Unified HW/SW Coverage:  
A Novel Metric to Boost Coverage-guided Fuzzing for Virtual Prototype based HW/SW Co-Verification

Abstract—Coverage-guided Fuzzing (CGF) has been shown to be a very effective verification technique in the Software (SW) domain. However, the application of CGF in the embedded system domain is much more limited so far. Beside the necessary integration effort of the fuzzing engine, a main limiting factor is the employed coverage metric to guide the CGF process. Since embedded systems integrate Hardware (HW) and SW parts, the coverage metric should reflect both parts instead of reasoning exclusively about the SW execution in the CGF process.

Therefore, in this paper, we propose a novel unified HW/SW coverage metric to boost state-of-the-art CGF for HW/SW co-verification. Following the modern design flow for embedded systems, we leverage a Virtual Prototype (VP) to represent the HW part. We designed effective representations of the unified HW/SW coverage to capture all relevant coverage information at run time in the VP and tailored it for integration with a modern CGF process. Our RISC-V experiments demonstrate the practical applicability of our proposed approach. Our proposed unified HW/SW coverage can be effectively managed at runtime and enables to reach deeper bugs compared to existing state-of-the-art CGF.

I. INTRODUCTION

The fast-growing Internet-of-Things (IoT) domain has unleashed a high demand for scalable and customized computing cores with rapidly changing requirements. As a reaction, Instruction Set Architectures (ISAs) are moving into the focus of the attention in the design flow. In particular, the modern free and open source RISC-V [1], [2] ISA has experienced a significant boost, as RISC-V is designed in a modular and extensible way in order to facilitate building application-specific processors.

The usual starting point for defining an Instruction Set Extension (ISE) is profiling of the Software (SW) application. The computationally most demanding segments, known as hot spots, are identified in this step. These hot spots are then analyzed in turn to identify instructions that should be optimized to boost the execution performance or reduce the power consumption of the overall system. The potential optimisations are evaluated using Instruction-Set Simulators (ISSs) as part of an Virtual Prototype (VP) in modern design flows [3], [4]. A VP is essentially an executable abstract model of the entire Hardware (HW) platform and often implemented using C++-based hardware description libraries.

However, not only rapid development methods are essential to achieve a short time-to-market, but also highly efficient verification approaches. Thanks to their scalability and user-friendliness, simulation-based verification techniques are still popular in this context. The verification approach named fuzzing enjoys extraordinary popularity and tremendous success in various application fields of the SW domain. The historic root of fuzzing goes back to [5] as a randomized test generation technique. Nowadays, state-of-the-art fuzzing techniques are guided by coverage and rely on mutation-based algorithms to generate new inputs. There are two common ways to measure coverage to guide fuzzers. One metric is to count the execution of Basic Blocks (BBs), and the other one is edge coverage, which counts the transitions between two BBs. The two most prominent and successful Coverage-guided Fuzzers (CGFs) are LLVM libFuzzer [6] and AFL [7]. In contrast to the software area, the application of fuzzing in the hardware area is still minimal. Moreover, embedded systems include HW and SW components and thus it is important to integrate coverage metrics into the fuzzing process that allow reasoning about the HW and SW execution in combination.

Contribution: In this paper, we propose a novel unified HW/SW coverage metric to enhance state-of-the-art CGF for HW/SW co-verification. Following the modern design flow for embedded systems, we leverage VPs to represent the HW part. The SW part is then executed on the VP. Both VP and SW are instrumented accordingly to collect coverage information at runtime which we combine into our proposed unified HW/SW coverage. We designed effective representations of the unified HW/SW coverage and tailored them for integration with a modern coverage-guided fuzzing process. As a case study, we leveraged the AFL fuzzer [7] as the underlying fuzzing engine and the open source RISC-V VP [8] for SW execution. Our evaluation consists of two parts. First, a performance evaluation of employing the unified HW/SW coverage based on the modern Embench benchmark set, designed specifically for embedded applications. Second, an assessment of the verification quality of unified HW/SW coverage using a verification task with a practical HW/SW example application that

978-1-6654-7332-3/22/$31.00 ©2022 IEEE
integrates an ISE at the VP-level in combination with CGF. Our experiments show that our proposed unified HW/SW coverage can be effectively managed at runtime and enables to reach deeper bugs compared to normal state-of-the-art CGF. As such unified HW/SW coverage is a practical metric to boost the HW/SW co-verification process in a VP-based design flow.

II. RELATED WORK

Fuzzing can look back at a long and successful history in the SW verification domain to the point that modern CGF-based approaches are integrated on a large scale by industry [9], [10]. The ongoing success story has sparked a strong interest in the research community to further improve fuzzing-based approaches and broaden their scope beyond verification of high-level application SW.

In the embedded SW domain several approaches to improve state-of-the-art CGF based on AFL have been proposed. The fuzzer AFL [7] itself supports the so-called QEMU mode that executes the SW binary using the emulator QEMU. In contrast to our approach, AFL with QEMU mode does not consider HW coverage and uses traditional edge coverage that has a known hash collision issue. This issue is resolved by ColiAFL [11], which is an approach to minimize the hash collision issue of AFL using multiple hash functions that are selected based on how many precedents one BB has. Another approach is PathAFL [12], which makes path coverage usable for SW fuzzing. Unlike our approach, it does not enhance the coverage granularity by considering the HW coverage but reduces the granularity of SW path coverage. Unified HW/SW Coverage is so accurate that it even considers information of every executed instruction. Also, PathAFL uses a weak hashing function for the paths, which inevitably leads to hashing collisions and reduced coverage accuracy. Moreover, these approaches do not integrate HW and SW coverage in a unified representation.

On the interface between HW and SW, fuzzing has also been leveraged to verify instruction set simulators, i.e. abstract SW models that emulate a CPU. [13] targets assembly test generation for the RISC-V ISA, guided by coverage metrics embedded in the ISS and mutations tailored for RISC-V instruction sequences. [14] employ fuzzing to generate test cases for a wide range of different simulators by relying on generic randomized techniques. However, these verification approaches target a different abstraction level than ours and only utilize the coverage of the SW model of the ISS.

Looking beyond the SW level, designated fuzzing approaches have been designed for verification at the HW level, e.g. [15], [16] targeting the register-transfer level, but pure HW verification is not the focus of our approach.

Finally, [17] proposed a CGF-based verification approach that leverages HW and SW coverage based on LLVM libFuzzer. However, the approach considers the HW and SW coverage in isolation instead of using a unified representation.

III. UNIFIED HW/SW COVERAGE-BASED VERIFICATION

In this section, we present our proposed unified HW/SW coverage and how it can be used for early SW verification in a HW/SW co-design flow. The key idea of our unified HW/SW coverage is to improve coverage measurement granularity through the enrichment of the SW coverage with the coverage of a VP that represents the HW and acts as an executable model for the SW. The enhanced coverage granularity has the goal of improving the guidance for the test generation through the fuzzer.

Our key idea resembles the target enlargement technique named virtual coverage [18] that has the goal of enhancing coverage-guided verification performance by improving the coverage granularity through inserting synthetic coverage points. However, the difference is that in opposite to virtual coverage, unified HW/SW coverage does not use synthetic coverage points but real coverage points of the VP that is used to execute the SW.

As a running example we use a RISC-V ISE. In particular, the 32bit DIV (DIV) instruction which is specified in the RISC-V multiplication/division standard ISE [1]. The DIV instruction performs a 32 by 32 bits signed division of the two source registers, rounding towards zero, and storing the result in a destination register. The semantics for the special cases of division by zero and division overflow are summarized in Table I.

A coverage metric that is only based on SW coverage like typical edge coverage does not differentiate between different cases of an instruction. In the case of the instruction DIV, a coverage metric would not differentiate if a normal division, division by zero, or a division overflow was executed, i.e.: edge coverage is pure based on branches in the SW and does not differentiate different HW execution cases of instructions.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Dividend</th>
<th>Divisor</th>
<th>DIV</th>
</tr>
</thead>
<tbody>
<tr>
<td>Division by zero</td>
<td>$x$</td>
<td>0</td>
<td>$-1$</td>
</tr>
<tr>
<td>Overflow (signed only)</td>
<td>$-2^{L-1}$</td>
<td>$-1$</td>
<td>$-2^{L-1}$</td>
</tr>
</tbody>
</table>

In the following, we give an overview of our verification approach that is using unified HW/SW coverage.

A. Overview

An overview of our approach is shown in Fig. I. Our approach starts with the C/C++ program source code of the to be verified SW, which is compiled with the aid of the LLVM-Toolchain and C/C++ libraries into LLVM-Bytecode (top of Fig. I). The LLVM-Bytecode is instrumented using our designated custom instrumentation pass to collect the coverage information (top right of Fig. I). At every start of a Basic Block (BB), our instrumentation pass adds a write instruction. The write instruction writes the BB ID to a special memory address but are directly passed by calling the Coverage Observer peripheral. The instrumented LLVM-Bytecode is compiled and linked into an executable RISC-V ELF file. The ELF file is loaded into the memory of the VP in order to execute it. The VP is instrumented like the SW. The difference is that the BB IDs are not written to a special memory address but are directly passed by calling the Coverage Observer peripheral coverage functionality. Additionally to the ELF file, a test vector is loaded into the memory of the VP. This test vector is generated by a coverage-guided Fuzzer using Mutations (bottom of Fig. I). The SW and HW coverage points are collected in the Coverage Observer during the execution. The injected SW coverage code is interpreted in the CPU Core (implemented as an instruction set simulator) that executes the write instructions using the given memory interface. The write operation triggers a TLM 2.0 write transaction that is routed by the instrumented TLM 2.0 Bus to the Coverage Observer peripheral. To prevent huge performance differences and potential endless loops, the HW coverage instrumentation calls the Coverage Observer directly. If the HW coverage instrumentation used write-transactions, HW coverage points would trigger HW coverage points in the bus by themselves. All components of the VP are instrumented to allow the measurement of the whole functionality of the VP. The
Coverage Observer peripheral combines the coverage of the VP and SW into a new fine-grained unified HW/SW coverage metric, which is used to guide the test generation process in the fuzzer (bottom of Fig. 1). Beside the coverage feedback, the fuzzer is initialized with a seed of initial test vectors in the beginning and produces a test set that maximizes the unified HW/SW coverage as the final result.

In the following Section III-B we describe the structure of our new unified HW/SW coverage metric. Afterwards, we explain in Section III-C how unified HW/SW coverage is measured dynamically. Last but not least, Section III-D describes how the fuzzer uses unified HW/SW coverage to guide the test generation.

B. Unified HW/SW Coverage

In this section we describe our novel unified HW/SW coverage metric. We illustrate the structure and properties of the metric through a step-by-step transformation of the coverage trace of our running example into a unified HW/SW coverage representation. The source code of the SW part for our running example is shown in Fig. 2 and Fig. 3 shows the corresponding SW coverage point trace. The source code uses a loop that iterates over the values $-1, 0, 1, 2$. The values are used to call the previously introduced DIV instruction (see: Table I). Hence, the loop calls the Division by zero case, the Overflow case and two times the normal case of the instruction (four times SW1). In the following, we present our idea in 3 steps.

```c
int result = 0; //SW0
for(int i = -1; i < 3;++i) {
    result += INT32_MIN / i; //SW1
}
return result; //SW2
```

Fig. 2. DIV example SW part

![Fig. 1. Overview: Unified HW/SW Coverage-based Verification Flow](image)

![Fig. 3. Generated SW coverage point trace by the DIV example SW](image)

**Step 1: Coverage Trace Splitting:** Fig. 3 shows firstly the path coverage trace and a trace that was transformed into a corresponding edge coverage.

Every coverage trace begins with a sequence of HW coverage points related to the initialization of the VP, the loading of the SW binary, and the parsing of the first instruction (see: HW0). The first SW coverage point represents the first executed BB of the SW (see: SW0). This SW coverage point is followed by a sequence of HW coverage points that belong to the executed instructions of the BB (see: HW... that represents multiple HW nodes). This sequence is terminated by the next SW coverage point representing the beginning of the following BB (see: SW1). This structure is repeated until the trace ends with a sequence of HW coverage points that represent the last executed BB and the termination of the simulation.

Directly under the unprocessed coverage trace, the diagram in Fig. 3 shows the coverage trace split using the SW coverage points as delimiters and with added frequency counters. The added counters measure, how often the edges between coverage points were traversed. It is sufficient to add the counter to the last edge of a split coverage trace because all other edges of this linear trace are executed just as frequently as the last edge. The splitting prevents scalability issues, because it reduces the coverage data size by a big magnitude. As result of the splitting, the coverage traces only represent how often a path between two SW coverage points was executed, and not the exact execution order of all SW coverage points. This coverage trace resembles the traditional edge coverage with the substantial difference, that it contains HW coverage points that were ignored until now.

Now we zoom into the HW coverage points, that were put
void ISS::exec_step() {
    ... //HW21
    switch (op) {
        ... case Opcode::DIV: {
            //HW42
            REQUIRE_Isa(M_Isa_EXT);
            auto a = regs[instr.rs1()];
            auto b = regs[instr.rs2()];
            if (b == 0) {
                //HW43
                regs[instr.rd()] = -1;
            } else if (a == REG_MIN && b == -1) {
                //HW44
                regs[instr.rd()] = a;
            } else {
                //HW45
                regs[instr.rd()] = a / b;
            } //HW46
            break;
        } ...}
}

Fig. 5. DIV Instruction Source: rv32/iss.cpp

...
C. Coverage Observer

In the following, we describe the functionality of our Coverage Observer. Algorithm 1 describes how the coverage is measured in the case a HW coverage point is hit. At the start of the simulation, the variable node is initialized with the auxiliary root node. If a HW coverage point was hit, the function covHW is called. First, the function checks if the current node is the root node. In this case, the hit HW coverage point is the first coverage point of the overall coverage trace (cf.: Fig. 8). Thereupon, a new node is created using the function getOrCreateChild, and the associated coverage point frequency counter is set to value one. Additionally, the new node is set as the current progenitor node. A progenitor node is the first coverage point node of every split coverage trace. The additional auxiliary duty of the progenitor node is to keep record of subsequently created nodes (descendants) and back edges. If the current node is not the root node, the algorithm checks if the new hit coverage point is already a known child node of the current node. If so, the next node is received from the current node using the function getChild. If the newly hit HW coverage point is not a known child of the current node, the algorithm checks if it is a known descendant of the current progenitor node. In this case, a new back edge is inserted between the current node and the descendant, and the newly created edge is saved in the progenitor node. Consequently, the current node is set to the descendant node. This just described functionality of back edge creation serves the purpose of realizing the repeating HW coverage point removal (cf.: Fig. 6). If the hit HW coverage point is neither a known child nor descendant, then a new node will be created using the function getOrCreateChild, set as current node, and saved as a descendant of the progenitor node. In the following, we describe how to handle SW coverage points.

Algorithm 1 covHW

1: procedure covHW(id)
2: if node = root
3: node ← getOrCreateChild(node, id)
4: progenitor ← node
5: else
6: if isChildOf(node, id)
7: node ← getChild(id)
8: else
9: if isDescendantOf(node, id)
10: descendant ← GetDescendant(id)
11: backedge ← AddEdge(node, descendant)
12: saveBackEdge(progenitor, backedge)
13: node ← descendant
14: else
15: node ← getOrCreateChild(node, id, HW)
16: saveDescendant(progenitor, node)
17: end if
18: end if
19: end if
20: end procedure

The Algorithm 2 function describes how a coverage trace is terminated, and coverage point hit frequencies are saved. First, this function iterates over every back edge of the progenitor node and checks if any was hit. The edge id and counter are concatenated to the beResults list if a back edge was hit. Next, the node frequency counter and the beResults are united to the variable result. Afterward, it is checked if result is already in the resultMap of the last node of the coverage trace. In this case, the corresponding counter is increased. Otherwise, the counter is set to one, and the result will be saved in the resultMap. Last but not least, the algorithm checks if the set terminals contains the last node of the trace and otherwise inserts the node to the set. The set terminals is essential because the total coverage is collected through the iteration over the terminal nodes at the end of a simulation run. At the end of the simulation, this function is called one more time because the overall coverage trace ends with a HW coverage point (cf.: Fig. 4).

Algorithm 2 covSW

1: procedure covSW(id)
2: node ← getOrCreateChild(node, id, SW)
3: terminate()
4: node ← getOrCreateChild(node, id, SW)
5: progenitor ← node
6: end procedure

Algorithm 3 terminate

1: procedure TERMINATE(node)
2: for be ∈ progenitor.backedges do
3: if be.count ≠ 0 then
4: beResults ← beResults ∪ {be.id, be.count}
5: be.count ← 0
6: end if
7: end for
8: result ← (node.counter, beResults)
9: node.counter ← 0
10: if result ∈ node.resultMap then
11: result.counter ← result.counter + 1
12: else
13: result.counter ← 1
14: node.resultMap ← node.resultMap ∪ result
15: end if
16: if node ∉ terminals then
17: terminals ← terminals ∪ node
18: node ← root
19: progenitor ← ϵ
20: end if
21: end procedure
D. Fuzzer

In the following, we describe how a fuzzer can use our unified HW/SW coverage metric to generate test vectors. First, the fuzzer connects to the VP through a network connection. The fuzzer writes the to be evaluated test vector in a file and sends the command to start the VP simulation. Thereupon, the VP forks the process and starts the simulation. After the simulation ends, the VP collects the coverage and sends it back to the fuzzer. The coverage data consists of the hit frequency information and the structure of the newly found coverage paths. Next, the fuzzer processes the coverage data using the ResultEvaluator. The ResultEvaluator checks whether new paths were created or known paths have unknown hit frequencies. Internally, the ResultEvaluator manages the hit frequency of the coverage paths using a tree structure. According to the execution result of the executed test vector, the ResultEvaluator uses a dedicated tree. Traditionally, the execution results are grouped in Queue, Crash, and Timeout. Queue describes a normal execution resulting in the return value 0 and Crash is characterized with return value $\neq 0$. The group timeout contains test vectors whose execution run time overshoot a defined run time limit. In order to keep the coverage structure consistent, the fuzzer initializes the VP coverage structure with an increment if new paths are detected. The new path coverage structure is helpful for prioritizing test vectors that disclose new coverage paths. In addition, the incremental initialization saves runtime because the whole coverage graph does not have to be created repeatedly.

IV. Evaluation

In this paper, we propose a unified HW/SW coverage metric to enhance state-of-the-art CGF verification performance in the context of HW-SW co-design to verify whole product prototypes including the Device Under Test (DUT) and Software Under Test (SUT).

In the following, we present our evaluation. The evaluation is divided into two parts. The first part is a case study with the goal of benchmarking the performance of employing unified HW/SW coverage. This case study was realized using the modern embedded benchmark suite named Embench [19]. The second part is a verification case study based on the fuzzer AFL [7], in version 2.52b, in combination with our unified HW/SW coverage.

Since AFL suffers from a known hash collision issue where two different edges could have the same hash, we modified it using a growing data structure (c.f.: CollAFL [11]). We conduct our case study, based on this repaired version of AFL, and compare AFL using unified HW/SW coverage against AFL that uses edge coverage. The implementations of both case studies are based on the open source RISC-V VP which is available at GitHub [20]. All experiments are conducted on a Linux laptop with an AMD Ryzen 7 PRO 4750U CPU and 32GB RAM.

A. Coverage Metric Benchmark

Fig. 9 shows the run time and Fig. 10 the memory results of our Embench runs. In these diagrams, we compare the execution of the original SW (original), edge coverage (edge), path-based HW/SW coverage (path, see: Fig. 8 Step 2) and our unified HW/SW coverage (HW/SW) with each other.

The average run time of the path-based coverage metric compared to unified HW/SW coverage is higher by a factor of 1.58. The

---

1This is problematic because the fuzzer can not differentiate these edges, which leads to coverage inaccuracy and restrict the verification capacity of AFL.
average memory consumption of the path-based coverage metric compared to our unified HW/SW coverage metric is higher by a factor of 1.75. Particularly striking are the results of the benchmark cubic because the memory consumption of the path-based coverage metric is so high that execution crashes because no additional memory can be allocated. Thus, we come to the conclusion that path-based coverage metrics (i.e. used by PathAFL [12]) are not usable for test generation that considers SW and HW coverage in a unified representation.

Table II shows the full results of original SW, collision free edge coverage and unified HW/SW coverage runs. For clarification, the values in the column factor (Fac.) behind a memory or time column are the factors of the difference between the last time or memory value and the corresponding value of the original SW. The column factor2 (Fac2.) is the difference between the last time or memory value and the corresponding collision free edge coverage value. The column #CovTraces contains number of unique coverage traces (see: Fig. 11). The last column named Results, contains whether the number of the unique coverage traces (#CovTraces) are matching. Traditional edge coverage does not cover the initialization and the final coverage trace of the HW. For this reason, the formula is as follows: \( \text{match} := \text{covtraces(E)} + 2 \leq \text{covtraces(V) } \). Edge Coverage has an average run time overhead of the factor 3.41 and a memory overhead of the factor 1.02. In sum, edge coverage finds 3697 unique coverage traces over 19 benchmarks. Unified HW/SW Coverage has a time overhead of the factor 6.0 in comparison to the normal SW and of the factor 1.73 in comparison to edge coverage. The average memory usage of the unified HW/SW coverage compared to the normal SW is higher by a factor of 1.46 and 1.42 compared to edge coverage. Overall, unified HW/SW coverage finds 4014.0 unique coverage traces. In 9 benchmarks, unified HW/SW coverage finds more unique coverage traces than edge coverage (see column: Result). Thus, we have shown that unified HW/SW coverage is a more granular coverage metric, on the widespread Embench benchmark set. It finds more coverage traces in 9/19 cases using the standard multiplication/division extension [1]. Additionally, we showed that our coverage metric has a low overhead and is consequently very suitable for coverage-guided verification.

B. Verification Benchmark

For our verification case study, especially in the context of HW/SW co-design using VPs, we designed a new verification benchmark inspired by typical fuzzing benchmarks. The benchmark considers the exceptional strength of fuzzers to verify branch-based targets. The benchmark consists of SW that interacts with a newly designed RISC-V Instruction Set Extension (ISE). The SW reads a 64bit long test vector into variables. The HW implementation of the ISE evaluates 32bit of the data. The result of the ISE and the other 32bit of the test vector are used in the SW. Because fuzzing is based on random mutations, a convincing fuzzing case study needs multiple runs and a statistical evaluation. Our case study uses seven fuzzing runs and the Mann-Whitney U statistical test. Mann-Whitney U is a non-parametrical statistical test suitable for small sample sizes because it does not assume a normal distribution [21]. To make the case study realistic, the fuzzing runs have a run time limit of 24 hours and use random seeds. As a corpus, we considered the 64bit long value 0x00000000.

Fig. 11 illustrates the fuzzer execution results of our case study. It allows comparisons between the executions of the state-of-the-art fuzzar AFL using collision free edge coverage (edge0-6) and AFL using our unified HW/SW coverage (HW/SW0-6). The diagram on the left shows how many test vectors, which increase the coverage, were generated over time (logarithm scale). The table on the right shows the final results of the runs. For clarification, the values in the column Count are the number of the test vectors that increase the coverage. Moreover, the column time contains the runtime needed to find the error. The rows of the runs edge0-6 show that every edge coverage-based verification run generates four test vectors. This can be attributed to the low granularity of edge coverage. This low granularity also leads to the fact that not one of the runs could find the error within the run time limit. The corresponding lines in the diagram on the left show a high variance in how fast the few test vectors could be found. The runs HW/SW0-6, generated between 91 and 105 test vectors before every run has found the error. The U-value for the generated test vectors is 0. The critical value of U at \( p < 0.01 \) is 6. Therefore, the result is significant at \( p < 0.01 \). The z-score is 3.06661. The p-value is 0.00107. Therefore, the result is significant at \( p < 0.01 \). The fuzzier needed between 1.43 hours and 4.4 hours to find the error. For the run time, the U-value is 0, the z-score is \( -3.06661 \) and the p-value is 0.00107. Therefore, the result

### TABLE II

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Original SW</th>
<th>Edge Coverage</th>
<th>Unified HW/SW Coverage</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time(s)</td>
<td>Memory(kb)</td>
<td>Fac.</td>
<td>Memory(kb)</td>
<td>Fac.</td>
</tr>
<tr>
<td>alt</td>
<td>3.38</td>
<td>115.0</td>
<td>43.0</td>
<td>1.56</td>
</tr>
<tr>
<td>cubic</td>
<td>2.37</td>
<td>3.58</td>
<td>1.55</td>
<td>1.17</td>
</tr>
<tr>
<td>cubic</td>
<td>1.68</td>
<td>87.0</td>
<td>3.58</td>
<td>2.07</td>
</tr>
<tr>
<td>cubic</td>
<td>1.17</td>
<td>80.0</td>
<td>3.4</td>
<td>2.08</td>
</tr>
<tr>
<td>cubic</td>
<td>1.17</td>
<td>80.0</td>
<td>3.4</td>
<td>2.08</td>
</tr>
</tbody>
</table>

Fig. 11 illustrates the fuzzer execution results of our case study. It allows comparisons between the executions of the state-of-the-art fuzzar AFL using collision free edge coverage (edge0-6) and AFL using our unified HW/SW coverage (HW/SW0-6). The diagram on the left shows how many test vectors, which increase the coverage, were generated over time (logarithm scale). The table on the right shows the final results of the runs. For clarification, the values in the column Count are the number of the test vectors that increase the coverage. Moreover, the column time contains the runtime needed to find the error. The rows of the runs edge0-6 show that every edge coverage-based verification run generates four test vectors. This can be attributed to the low granularity of edge coverage. This low granularity also leads to the fact that not one of the runs could find the error within the run time limit. The corresponding lines in the diagram on the left show a high variance in how fast the few test vectors could be found. The runs HW/SW0-6, generated between 91 and 105 test vectors before every run has found the error. The U-value for the generated test vectors is 0. The critical value of U at \( p < 0.01 \) is 6. Therefore, the result is significant at \( p < 0.01 \). The z-score is 3.06661. The p-value is 0.00107. Therefore, the result is significant at \( p < 0.01 \). The fuzzier needed between 1.43 hours and 4.4 hours to find the error. For the run time, the U-value is 0, the z-score is \( -3.06661 \) and the p-value is 0.00107.
is significant at $p < 0.01$. Thus, it can be seen that the results of our verification case study are statistically highly significant. The fact that every fuzzer run that uses unified HW/SW coverage finds the error demonstrates that our fine-grained coverage metric is, in comparison to edge coverage, much more suitable to guide fuzzing to perform a deeper state space exploration.

V. CONCLUSION AND FUTURE WORK

In this paper, we have proposed a novel unified HW/SW coverage metric that enhances the verification performance of CGF significantly.

In the performance benchmark experiments based on embench, we have been able to show that 1) path-based coverage is not suitable for HW/SW verification, and 2) that unified HW/SW coverage has a low-performance overhead and is more fine-grained than edge coverage. As a verification case study, we considered a practical example combination of SW and an ISE of the RISC-V RV32I ISA that is especially tailed for the characteristics of coverage-guided test generation. Our results show that a state-of-the-art fuzzer reaches deeper bugs with unified HW/SW coverage than with existing edge coverage.

In addition, we envision three extensions to improve CGF with our unified HW/SW coverage further, by boosting coverage maximization in the CGF process:

1) Initialize the unified HW/SW coverage graph using structural information that are obtained using static analysis during the instrumentation.
2) Optimize the seed selection and mutation heuristics-based on the unified HW/SW coverage.
3) Integrate with other test generation methods and bootstrapping of the fuzzer for ISE-based verification targets using test vectors obtained from preliminary tests of the unextended ISA.

ACKNOWLEDGMENTS

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project Scale4Edge under contract no. 16ME0127 and within the project VerSys under contract no. 01IW1900 and within the project ECXL.

REFERENCES