order to achieve the desired area, performance, and power constraints.

The main problem with this approach is that it is time consuming and error prone. As shown in the figure, SW can be quickly compiled using a standard compiler like gcc or g++, and the compiled program simulated fast. HW designers use slow RTL simulators (e.g., VCS or Modelsim) and rely on large waveforms to find bugs. These waveforms are extremely large as HDLs are much more verbose than SW descriptions.

Other approaches suggested include the use of formal verification techniques [3]. These are nevertheless still complex to use and expensive, hence not everyone has the adequate training of how to use these effectively nor access to them.

DAC '24, June 23–27, 2024, San Francisco, CA, USA

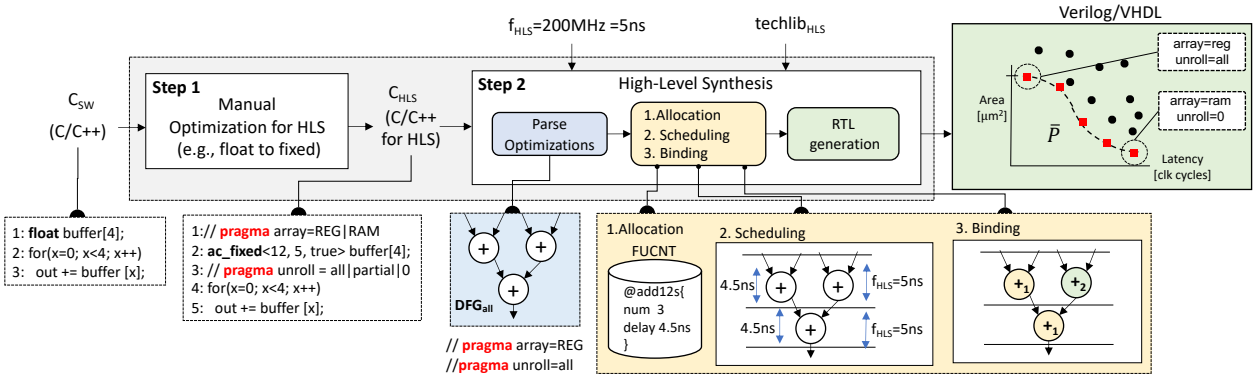

Figure 2: Overview of how to synthesize behavioral descriptions using HLS composed of two steps: Step 1: Manual refinement of C/C+ code to make it synthesizable. Step 2: HLS detail including its main phases: 1. Allocation, 2. Scheduling and 3. Binding.

Based on this, the main goal of this work is to help the HW designers find the bugs in the manually written RTL code by using the original behavioral SW description and synthesizing this describing with HLS in order to obtain a cycle-accurate RTL description that generates the golden reference outputs. This new description is then used to find bugs in the manually written RTL code. The goal is not to use HLS to build the final HW, but to enable a path to find the bugs in the manually generated RTL code. In summary, the main contributions of this work are:

- Introduce design flow that automatically generates 'similar' HW architectures from untimed behavioral description as compared to manually optimized RTL code.
- Use a golden error-free RTL description generated through HLS to find bugs in manually written RTL code.
- Present a comprehensive evaluation showing the effectiveness of our proposed flow.

2 RELATED WORK

The complexity of state-of-the-art VLSI systems is making the verification process tremendously challenging and one of the most important design cycle bottlenecks. Most HW designers still rely on test coverage through comprehensive RTL simulations [4, 5]. Some work even assumes that bugs cannot be completely eliminated at the pre-silicon stage and hence, propose online design bug detection mechanisms [6, 7].

Another direction in this domain proposes the use of formal verification techniques [8]. These include property checking and equivalence checkers and do not require any simulations as these properties are mathematically proven. Although these formal verification techniques are mainly applied at the RT-level, some EDA vendors have created C/C++ to RTL equivalence checkers like Cadence Jasper C2RTL App and Siemens EDA SLEC. The main problem with these checkers is that they only work with specific input languages subsets (e.g., synthesizable C++ and SystemC) and have severe limitation in the RTL descriptions that they support (e.g., pipelining vs. not pipelining).

To the best of our knowledge this is the first work that aims at leveraging the untimed behavioral description to find bugs in manually generated RTL code.

3 BACKGROUND

This section covers important background information required to better understand the proposed methodology. In particular it will cover the basics of HLS as the goal of this work to find a functional equivalent architecture to the manually generated RTL code with similar underlying structure and defines the main functional faults (bugs) that we consider in this work.

High-Level Synthesis: Fig. 2 shows a typical flow used to synthesize behavioral descriptions using HLS. As shown, the flow is composed of two steps.

Step 1 is done manually and involves making the C/C++ synthesizable in case that it is not. Most HLS tools do not support recursion and dynamic memory allocation. Thus, the original SW description might need to be modified to make it synthesizable. Other typical optimizations are data type refinements. SW developers usually declare their variables that require real numbers as float. Floats are nevertheless very expensive to be implemented in HW. Thus, HW designers often have to convert these floats into fixed-point data types. As shown in the code snippet in the figure, the original array in C_{SW} (buffer[4]) was originally declared as a float and is now declared as a signed fixed-point data type of 12-bits using the arbitrary precision data types(ac_fixed) [9]. This manual data type refinement is one of the main source of errors when converting pure SW description into behavioral descriptions optimized for HLS and requires extensive numerical precision analysis.

Step 2 then performs the actual HLS. As shown in the figure, the HLS process contains multiple passes. The first one is parsing the behavioral description and performing technology independent optimizations. The parser basically checks for syntactical errors and the technology independent optimization pass does typical compiler optimizations like dead-code elimination and constant propagation. The results of this stage are a Data Flow Graph (DFG) that represents the C/C++ code and its dependencies as a graph. The DFG generated depends on the code itself and also synthesis directives in the form of pragmas specified by the HLS user in the synthesizable C/C++ code.

HLS vendors make extensive use of synthesis directives in the form of pragmas to mainly control how to synthesize arrays, loops, and functions. Arrays can be synthesized as registers or RAM, loops unrolled, partially unrolled, not unrolled or pipelined and functions inlined or not. Setting different combinations of these pragmas

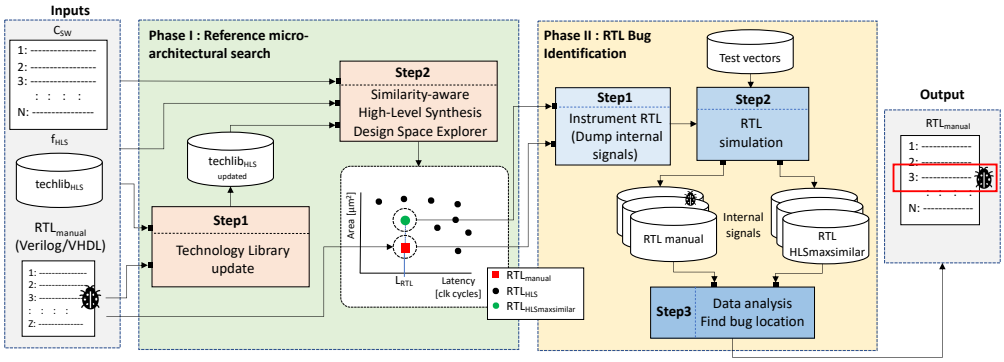

Figure 3: Overview of the complete proposed flow composed of two phases: Phase I: Reference micro-architectural search. Phase II: RTL Bug Identification.

lead to hardware implementations of different area vs. performance trade-offs. In the code snippet in Fig. 2 if the array is synthesized as registers and the loop fully unrolled, then the for loop can be fully unrolled as shown in the DFG_{all}.

This DFG is then converted into RTL through HLS's main steps: (1) Allocation, (2) Scheduling, and (3) Binding. Allocation extracts from the technology library ($techlib_{HLS}$) the basic operators required to synthesize the DFG and generates a Functional Unit Constraint File (FUCNT). In this example, the FUCNT includes three 12-bit signed adders (add12s). As shown in the figure, these adders have a delay of 4.5 ns. The scheduling step then assigns different portions of the DFG to individual clock steps based on the target synthesis frequency (f_{HLS}), in this example set to 200 MHz=5 ns, the number of functional units (FUs) specified in the FUCNT file and the delay of these FUs. In this particular case because the delay of the adder is 4.5 ns and the target HLS synthesis frequency is 5 ns, only two adders can be scheduled in a single clock step. Hence only two adders are being used as one of the two adders is re-used in the next clock cycle. Finally, the binding stage decides which additions are mapped to which adders in the FUCNT.

The back-end of the HLS process takes these results and outputs the synthesizable RTL. As shown in Fig. 2, setting different pragma combinations leads to HW implementations of different area vs. performance trade-offs. Because area and performance are conflicting design metrics there is no single HW implementation that dominates all of the others. Instead, out of all the pragma combinations there are some that lead to Pareto-optimal designs that form the Pareto frontier ($\bar{P}$). The process of finding these Pareto-optimal designs can be automated and is typically referred to as HLS design space exploration (DSE) [10]. In this particular work, our goal is not to find the Pareto-optimal designs, but to find the HW implementation that resembles the most the manually optimized RTL design such that we can exactly pin-point the location of the fault in it.

Faults (Bugs) Definitions: Multiple RTL faults have been presented in literature [11]. In this work we address the main faults introduced when manually converting a SW description into RTL that lead incorrect outputs [12]. These include: (1) Data type conversions from e.g., float to fixed-point and from larger integer bit widths to smaller precisions. (2) Logic/arithmetic operations inversions, and finally (3) Constant/variable switch.

For our flow to work we make the following assumptions that we believe are reasonable and that do not affect the generality of our approach. Assumption 1: The initial SW description (C_{SW}) produces correct results which are used as golden outputs to test the HW. Assumption 2: The SW description (C_{SW}) is synthesizable. It can still have floating point variables, but the description should be synthesizable with HLS. Assumption 3: We know the target operating frequency of the final HW and have access to the technology library.

4 PROPOSED FLOW

Fig. 3 shows an overview of the complete proposed flow. The inputs to the proposed flow are the manually written RTL code (RTL_{manual}) that has bugs in it (the simulation output does not match the golden output), the C code generated by the SW designer from where the RTL_{manual} was derived (C_{SW}) and the HLS technology library ($techlib_{HLS}$) and target synthesis frequency (f_{HLS}) which match the logic synthesis technology library and operating frequency of the target hardware.

The proposed flow is composed of two main phases. Phase I is responsible for automatically finding a micro-architecture that is as similar as possible to the manually generated one (RTL_{manual}) by performing a similarity-aware HLS design space exploration (DSE). Phase II then uses this newly generated RTL code to find the bug(s) in RTL_{manual} by comparing the simulation results of internal signals. The next subsections describe these phases in detail.

Phase I: Reference Micro-architectural Search

This first phase aims at automatically finding a hardware design ($RTL_{HLSmaxsimilar}$) that is as close as possible to the manually created HW (RTL_{manual}) (to be introduced in step 2) . For this we leverage one of the main advantages of HLS: Its ability to generate different micro-architectures from the same behavioral description by setting different synthesis options as shown in the background information section.

Even though the generated HW design from HLS is not expected to be as optimized as the manually generated one, the idea is to find a similar architecture that helps finding the bugs in RTL_{manual}. This phase is further sub-divided into two steps as follows.

Step 1: Technology Library Update: As described in the previous section and shown in Fig. 2, HLS requires the delay information of the FUs in order to accurately schedule the operations and therefore

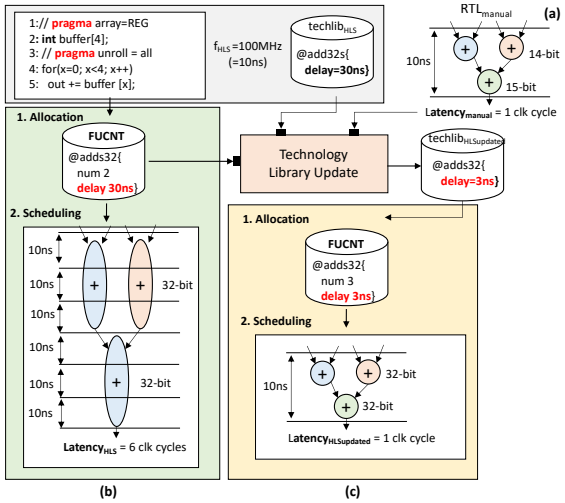

Figure 4: Technology library update overview. (a) Manually scheduled RTL code chaining three adders. (b) Regular HLS allocation and scheduling steps when variables are declared as int (32-bit). (c) Allocation and scheduling result if HLS technology library modified delay of 32-bit adder.

generate an optimized HW circuit. This information is even more critical when trying to synthesize a SW description that is not optimized for HLS like in this case (C_{SW}) as most variables might be declared as floats. This is important because floating-point FUs have much larger delays than inter FUs, and data type refinement is a traditional source of errors, and, hence, has to be included in our flow.

If C_{SW} is directly synthesized (HLS) with floats or large integers then the resultant HW circuit might never be close enough to RTL_{manual} and hence, our flow might not be able to pin-point the location of any bugs.

Thus, our flow has to consider the quantized data types in RTL_{manual}. Fig. 4 shows an example of why this is important. The figure shows the same code snippet used previously, where in this case the variables are declared as 'int'. This implies that the adders required have to be 32-bit signed adders in HLS. If f_{HLS} is set to 100MHz (10ns) and the delay of the 32-bit adder is 30ns, then HLS typically performs a multi-cycle operation (basically allows the addition to finish in 3 clock cycles). As shown in Fig. 4(b), this results in a design of latency=6 clocks steps. In contrast, the manually generated RTL code (RTL_{manual}) optimizes the bitwidth of the adders to 14 and 15-bit adders allowing them to be chained together as shown in Fig. 4(a). This leads to a circuit of latency=1 clock cycle.

This example shows that no matter what pragmas we set, the resultant HW circuit generated from HLS will never be able to replicate the manually generated design (RTL_{manual}).

To address this, in this first step our method parses RTL_{manual} and extracts the type and bitwidth of all the FUs in RTL_{manual}. It then updates the FUs in the HLS technology library ($techlib_{HLS}$) that the HLS process requires when synthesizing C_{SW}, by back-annotating the extracted FU delay from RTL_{manual}.

As shown in Fig. 4(c) the allocation stage of the HLS process is called first. This generates a FUCNT file that includes the number and delay of the FUs required. In this example it needs 32-bit signed adders. Our flow identifies that RTL_{HW} uses 14-bit adders that have a 3ns delay and hence, generates a new HLS technology library ($techlib_{HLSupdate}$) where the add32s adder now has a delay of 3ns instead of 30ns as in the original technology library. This enables the HLS process to generate the exact same schedule as in RTL_{manual}, but using 32-bit signed adders as shown in Fig. 4(c). Thus, the resultant HW still has the same numeric precision as the original SW description, while the circuit generated has the exact same structure to the manually generated one.

The obvious problem with this approach is that the generated RTL code from HLS will not meet the target synthesis frequency. This is not important as we only want to generate a functional correct circuit that is cycle-accurate for verification purpose (basically, performing logic synthesis on this RTL code will lead to timing errors).

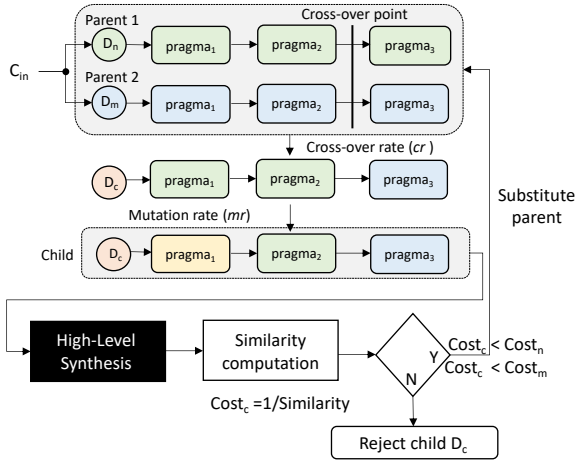

Figure 5: Genetic algorithm-based similarity-aware HLS Design Space Explorer.

Step 2: Similarity-aware HLS Design Space Explorer: Once the new technology library is generated, our flow continues by automatically finding the HW micro-architecture that is the closest to RTL_{manual}. For this we make use of a genetic algorithm as shown in Fig. 5. The explorer basically starts by generating two unique designs (parents) with two random settings of synthesis directives (pragmas) leading to $D_n = \{pragma_l, pragma_m, \ldots, pragma_n\}$ and $D_m = \{pragma_x, pragma_y, \ldots, pragma_z\}$. The genetic algorithm then chooses a random cross-over point based on a specific cross-over rate (cr) and mixes the pragmas as shown in the figure to generate a new child design (D_c). Then individual pragmas are mutated based on a given mutation rate (mr). The cr is typically a large value (90%) and the mutation rate low (20%). The explorer then synthesizes this new child and computes its cost. In normal HLS DSE the cost function is formulated as a weighted sum of the Area (A) and Latency (L) of the circuit like $C = \alpha \cdot A + \beta \cdot L$. Modifying the weights (α and β) sets more weight on either minimizing the area or the latency, and hence, allows to completely sweep the search space in order to find the Pareto-optimal designs.

In our case, we are not interested in finding the Pareto-optimal designs, but a design with the highest architectural similarity to RTL_{manual}. Thus, we formulate the cost function as $C = 1/SI$, where SI is the Similarity Index between RTL_{manual} and the RTL

generated from HLS (RTL_{HLS}). The goal of the automated explorer is to find a new design that minimizes the cost function, hence, maximizes the *SI*. Another option to speed up the search would be to build a predictive model similar to [13], although this comes with innacuracies.

If the cost of the child (D_c) is lower than any of the parents, then the explorer substitutes the parent and the search continues. If not, then D_c is ignored and a new child is generated from the two parents. The process is repeated until the cost function cannot be reduced further (in our case 10 iterations).

The question in this case is how to calculate the similarity index (SI)? For this we borrow the concept of *diversity* used in fault-tolerance. Hardware redundancy techniques are extensively used for enhancing system reliability. However, traditional identical *N*-modular redundancy (NMR) cannot protect against Common Mode Failures (CMFs). This is, when multiple faults happen at the same time in the *N*-modular redundant system. One method that has been proposed to protect against CMFs is the use of dissimilar (diverse) module redundancy [14].

The main idea is to make faults occurring in both modules at the same time visible at the outputs such that each module generates a different value. The worst-case scenario in any fault tolerant system occurs if a system has a fault and the majority of the NMR modules produce the same erroneous output. Asymmetric redundancy aims at specifically avoiding this case.

In [15] the authors proposed a fast method called Diversity Metric based on circuit Path analysis (DIMP) based on the static analysis of the different gate netlists. The idea is based on the concept that lack of diversity happens when a given input signal I_i propagates to a given output signal O_j through similar gates in both circuit instances. So, to quantify diversity, DIMP takes into account whether the same netlists are traversed in the same order from I_i to O_j in both circuit instances. To achieve this, all paths of a given circuit are taken as inputs and circuit paths timing reports are obtained as output after logic synthesis listing all paths. With this information DIMP is calculated as follows:

$$\begin{aligned} DIMP &= \sum weight_{(p^1_{i,j}, p^2_{i,j})} \cdot (1 - overlap_{(p^2_{i,j}, p^2_{i,j})}) \\ MaxDIMP &= \sum weight_{(p^1_{i,j}, p^2_{i,j})} \\ Diversity &= \frac{DIMP}{maxDIMP} \end{aligned} \tag{1}$$

In this case, overlap is the number of gates repeated across paths ($p^1_{i,j}$ and $p^2_{i,j}$) from primary inputs to primary outputs traversed in the same order, and weight is the maximum gate count of those paths. Two identical circuit instances will lead to a diversity of 0 since all of the paths are identical. The closer the diversity value is to 1, the more diverse the designs are and intuitively less susceptible to CMF.

We extend [15] in this work by applying it to RTL descriptions. Basically, our method parses RTL_{manual} and any new RTL generated by HLS (RTL_{HLS}) generating their respective DFGs. It then traverses these DFGs from inputs to outputs similar to [15], but instead of computing the dissimilarities of the paths, we compute the similarities. We call this the *Similarity Index* (SI). An SI equal to 1 means that the circuits are completely identical, while an index of 0 implies no common paths and, hence, very different circuits. The output of this first phase is the RTL description with highest SI ($RTL_{HLSmaxsimilar}$).

Phase II: RTL Bug Identification: This second phase takes as input the result from phase I ($RTL_{HLSmaxsimilar}$) and identifies where the bug(s) in RTL_{manual} are located. To achieve this, this phase is further split into three steps as follows:

Step 1: RTL instrumentation: This first step takes as input the two RTL descriptions ($RTL_{HLSmaxsimilar}$ and RTL_{manual}) and instruments these descriptions by dumping every internal signal onto a separate database. This will allow our method to investigate the dissimilarities between both designs.

Step 2: RTL Simulation: This step simulates the two instrumented RTL descriptions using the given test vectors. These test vectors are the same as the ones used in the verification of C_{SW}. The result of the simulation is a database of simulation results of every primary IO as well as internal signals in both designs.

Step 3: Data Analysis and Bug Location: This last step analyzes the data base generated by comparing the results from both RTL simulations. To compare the results we again re-use the similarity pass performed in step 2 of phase I and extract all of the paths from primary inputs to primary outputs that have high SI values (0.85 to 1). The signal values along these paths are then used to verify where the bug is located.

5 EXPERIMENTAL RESULTS

In order to test our proposed flow, we took 7 benchmarks that were available in RTL and C open-source repositories. In particular, we took 7 RTL benchmarks from OpenCores [16] to act as our manually converted RTL code (RTL_{manual}), and the equivalent same 7 benchmarks from C open-source repositories representing C_{SW}. In particular, MiBench [17] and CHStone [18]. We made sure that a common set of test vectors lead to the exact same results in both cases as the implementation of some of the benchmarks were slightly different and hence, we had to manually modify the benchmarks. We then inserted a set of typical mistakes into the RTL_{manual} benchmarks randomly like reducing the bitwidth of a functional unit (FU) (**FU bitwidth**), and also randomly modifying an RTL operation for another one (**Operation Inversion**). E.g., modify an addition by a subtraction, or an XOR by an OR operation. Finally, we also set random stuck-at-faults of individual bits (**Stuck-at-fault**). Each of these faults was inserted randomly 100 times and we count the number of faults detected.

We use NEC CyberWorkbench v.6.1 as HLS tool, Synopsys VCS 2018.06 as RTL simulator. Nangate 45nm Open source is used as technology library and a target synthesis frequency of 200MHz in all cases. The verification is performed by simulating 10,000 uniformly random test vectors. The experiments were run on a server running CentOS v.8 on an Intel Xeon E5-2603 CPU @ 1.6GHz with 32 Gbytes of RAM.

Table 1 summarizes the results including the benchmarks names and their complexity in terms of lines of code. From the results we can make the following observations:

Observation 1: Our proposed flow is able to find for all of the benchmarks a *similar* architecture from the behavioral description to RTL_{manual}, denoted by the high similarity index (SI) obtained. On average the SI=0.90 (1 being the exact same design). This is

Table 1: Experimental results inserting three different types of faults into RTL_{manual} (100 random faults of each type)

Bench	C_{SW} #lines code	RTL_{manual} #lines code	Similarity Index (SI)	FU bitwidth Faults found	Operation Inversion Faults found	Stuck-at-fault Faults found
sobel	32	194	0.98	100/100	100/100	100/100
fir	58	203	0.98	100/100	100/100	100/100
decim	75	285	0.95	100/100	100/100	93/100
dither	77	301	0.94	91/100	100/100	85/100
divider	98	305	0.75	83/100	95/100	79/100
aes	134	745	0.85	100/100	97/100	85/100
jpeg	234	815	0.87	89/100	93/100	86/100
Avg.			**0.90**	**95/100**	**98/100**	**90/100**

particularly important here because the behavioral benchmarks and RTL_{manual} were actually obtained from two different sources. In a more realistic scenario, where RTL_{manual} is obtained from C_{SW} we would expect an even higher SI.

Observation 2: On average we can find 93.5% of all the bugs in the RTL code (averaging the bugs found in all of the four categories). The lowest rate happens in the stuck-at-fault faults (86%) as these are single bit faults that cannot be detected if they are in one of the non-overlapping paths. The easiest faults to find are the operation inversion with an average of 98% of faults found followed by FU bitwidths with 95%.

Observation 3: The Similarity Index (SI) obtained is highly correlated with the faults found. A correlation of 0.74, 0.78 and 0.90 is obtained between the SI and the different fault categories. This makes sense as a low SI means that the two HW architectures are too dissimilar and prevents our method from finding the faults.

Table 2 summarizes the average runtime of our proposed flow categorized by its phase and step within each phase. The geometric average is used to account for benchmark size differences. As shown the most time-consuming part is the similarity-aware HLS DSE step as this requires the generation of multiple designs and fully synthesizing (HLS) them in order to evaluate their similarity with RTL_{manual}. On average this exploration took 3h57min for all of the benchmarks. The second most time-consuming step is the RTL simulation for RTL_{manual} and $RTL_{HLSmaxsimilarity}$. The runtime of the other steps are marginal.

Table 2: Average runtime summary based of proposed flow.

Proposed Flow		Runtime [h:m]
Phase I	Step 1: Techlib update	0h:01min
	Step 2: Similarity-aware HLS DSE	3h:57min
Phase II	Step 1: Instrument RTL	0h:01min
	Step 2: RTL simulation	0h:09min
	Step 3: Data Analysis find bug	0h:03min

Based on these results we believe that the proposed fully automated flow is very efficient in finding bugs in manually optimized RTL code when a synthesizable behavioral description is available.

6 CONCLUSION

In this work, we have introduced an automated flow to aid HW designers that manually convert untimed behavioral descriptions into optimized RTL descriptions in finding bugs. Two key ideas are used. The first is the modification of the HLS technology library in order to allow HLS to generate similar HW circuits using higher precision functional units (integer or floating-point). The second is the use of the similarity index of two RTLs by using a diversity metric introduced in the past in the context of fault-tolerance. These two key ideas are used in an automated genetic-based HLS design space explorer to automatically find HW architectures similar to the manually optimized one, which in turn allows to find the bugs.

REFERENCES

[1] B. Carrion Schaefer, *High-Level Synthesis Made Easy.* highX Technlogies, 2023. [Online]. Available: www.hlsbook.com

[2] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A survey and evaluation of fpga high-level synthesis tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, 2016.

[3] H. Foster, "Out of the Verification Crisis: Improving RTL Quality," 2021. [Online]. Available: https://www.eetimes.com/out-of-the-verification-crisis-improving-rtl-quality/

[4] M. Keaveney, A. McMahon, N. O'Keeffe, K. Keane, and J. O'Reilly, "The development of advanced verification environments using system verilog," in *IET Irish Signals and Systems Conference (ISSC 2008)*, 2008, pp. 325–330.

[5] P. Dwivedi, N. Mishra, and A. Singh-Rajput, "Assertion & functional coverage driven verification of amba advance peripheral bus protocol using system verilog," in *2021 International Conference on Advances in Electrical, Computing, Communication and Sustainable Technologies (ICAECT)*, 2021, pp. 1–6.

[6] K. Constantinides, O. Mutlu, and T. Austin, "Online design bug detection: Rtl analysis, flexible mechanisms, and evaluation," in *2008 41st IEEE/ACM International Symposium on Microarchitecture*, 2008, pp. 282–293.

[7] K. Dharsee, E. Johnson, and J. Criswell, "A software solution for hardware vulnerabilities," in *2017 IEEE Cybersecurity Development (SecDev)*, 2017, pp. 27–33.

[8] B. Alizadeh, P. Behnam, and S. Sadeghi-Kohan, "A scalable formal debugging approach with auto-correction capability based on static slicing and dynamic ranking for rtl datapath designs," *IEEE Transactions on Computers*, vol. 64, no. 6, pp. 1564–1578, 2015.

[9] Siemens EDA, "Algorithmic C (AC) Datatypes Reference Manual," 2022.

[10] B. C. Schafer and Z. Wang, "High-level synthesis design space exploration: Past, present, and future," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 2628–2639, 2020.

[11] F. Ferrandi, F. Fummi, and D. Sciuto, "Implicit test generation for behavioral vhdl models," in *Proceedings International Test Conference 1998 (IEEE Cat. No.98CH36270)*, 1998, pp. 587–596.

[12] J. Long and R. K. Brayton, "A simple c to verilog compilation procedure for hardware / software verification," 2016. [Online]. Available: https://api.semanticscholar.org/CorpusID:37142035

[13] F. N. Taher, A. Balachandran, and B. Carrion Schafer, "Learning-based diversity estimation: Leveraging the power of high-level synthesis to mitigate common-mode failure," in *2019 IEEE 37th International Conference on Computer Design (ICCD)*, 2019, pp. 460–467.

[14] S. Mitra, N. R. Saxena, and E. J. McCluskey, "A Design Diversity Metric and Analysis of Redundant Systems," *IEEE Transactions on Computers*, 2002.

[15] S. Alcaide, C. Hernandez, A. Roca, and J. Abella, "DIMP: A Low-Cost Diversity Metric Based on Circuit Path Analysis," in *Design Automation Conference (DAC)*, 2017.

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

[17] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, 2001, pp. 3–14.

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