Using LibFuzzer, KLEE, and LegUp to Validate Hardware Designs

Josué Nunes Campos[1] Maria Dalila Vieira[1] Michael Canesche[2] Ricardo S. Ferreira[2] José Augusto M. Nacif[1]
[1]Science and Technology Institute, Universidade Federal de Viçosa, Florestal, Minas Gerais, Brasil
[2]Informatics Departament, Universidade Federal de Viçosa, Viçosa, Minas Gerais, Brasil
{josue.campos, maria.d.vieira, michael.canesche, ricardo, jnacif}@ufv.br

***Abstract*—Given the existing difficulties in developing applications to the FPGA platform, High-Level Synthesis (HLS) is a promising solution to abstract the hardware implementation details. However, questions regarding the quality of the results of these tools still a problem, as well as whether they are ready for use by programmers. We present a case study using the LLVM library LibFuzzer and the execution tool KLEE in conjunction with the HLS tool LegUp to evaluate both tools in terms of their data flows, inputs, and outputs. Our results show that the LegUp HLS tool presents a Verilog code very close to the C applications with the LibFuzzer library in the KLEE tool for the generated tests with satisfactory execution times.**

***Index Terms*—FPGA, HLS, data flow, hardware validation, software simulation**

I. Introduction

Over the years, the hardware accelerators adoption has increased to meet the high demand for the use of CPUs in terms of performance and energy consumption. Hardware accelerators have become a widely-adopted solution, and the most popular platforms are Graphics Processing Units (GPUs), Field Programmable Gate Arrays (FPGAs), and Application Specific Integrated Circuits (ASICs). In this context, FPGAs focus on parallelizable operations through the pipeline, achieving competitive performance and low energy consumption.

Nevertheless, the application development for FPGAs requires hardware knowledge, even when we use OpenCL or HLS. Thus, the hardware development for FPGAs remains a challenge due to the possible gaps between the existing high-level programming environments and the demand from the programmers. Fortunately, HLS tools are regularly updated. So, we aim to exploit the current viability of these tools in the industrial scenario.

In this sense, recent research in the area shows the use of automated software testing to assess code vulnerabilities [1]. Furthermore, with the increase in the use of accelerators, the evaluation and optimization of applications for hardware have been necessary to allow programmers, in general, to develop projects with quality [2].

We present a case study to evaluate the flow of C code executed in software using Klee and LibFuzzer. Further, we compare it with the flow of the version of the same C Language code in Verilog generated by the LegUp HLS tool. We also evaluate the execution time of these codes to check the quality of the automatically generated designs.

We have organized the remaining paper as follows: in section II, we present some related work in the context of High-Level Synthesis and automated software testing. Section III shows an overview of the tools used, as well as our objectives. Section IV describes the methodology used to generate the results, and we present code for our case studies. In section V, we discuss results and, finally, in section VI, we present our conclusions and future work.

II. Related Work

The High-Level Synthesis (HLS) tools, with a focus on C and C++, aim to facilitate the implementation in accelerators such as FPGAs, allowing programmers to evaluate their applications in software and hardware as well. In this context, some works analyze the impact of applications focused on hardware [2], which measures the effect of optimizations made in OpenCL to verify the performance in FPGA. Jamieson *et al.* [3], presents a process of evaluating systems and projects in High-Performance Computing (HPC).

Yet, our work addresses the validation of the Verilog generated by the HLS LegUp tool through comparisons between inputs and outputs obtained with the KLEE. This tool has a widespread adoption to produce high-coverage tests for systems automatically. In this scenario, other works use the KLEE to deliver test entries [4]. Further, in the hardware accelerators context, Li *et al.* [5] and Chattopadhyay *et al.* [6] propose respectively GUPT and GKLEE that are frameworks to check the correctness and performance for C++ GPU programs.

Besides using the KLEE tool, we validate the RTL generated through the LLVM Library LibFuzzer, which plays a role similar to KLEE by creating input tests on top of the C application [1]. Serebryany *et al.* [7] address a tutorial using the library and AddressSanitizer, which is a memory error detector. Our paper uses these two resources to generate input tests to obtain the same outputs in C code and the generated Verilog code.

III. Background

As part of the development process, we commonly do the validation performing execution tests. Therefore, the KLEE tool has an approach that verifies the entire execution flow of a C application is made [4], allowing the programmer greater

knowledge about the functioning of his code, in addition to contributing as a bug-finding tool. On the other hand, developing applications for accelerators has been a challenge for programmers in general. In this sense, the HLS tools bring an approach to facilitate the development process. The LegUp tool allows the programmer to directly convert a program in C or C ++ to Verilog, to enable the programmer to obtain a design for FPGAs [8].

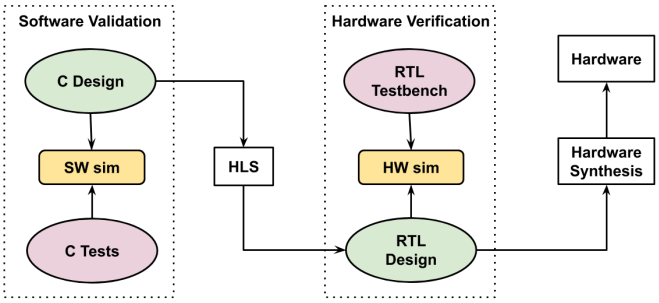

Fig. 1. High-level synthesis design steps.

Figure 1 shows the workflow of the development and verification of a Verilog module from the LegUp HLS. First, we validate the C design using a test module and the input tests. HLS tools commonly implement this functionality. If we achieve the expected results in these tests, we can do the high-level synthesis, which translates the high-level code (usually in C/C++) to RTL design. The next step consists of the RTL verification. This functionality in LegUp compares the outputs between the C and the Verilog codes. Finally, we could also do the hardware synthesis of this consistent Verilog using the HLS.

However, it is still a question whether the HLS tools are ready for programmers to use. Through this, our objective with this paper, as the Figure 1 shows, we aim to evaluate the Verilog generated by LegUp so that the data execution flows correspond to the original flows of C codes generated by KLEE and LibFuzzer.

A. KLEE

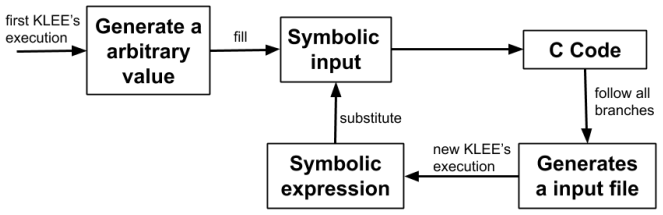

Fig. 2. KLEE execution flow.

We use the KLEE to generate inputs to our software validation and hardware verification through the symbolic execution. However, KLEE also works as a bug-finding tool. Figure 2 presents the steps of KLEE usage. This tool executes the first time allowing any arbitrary value for the target input. So, in the next steps, they substitute the current program for a new code whose inputs turn to operations that manipulate symbolic values. Therefore, this happens as the systems follow both branches for each ramification. When the tool achieves a bug or a path ends, it returns a test case with concrete values that reach the same path in a manual execution for a deterministic algorithm. For this reason, KLEE can generate high-coverage tests for complex systems [4].

B. LibFuzzer and AddressSanitizer

As an alternative to KLEE, the LibFuzzer also is an automatic generator of high-coverage tests for C programs. The LibFuzzer is coverage-guided evolutionary fuzzing differently to KLEE that define the coverage through the target code's branches. LibFuzzer stands out in the industrial scenario and is a well-established solution while the other remains in the academic research context. This tool generates the entries via a specific fuzzing entry-point, a target function that has a vector as an input parameter. However, we must use the LLVM's SanitizerCoverage instrumentation to access the coverage information of these tests. Furthermore, some of them can make random memory access or have undefined behaviors. We use the addressSanitizer(ASan) and UndefinedBehaviorSanitizer(UBSan), which are the most used in their contexts. The Asan is a memory error detector, and The UBSan is an undefined behavior detector.

C. LegUp

The LegUp HLS has a free and commercial version, both of which receive the C code as input to convert to Verilog [9]. However, the second version has support for C ++ code as well. The programmer has access to an intuitive graphical interface similar to an IDE in this version aimed at programmers developing their applications. However, the features (Table I) allow the programmer to follow the entire conversion process from the execution of the C application C to the synthesis of the Verilog code for the target FPGA. Although there are limitations regarding the structure of the C code, such as dynamic memories and recursion, the tool has advantages over other HLS tools in terms of simplicity and useful resources for the programmer to develop and evaluate their applications.

TABLE I
LEGUP COMMERCIAL VERSION COMMANDS

LegUp Command	Target	Description
init	Project	Starts a LegUp project
sw	C code	Execute C code with GCC
hw	Verilog	Generate Verilog Code
cosim	C Code/RTL	C/RTL Co-Simulation
sim	Verilog	Simulate Verilog Code
fpga	Verilog	Synthesize Verilog Code
clean	Generated files	Clean generated files

Table I presents the most important commands from the Legup commercial version. The command names are in the first row. Next, the second shows the target for each one,

meaning the object of the command action. Finally, the last row has the descriptions for these actions.

IV. Methodology

We aim to check the quality of the designs automatically generated by the LegUp HLS tool. Thus, we propose a workflow to produce inputs to any standard code and create a corresponding Verilog with LegUp. We also check the resulting RTL design correctness. Figure 3 shows this process as a whole, but it has three major independent stages: input production, software validation, and hardware verification.

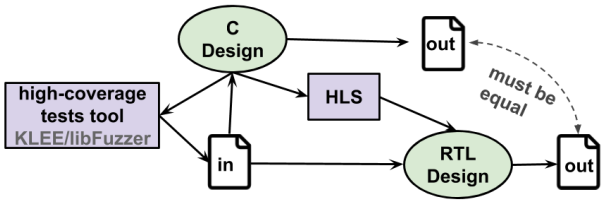

Fig. 3. Tools execution flow.

Figure 3 presents our workflow. Each transition label identifies a step. First, we use the KLEE tool and LibFuzzer to validate the C code and generate a list of inputs for it (1). Next, we execute the C design with this test entries to store the outputs (2). So, we produce the RTL design corresponding to this algorithm using the LegUp High-Level Synthesis tool (3). Finally, we check the Verilog output and compare it with the C output to validate the generated design (4). Thus, at this process ends, we can synthesize the RTL in a hardware platform.

A. Case Study: The Shellsort Algorithm

```
#include <klee/klee.h>
void shellSort(int arr[], int n){--Algorithm--}
int main() {
  int arr[5], i, in, n = 5;
  for (i=0; i<n; i++) {
    klee_make_symbolic(&in, sizeof(in), "in");
    arr[i] = in;
  }
  shellSort(arr, n);
  return 0;
}
```

Fig. 4. C Language code for KLEE.

Figure 4 shows our study case with adjustments for KLEE. We choose it because of its alternative flows, its size in lines, and the use of structures commonly used in real applications. As the 6th line shows, we must specify the target inputs with KLEE. First, we compile this code to the LLVM bytecode using the flag "clang -emit-llvm" on the clang compiler. Next, we execute the resulting bytecode using the "klee" command. So, we obtain some input files that we must read using the command "ktest-tool". Thus, we use the pipe(|) to route this command output to a standard text file. Figure 5 shows an

num objects: 2	object 1: **name:** 'in'
object 0: **name:** 'in'	object 1: **size:** 4
object 0: **size:** 4	object 1: **data:** b'\x01\x00
object 0: **data:**	\x80'
b'\x01\x00\x00\x80'	object 1: **hex :** 0x01002080
object 0: **hex :** 0x01000080	object 1: **int :** -2145386495
object 0: **int :** -2147483647	object 1: **uint:** 2149580801
object 0: **uint:** 2147483649	object 1: **text:** .. .
object 0: **text:**	

Fig. 5. Inputs generated by KLEE.

example of the inputs that KLEE generates. As our code has only integer input, we need the integer values.

Figure 6 presents our study case with adjustments for LibFuzzer. First, we must include two libraries. Next, we call our main function LLVMFuzzerTestOneInput, to receive one input vector. Further, we add (extern "C") before the return of this function. Finally, we specify restrictions to the size of the input vector or its values. To execute this code, we use the following commands:

- clang++ -g -fsanitize=signed-integer-overflow, address, fuzzer shell.c
- ./a.out -runs=10 > a

The first command compiles "shell.c" into a fuzzer binary using the LLVM. We use the sub-command "-fsanitize" to specify the instrumentation and link it to the libFuzzer library. Thus, "signed-integer-overflow" builds the target binary with the UBSan, and "address" calls the ASan. As mentioned above, UBSan detects undefined behaviors, and ASan checks memory errors. Further, the second command executes the resulting executable file. In this line, the flag "-runs" determines the number of individual test runs, which we must set to -1 to run indefinitely times.

```
#include <stdint.h>   #include <stddef.h>
void shellSort(int * arr, size_t n){--Algorithm--}

extern "C" int LLVMFuzzerTestOneInput(int *data,
    size_t size) {
  if(size <= 5)   return -1;
  shellSort(data, size);   return 0;
}
```

Fig. 6. C Language code for LibFuzzer.

We translate our C code to a Verilog module using the LegUp HLS. After that, we run a tool feature called SW/HW Co-Simulation, which performs simulation with C and Verilog codes. In this step, LegUp automatically creates the Verilog testbench module and initialize the data to the RTL design simulation according to the C code data. We perform a manual comparison with the outputs logs of both simulations to assess they are equal, meaning that we have a consistent Verilog. Moreover, we use another way to validate the generated hardware, checking that the generated Verilog is efficient at run time.

In addition to performing tests for the Shellsort algorithm, we have also run other code taken directly from GitHub that uses multiple selection structures (if/else) to validate the

generated design for a real application. We carry out the same configuration process for generating inputs and outputs.

V. Experimental Results

We execute 10 input and output tests for each algorithm, and we obtained the same data flow correspondence in all cases, that is, the code in Verilog is very close to the application in C.

Moreover, as shown in the Figure 7, it is possible to analyze the execution graph of the algorithm and observe the flow of the Verilog code generated from the C code, providing the programmer the possibility of analyzing the operation of the program also in hardware.

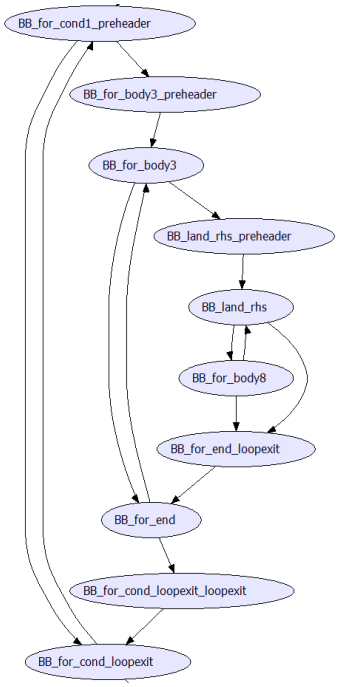

Fig. 7. Data flow of Shellsort generated by LegUp.

Table II compares the average time of the 10 executions of the generated Verilog codes and the initial C applications, showing that there is a time speedup for both codes. This indicates that the HLS tool has good results for use by programmers in general, providing resources to visualize the behavior of the generated code, as well as the flow tests of the KLEE tool and LibFuzzer library.

TABLE II
Average Execution Time of Software and Hardware Codes

Codes	C Average time (ms)	Verilog by LegUp Average time (ms)
Ext_hashfunc	0.4703	0,000126
Shellsort	0.00222	0,00166

VI. Conclusion

We do a study case that presents the workflow of the development and verification of a Verilog module from a high-level synthesis with the LegUp tool. Thus, we aim to know if the HLS tools supply the usability demand from the programmers and check the quality of the automatically generated designs.

The LegUp HLS is a well-established tool in the high-level synthesis context. Our results show that the generated Verilog is very close to the C codes with satisfactory execution times. According to our study case, a usual programmer can use this HLS tool without previous knowledge about hardware development. Further, the interface is intuitive even for command-line use. Also, we can parallel the code using the pragmas which do loop unrolling, for example, without knowing their effects on the hardware.

As future work, we aim to use KLEE and LibFuzzer to compare the data flow with other HLS tools currently available on the market to evaluate for programmers their usability and performance in real applications, just as we did in this article with LegUp HLS.

Acknowledgments

We thank LegUp Computing for providing the evaluation license for the commercial version of the tool. We also thank professor and researcher Fernando Magno Q. Pereira from UFMG for support given to automated test tools. Furthermore, we would like to thank CNPq, CAPES and FAPEMIG for the financial support and the companies Intel and NVIDIA for access to their tools.

References

[1] W.-C. Chao, S.-C. Lin, Y.-H. Chen, C.-W. Tien, and C.-Y. Huang, "Design and implement binary fuzzing based on libfuzzer," in *2018 IEEE Conference on Dependable and Secure Computing (DSC)*. IEEE, 2018, pp. 1–2.

[2] A. Sanaullah, R. Patel, and M. Herbordt, "An empirically guided optimization framework for fpga opencl," in *2018 International Conference on Field-Programmable Technology (FPT)*. IEEE, 2018, pp. 46–53.

[3] P. Jamieson, A. Sanaullah, and M. Herbordt, "Benchmarking heterogeneous hpc systems including reconfigurable fabrics: Community aspirations for ideal comparisons," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 2018, pp. 1–6.

[4] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[5] G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, and S. P. Rajan, "Gklee: Concolic verification and test generation for gpus," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 215–224. [Online]. Available: https://doi.org/10.1145/2145816.2145844

[6] S. Chattopadhyay, P. Eles, and Z. Peng, "Automated software testing of memory performance in embedded gpus," in *2014 International Conference on Embedded Software (EMSOFT)*, 2014, pp. 1–10.

[7] K. Serebryany, "Continuous fuzzing with libfuzzer and addresssanitizer," in *2016 IEEE Cybersecurity Development (SecDev)*. IEEE, 2016, pp. 157–157.

[8] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski, "Legup: high-level synthesis for fpga-based processor/accelerator systems," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, 2011, pp. 33–36.

[9] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson, "Legup: An open-source high-level synthesis tool for fpga-based processor/accelerator systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 2, pp. 1–27, 2013.