# An Endeavor to Industrialize Hardware Fuzzing: Automating NoC Verification in UVM

Ruiyang Ma[1], Huatao Zhao[2], Jiayi Huang[3], Shijian Zhang[2], and Guojie Luo[1*]
[1]School of Computer Science, Peking University, Beijing, China
[2]Computing Technology Lab, Alibaba DAMO Academy, Hangzhou, China
[3]Microelectronics Thrust, The Hong Kong University of Science and Technology (Guangzhou), Guangzhou, China

***Abstract*—We endeavor to make hardware fuzzing compatible with the standard IC development process and apply that to NoC verification in a real-world industrial environment. We systematically employ fuzzing throughout the entire NoC verification process, including router verification, network verification, and stress testing. As a case study, we apply our approach to an open-source NoC component in OpenPiton. Remarkably, our fuzzing methods automatically achieved complete code and functional coverage in the router and mesh network, and effectively detect injected starvation bugs. The evaluation results clearly demonstrate the practicability of our fuzzing approach to considerably reduce the manpower required for test case generation compared with traditional NoC verification.**

***Index Terms*—Design Verification, Hardware Fuzzing, Network on Chip, Automatic Test Generation**

## I. Introduction

Network-on-Chip (NoC) has surfaced as a crucial interconnection strategy in modern digital systems, thereby demanding meticulous verification. Due to its multiple nodes and high concurrency, verifying a NoC is labor-intensive, making it complex to generate a multitude of test cases.

Recently, hardware fuzzing has emerged as a potentially powerful automated approach for hardware verification. However, when we attempted to apply these fuzzing techniques to our internally developed NoC design, we found that they were incompatible with the industrial verification of NoC. Specifically, these techniques often target novel coverage metrics [1], [2], which are not the primary concern of industrial test plans. Moreover, they rely on open-source hardware verification workflows that offer limited support for SystemVerilog and UVM, therefore markedly deviate from traditional industrial verification environments [3], [4].

In this study, we investigated (i) how to integrate fuzzing with the UVM framework, (ii) how to apply fuzzing to a multi-port NoC design, and (iii) how to employ fuzzing to automate the completion of the test plan across all stages of NoC verification.

This work was partly supported by the Alibaba Innovative Research (AIR) Program, the National Natural Science Foundation of China (Grant No. 62090021), and the National Key R&D Program of China (Grant No. 2022YFB4500500).

E-mail: ruiyang@stu.pku.edu.cn, gluo@pku.edu.cn, hjy@hkust-gz.edu.cn, {zhaohuatao.zht, zsj269889}@alibaba-inc.com
*Corresponding author: Guojie Luo.

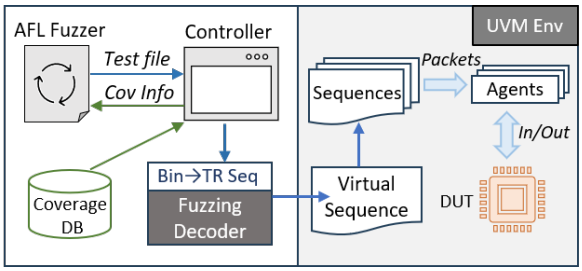


Fig. 1: UVM fuzzing framework for NoC verification.

## II. Method

### A. UVM Fuzzing Framework

Figure 1 shows the fuzzing framework. The NoC UVM environment is structured with individual agents for each NoC input port. The virtual sequence is tasked with allocating sequences to each agent [5]. We employ AFL [6] as our fuzzing engine, which mutates and generates binary-format input test files. A controller is set up to manage the entire fuzzing flow, facilitating communication with AFL. When a new binary test input is received, the fuzzing decoder translates that into a valid UVM virtual sequence in accordance with the user-defined *hardware fuzzing grammar*. Then the simulation is launched and the coverage data is sent back to AFL to guide subsequent mutation.

### B. Multi-Port Fuzzing Grammar Design

To generate valid NoC input, we first translate AFL-generated binary data into NoC instructions. Then we must devise a strategy to allocate these instructions across different ports of NoC.

We adopt a compact binary representation for our grammar, with a byte as the basic unit. An NoC Instruction consists of 1) 16-bit destination address field, 2) 8-bit packet length field, and 3) 8-bit free flag field. The free flag regulates the packet injection rate by keeping the input port active or idle. Binary test files are mapped to each instruction field, aligning them with their respective valid definition domains.

After the instructions are generated, they are sequentially assigned to the input queues of different ports following a fixed cyclical order. For instance, for a $3 \times 3$ Mesh Network, each generated instruction is systematically sent to the local input ports of routers $R_0$ to $R_8$.

979-8-3503-4859-0/DATE24/© 2024 EDAA

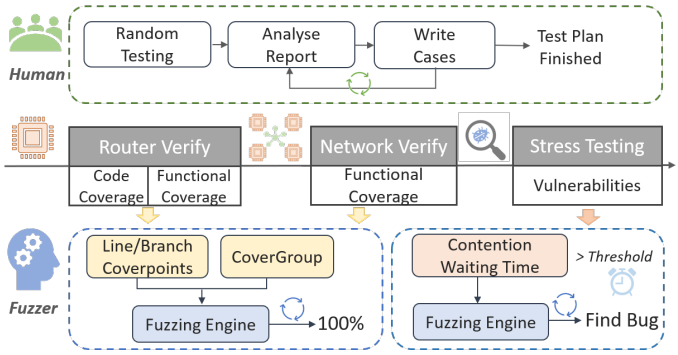


Fig. 2: Comparing manual testing and fuzzing in NoC verification.

### C. NoC Verification with Fuzzing

The verification plan for NoC can be divided into three parts, which is depicted in Figure 2.

**1) Router Verification:** Achieving full code and functional coverage of the router is essential in this stage. Unfortunately, random testing usually fails to reach numerous complex branches and functional points while writing test cases manually is both time-consuming and labor-intensive. Therefore, we utilize both code and functional coverage as feedback for fuzzing, aiming to achieve coverage closure automatically.

**2) Network Verification:** Once achieving a thorough verification of individual routers, they are interconnected for network-level testing. Given that full code coverage for each router has been achieved in the preliminary stage, it is not included in the test plan for the network. At this stage, our focus is on the network-level functional coverpoints, which are leveraged as feedback in the process of hardware fuzzing.

**3) Stress testing:** This stage involves high-pressure, long-duration testing aimed at uncovering hidden vulnerabilities. Generally, a timeout threshold is set. If a packet is not received within this threshold, a bug is triggered. Using random testing is difficult to find out overtime bugs. Therefore, we employ the *packet waiting time* as feedback for fuzzing. Mutations could progressively generate specific input sequences that lead to timeouts, thereby rapidly find the bug.

## III. Evaluation

In the evaluation part, we address the question: *Can fuzzing be utilized to reduce human or computational costs in NoC UVM verification?* We employ the NoC component from OpenPiton [7] as our DUT and utilize Synopsys VCS [8] to simulate the testbench. Previous hardware fuzzing works [1]–[4] are incompatible with our NoC UVM environment. Our objective is to adopt the fuzzing methodology into industrial verification workflows, so we benchmark against random testing, which is commonly used at the early stage of verification and requires no human intervention. If random testing fails to achieve coverage closure in the test plan, but fuzzing succeeds, then the subsequent human effort involved in writing test cases is effectively saved.

In addition to line and branch coverage for the router, we select several complex functional covergroups of the router and the $3 \times 3$ mesh network as fuzzing targets. We initiate a separate fuzzing task for each code coverage metric and functional covergroup. For vulnerability detection, we inject a starvation bug into the arbitration logic of the router. We set the overtime threshold of 1024 clock cycles. Each value of packet waiting time can also be viewed as a coverpoint in the implementation. To enhance the efficiency of AFL evolutionary algorithm, we use fuzzing seeds of shorter length. We configure 10 random initial seeds, each 40 bytes long for the router, and 72 bytes long for the $3 \times 3$ mesh network (corresponding to 2 instructions per input port). For starvation detection, we set the seed length to 100 bytes. We conduct each experiment 10 times and compute the average.

TABLE I: Performance comparison between Fuzz and Random.

| Verification Target | | Fuzz | | Random | |
|---|---|---|---|---|---|
| | | *Time (h)* | *Cov (%)* | *Time (h)* | *Cov (%)* |
| **Router** | Line | **2.6** | ***max*** | 120 | 96.7 |
| | Branch | **2.4** | ***max*** | 120 | 94.8 |
| | Covergroup1 | **3.3** | ***max*** | 120 | 98.8 |
| | Covergroup2 | **10.8** | ***max*** | 23.6 | *max* |
| **Mesh** | Covergroup1 | **9.2** | ***max*** | 120 | 99.8 |
| | Covergroup2 | **6.8** | ***max*** | 120 | 89.7 |
| **Starvation Detection** | | **0.29** | **max** | 24 | 73.3 |

A comparison of the performance of fuzzing and random testing is shown in Table I. For verification tasks challenging for random testing, fuzzing demonstrates impressive performance and achieves coverage closure.

## IV. Conclusion

This paper explores the use of fuzzing to automate the industrial NoC verification process. We developed a hardware fuzzing framework that integrates AFL into NoC UVM testbench. By designing the hardware fuzzing grammar for NoC and using verification targets at all stages of the NoC test plan as fuzzing feedback, we demonstrates the superior ability of fuzzing to generate high-quality test inputs. Moreover, it notably reduces the human labor expenditure required for writing test cases. This research indicates the practicality of industrial-grade hardware fuzzing method and its potential applicability to the industrial test plan of the DUT.

## References

[1] K. Laeufer, J. Koenig, D. Kim, J. Bachrach, and K. Sen, "Rfuzz: Coverage-directed fuzz testing of RTL on FPGAs," in *ICCAD*, 2018, pp. 1–8.

[2] J. Hur, S. Song, D. Kwon, E. Baek, J. Kim, and B. Lee, "Difuzzrtl: Differential fuzz testing to find CPU bugs," in *SSP*, 2021, pp. 1286–1303.

[3] T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks, "Fuzzing hardware like software," in *USENIX Security Symposium*, 2022, pp. 3237–3254.

[4] B. Fajardo, K. Laeufer, J. Bachrach, and K. Sen, "RTLFUZZLAB: Building a modular open-source hardware fuzzing framework," in *WOSET. https://woset-workshop.github.io/WOSET2021.html## article-10*, 2021.

[5] A. S. Eissa, M. A. Ibrahem, M. A. Elmohr, Y. Zamzam, A. El-Yamany, S. El-Ashry, M. Khamis, and A. Shalaby, "A reusable verification environment for NoC platforms using UVM," in *ICST*, 2017, pp. 239–242.

[6] american fuzzy lop, "http://lcamtuf.coredump.cx/afl/," 2018.

[7] J. Balkind, M. McKeown, Y. Fu, T. Nguyen, Y. Zhou, A. Lavrov, M. Shahrad, A. Fuchs, S. Payne, X. Liang *et al.*, "OpenPiton: An open source manycore research framework," *ACM SIGPLAN Notices*, pp. 217–232, 2016.

[8] VCS, "https://www.synopsys.com/verification/simulation/vcs.html."