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

Niklas Bruns
Institute of Computer Science
University of Bremen
Bremen, Germany
nbruns@uni-bremen.de

Vladimir Herdt
Institute of Computer Science
University of Bremen
Cyber-Physical Systems
DFKI GmbH
Bremen, Germany
vherdt@uni-bremen.de

Rolf Drechsler
Institute of Computer Science
University of Bremen
Cyber-Physical Systems
DFKI GmbH
Bremen, Germany
drechsler@uni-bremen.de

***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 CollAFL [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 *DIVision* (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 purley based on branches in the SW and does not differentiate different HW execution cases of instructions.

TABLE I
Semantics for RISC-V division by zero and division overflow.

Condition	Dividend	Divisor	DIV
Division by zero	x	0	-1
Overflow (signed only)	-2^{L-1}	-1	-2^{L-1}

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. 1. 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. 1). The LLVM-Bytecode is instrumented using our designated custom instrumentation pass to collect the coverage information (top right of Fig. 1). 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 mapped to a designated 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. 1). 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

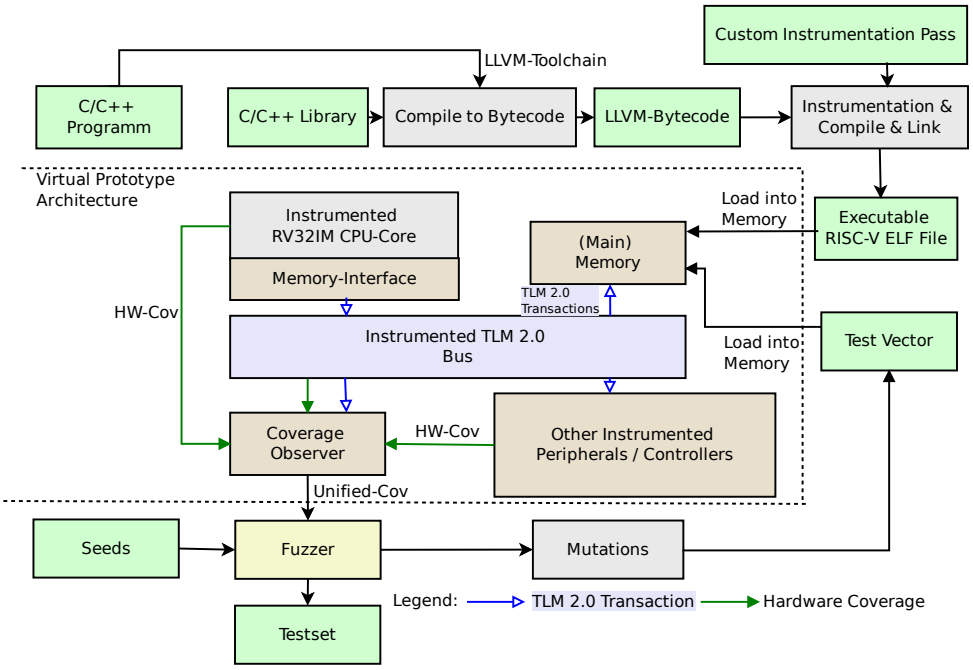


Fig. 1. Overview: Unified HW/SW Coverage-based Verification Flow

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.

```
1 int result = 0; //SW0
2 for(int i = -1; i < 3;++i) {
3   result += INT32_MIN / i; //SW1
4 }
5 return result; //SW2
```

Fig. 2. DIV example SW part

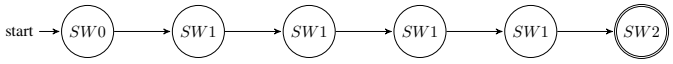


Fig. 3. Generated SW coverage point trace by the DIV example SW

Step 1: Coverage Trace Splitting: Fig. 4 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. 4 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

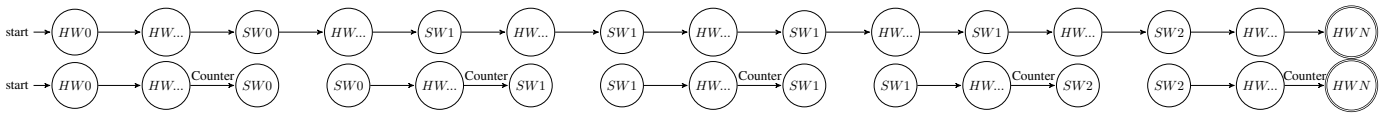


Fig. 4. HW/SW path coverage trace and the corresponding edge coverage

```
1  void ISS::exec_step() {
2    ... //HW21
3    switch (op) {
4      ...
5      case Opcode::DIV: {
6        //HW42
7        REQUIRE_ISA(M_ISA_EXT);
8        auto a = regs[instr.rs1()];
9        auto b = regs[instr.rs2()];
10       if (b == 0) {
11         //HW43
12         regs[instr.rd()] = -1;
13       } else if (a == REG_MIN && b == -1) {
14         //HW44
15         regs[instr.rd()] = a;
16       } else {
17         //HW45
18         regs[instr.rd()] = a / b;
19       }
20       //HW46
21     } break;
22     ...
23   }
24   ...
25 }
```

Fig. 5. DIV Instruction Source: rv32/iss.cpp [8]

aside until now. Fig. 5 shows the HW implementation of the DIV instruction of our running example and Fig. 8 presents the path coverage trace between the coverage points $SW0$ and $SW1$. The function *exec_step* of the VP, which has the purpose to execute the instructions of the SW, begins with a HW coverage point called $HW21$. This function contains a huge switch with cases for every supported instruction. The case for the DIV instruction begins with $HW42$. Each division case begins with a different HW coverage point:

- Division by zero: HW43
- Overflow: HW44
- Normal division: HW45

After the division, the instruction ends with HW46.

The execution of the coverage trace between $SW0$ and $SW1$ begins with the parsing of the first instruction of the BB with the SW coverage point $SW0$ and is followed by the call of the function *exec_step*. This function is called for every instruction. The execution of the division instruction starts with $HW42$, followed by $HW44$ for the Overflow case and ends with $HW46$. As you can see easily, i.e. the $SW21$ is repeated very regularly and makes the coverage trace very long. Later in the performance evaluation (see: Section IV-A), we show, why it is not reasonable to use this path-based coverage for CGF. In the next step, we optimize the coverage trace for scalilibity.

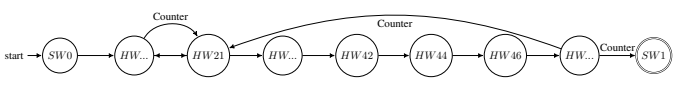


Fig. 6. SW0-SW1 Repeating Node Removal

Step 2: Removing Repeated HW Coverage Points: Fig. 6 shows the coverage trace without repeating HW coverage points. The repeated HW coverage points are removed, and a back edge is inserted. The objective of this step is to compress the coverage trace. The insertion of the back edges leads to an exception to the conjecture that only one counter per trace is needed because they lead to deviated edge transition frequencies. In order to handle this, we added additional counters at the back edges. The removal of repeating coverage points and the insertion of back edges reduce the execution order accuracy of the HW coverage points. Later in the performance evaluation (see: Section IV-A), we show, why we can not omit this step.

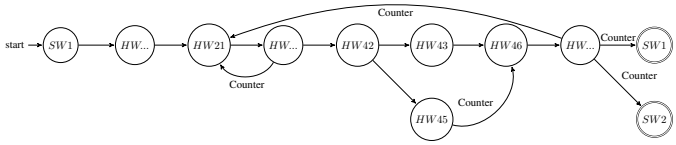


Fig. 7. SW1 to SW1 and SW2

Step 3: Merging of the Coverage Traces: Fig. 7 shows the merged coverage traces between SW1 and SW1 and between SW1 and SW2.

The difference between the two SW1-SW1 traces is, that one case is executed with the value 0 as divisor (Divison by Zero) and the other with 1 (Normal Case). As a consequence, these two traces have only the difference, that one includes $HW43$ (Division by Zero) and the other $HW45$ (Normal Case). To merge these two traces, we insert $HW45$. Because the child (46) of this node already exists in this trace, we insert a back edge as explained in step 2 *Repeating Node Removal*. Now we have merged these two traces. So the only thing missing is the trace SW1-SW2. This edge contains a normal case division that is already in the merged trace. Thus, to merge this trace into the already merged traces, we only need to add $SW2$ as leaf. In order to merge all coverage traces and not only traces with the same starting coverage point, we introduce a root node in order to create a unified graph structure. To this root node, we attach all generated traces like the traces in Fig. 7 and Fig. 6. As a result, the graph has a tree like structure with additional back edges. In the following, we label the direct child nodes of the root node as progenitors. The indirect childs of the progenitors are called descendants. The terminating childs of the tree like graph are called leafs. The leafs are the delimiting coverage points which mark the end of the coverage traces.

The transformations until now, described how a full coverage trace was transformed into unified HW/SW coverage. The advantage of this new metric in opposite to edge coverage is, that it is much more fine-grained through the enrichment with partial execution order informations and execution frequencies of the VP. In the practical usage of unified HW/SW coverage, the graph structure is not built on the base of a whole execution trace but dynamically during the run time. In the following, we describe the functionality of the Coverage Observer peripheral and, especially, how to build the unified HW/SW coverage graph dynamically.

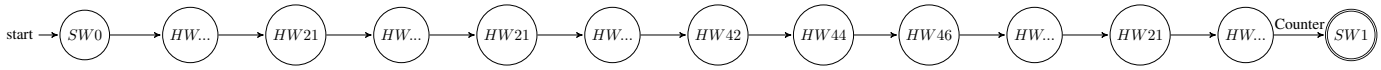


Fig. 8. DIV Instruction path coverage trace between SW0 and SW1:

C. Coverage Observer

In the following, we describe the functionality of our Coverage Observer. Algorithm 1 describes how the coverage is measured in

Algorithm 1 covHW

```
1: procedure COVHW(id)
2:     if node = root then
3:         node ← getOrCreateChild(node, id)
4:         progenitor ← node
5:     else
6:         if isChildOf(node, id) then
7:             node ← getChild(id)
8:         else
9:             if isDescendantOf(node, id) then
10:                decendant ← getDescendant(id)
11:                backedge ← AddEdge(node, decendant)
12:                saveBackEdge(progenitor, backedge)
13:                node ← decendant
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 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. 4). 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 2 describes how the coverage is measured in the case an SW coverage point is hit. If an SW coverage point is hit, the algorithm gets or creates a SW node using $getOrCreateChild$. Additionally, this function increases the associated coverage point

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
```

frequency or sets it to value one. Next, the coverage trace is terminated using the algorithm, that is shown in Algorithm 3. The current $node$ was set to the $root$ node in the $terminate$ function. It follows the repeated call of $getOrCreateChild$ to get or create the next progenitor node. Thus, the $covSW$ algorithm realizes the splitting of the coverage trace using SW coverage points as delimiters. In contrast to Algorithm 1, it is not necessary to check if a node is the root node because the first hit coverage point of a whole simulation is never a SW coverage point (cf.: Fig. 4). In the following, we describe the terminate function.

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
```

The Algorithm 3 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 $beResult$ list if a back edge was hit. Next, the node frequency counter and the $beResult$ 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).

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[1], 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

[1]This is problematic because the fuzzer can not differentiate these edges, which leads to coverage inaccuracy and restrict the verification capacity of AFL.

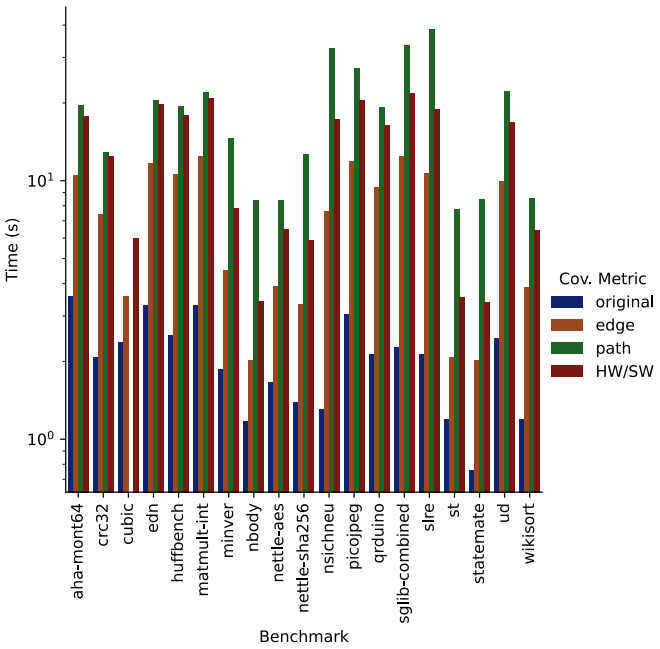


Fig. 9. *Embench* Time Results

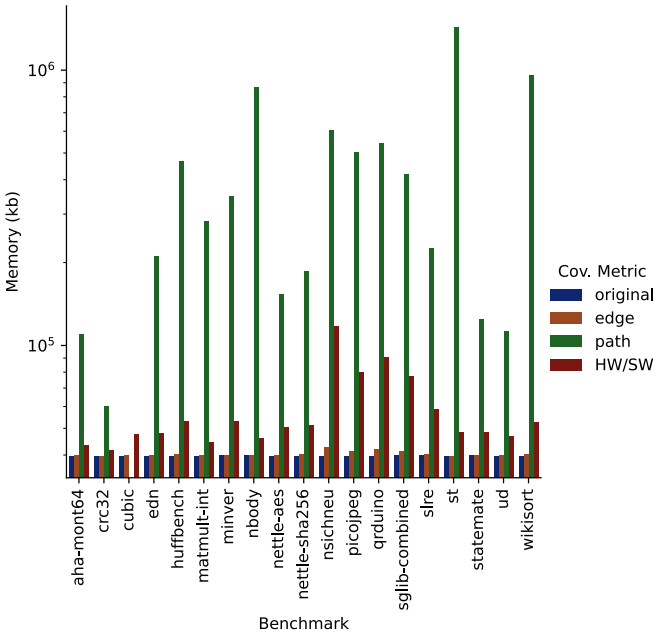


Fig. 10. *Embench* Memory Results

TABLE II
COVERAGE METRIC EVALUATION BASED ON *Embench*

Benchmark	Original.		Edge Coverage					unified HW/SW coverage							Result
	Time(s)	Memory(kb)	Time(s)	Fac.	Memory(kb)	Fac.	#Cov.Traces	Time(s)	Fac.	Fac2.	Memory(kb)	Fac.	Fac2.	#Cov.Traces	
aha-mont64	3.58	39584.0	10.48	2.93	39784.0	1.01	45.0	17.7	4.94	1.69	43412.0	1.1	1.09	47.0	Match
crc32	2.07	39680.0	7.44	3.59	39692.0	1.0	22.0	12.44	6.01	1.67	41644.0	1.05	1.05	24.0	Match
cubic	2.37	39672.0	3.57	1.51	39928.0	1.01	47.0	5.97	2.52	1.67	47608.0	1.2	1.19	89.0	Mismatch
edn	3.29	39660.0	11.71	3.56	39968.0	1.01	91.0	19.81	6.02	1.69	48212.0	1.22	1.21	93.0	Match
huffbench	2.52	39772.0	10.63	4.22	40120.0	1.01	151.0	18.0	7.14	1.69	53012.0	1.33	1.32	153.0	Match
matmult-int	3.31	39732.0	12.4	3.75	39892.0	1.0	54.0	20.78	6.28	1.68	44520.0	1.12	1.12	56.0	Match
minver	1.87	39780.0	4.52	2.42	40084.0	1.01	125.0	7.81	4.18	1.73	53044.0	1.33	1.32	156.0	Mismatch
nbody	1.17	39784.0	2.02	1.73	39896.0	1.0	48.0	3.41	2.91	1.69	46084.0	1.16	1.16	87.0	Mismatch
nettle-aes	1.66	39772.0	3.9	2.35	40048.0	1.01	115.0	6.5	3.92	1.67	50512.0	1.27	1.26	117.0	Match
nettle-sha256	1.39	39624.0	3.34	2.4	40140.0	1.01	113.0	5.89	4.24	1.76	51480.0	1.3	1.28	115.0	Match
nsichneu	1.31	39676.0	7.65	5.84	42704.0	1.08	789.0	17.25	13.17	2.25	117276.0	2.96	2.75	791.0	Match
picojpeg	3.05	39580.0	11.93	3.91	41220.0	1.04	464.0	20.44	6.7	1.71	80028.0	2.02	1.94	484.0	Mismatch
qrduino	2.13	39728.0	9.46	4.44	41932.0	1.06	593.0	16.42	7.71	1.74	90600.0	2.28	2.16	606.0	Mismatch
sglib-combined	2.26	39788.0	12.46	5.51	41332.0	1.04	461.0	21.82	9.65	1.75	77080.0	1.94	1.86	464.0	Mismatch
slre	2.13	39840.0	10.74	5.04	40372.0	1.01	220.0	18.96	8.9	1.77	58668.0	1.47	1.45	225.0	Mismatch
st	1.2	39748.0	2.08	1.73	39760.0	1.0	47.0	3.54	2.95	1.7	48364.0	1.22	1.22	153.0	Mismatch
statemate	0.76	39804.0	2.02	2.66	39932.0	1.0	87.0	3.4	4.47	1.68	48248.0	1.21	1.21	89.0	Match
ud	2.45	39724.0	9.98	4.07	40056.0	1.01	78.0	16.89	6.89	1.69	46780.0	1.18	1.17	80.0	Match
wikisort	1.2	39772.0	3.87	3.22	40204.0	1.01	147.0	6.43	5.36	1.66	52440.0	1.32	1.3	185.0	Mismatch
sum	39.72	754720.0	140.2	64.88	767064.0	19.32	3697.0	243.46	113.96	32.89	1099012.0	27.68	27.06	4014.0	
mean	2.09	39722.11	7.38	3.41	40371.79	1.02	194.58	12.81	6.0	1.73	57842.74	1.46	1.42	211.26	
median	2.13	39732.0	7.65	3.56	40056.0	1.01	113.0	16.42	6.01	1.69	50512.0	1.27	1.26	117.0	

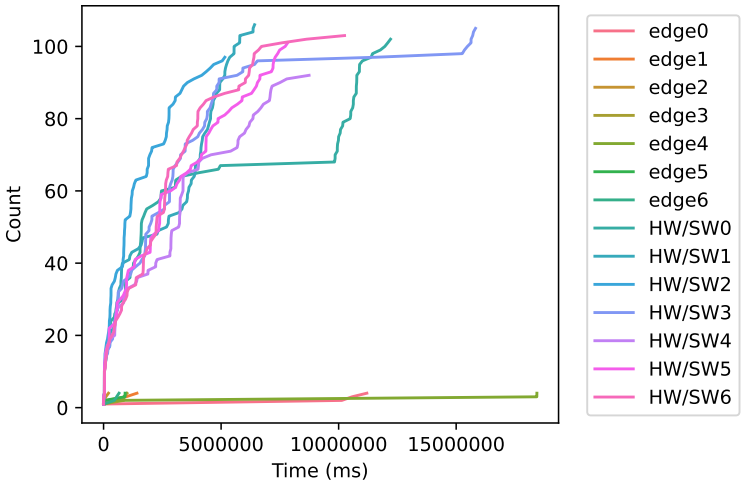
average memory consumption of the path-based coverage metric compared to our unified HW/SW coverage metric is higher by a factor of 1.75. Particularly striking are the results of the benchmark cubic because the memory consumption of the path-based coverage metric is so high that execution crashes because 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. 6). 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:: $match := covtraces(EDGE) + 2 == covtraces(VP)$. 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 overall 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 fuzzer 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 fuzzer 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



Run	Count	Time (ms)
edge0	4	Timeout
edge1	4	Timeout
edge2	4	Timeout
edge3	4	Timeout
edge4	4	Timeout
edge5	4	Timeout
edge6	4	Timeout
HW/SW0	101	12207030.0
HW/SW1	105	6423712.0
HW/SW2	96	5158201.0
HW/SW3	104	15822805.0
HW/SW4	91	8742939.0
HW/SW5	100	7832289.0
HW/SW6	102	10251573.0

Fig. 11. Verification Results

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 perliminary 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

[1] A. Waterman and K. Asanović, Eds., *The RISC-V Instruction Set Manual; Volume I: Unprivileged ISA*, 2019.
[2] ——, *The RISC-V Instruction Set Manual; Volume II: Privileged Architecture*, 2019.
[3] T. De Schutter, *Better Software. Faster!: Best Practices in Virtual Prototyping*. Synopsys Press, March 2014.
[4] L. Moore, D. Graham, S. Davidmann, and F. Rosa, "Cycle approximate simulation of RISC-V processors," in *Embedded World Conference*, 2018.
[5] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, pp. 32–44, 1990.
[6] "libFuzzer - a library for coverage-guided fuzz testing," https://llvm.org/docs/LibFuzzer.html, 2018.
[7] "american fuzzy lop," http://lcamtuf.coredump.cx/afl/, 2018.
[8] V. Herdt, D. Große, H. M. Le, and R. Drechsler, "Extensible and configurable RISC-V based virtual prototype," in *FDL*, 2018.
[9] "Oss-fuzz - continuous fuzzing for open source software," https://github.com/google/oss-fuzz, 2018.
[10] "Microsoft security development lifecycle," https://www.microsoft.com/en-us/sdl/process/verification.aspx, 2018.
[11] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," in *IEEE SP*, 2018, pp. 679–696.
[12] S. Yan, C. Wu, H. Li, W. Shao, and C. Jia, "Pathafl: Path-coverage assisted fuzzing," in *ASIA-CCS*, 2020, pp. 598–609.
[13] V. Herdt, D. Große, H. M. Le, and R. Drechsler, "Verifying instruction set simulators using coverage-guided fuzzing," in *DATE*, 2019, pp. 360–365.
[14] L. Martignoni, R. Paleari, G. F. Roglia, and D. Bruschi, "Testing CPU emulators," in *ISSTA*, 2009, pp. 261–272.
[15] 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.
[16] S. Canakci, L. Delshadtehrani, F. Eris, M. B. Taylor, M. Egele, and A. Joshi, "Directfuzz: Automated test generation for rtl designs using directed graybox fuzzing," in *DAC*, 2021, pp. 529–534.
[17] V. Herdt, D. Große, J. Wloka, T. Güneysu, and R. Drechsler, "Verification of embedded binaries using coverage-guided fuzzing with systemc-based virtual prototypes," in *GLSVLSI*, ser. GLSVLSI '20, 2020, p. 101–106.
[18] L. Fournier and A. Ziv, "Using virtual coverage to hit hard-to-reach events," in *Hardware and Software: Verification and Testing*. Springer Berlin Heidelberg, 2008, pp. 104–119.
[19] D. Patterson, J. Bennett, C. G. P. Dabbelt, G. Madhusudan, and T. Mudge, "Embench™: A modern embedded benchmark suite," 2020.
[20] "Risc-v based virtual prototype (vp)," https://github.com/agra-uni-bremen/riscv-vp/, 2018.
[21] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *SIGSAC*, 2018, pp. 2123–2138.