Received September 30, 2021, accepted November 16, 2021, date of publication November 29, 2021,
date of current version December 10, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3131213

Software/Hardware Co-Verification for Custom Instruction Set Processors

MARIE-CHRISTINE JAKOBS[1], FELIX PAUCK[2], MARCO PLATZNER[2], (Senior Member, IEEE), HEIKE WEHRHEIM[3], AND TOBIAS WIERSEMA[2]
[1]Department of Computer Science, Technical University of Darmstadt, 64289 Darmstadt, Germany
[2]Department of Computer Science, Paderborn University, 33098 Paderborn, Germany
[3]Department of Computer Science, Universität Oldenburg, 26129 Oldenburg, Germany

Corresponding author: Tobias Wiersema (tobias.wiersema@upb.de)

This work was supported in part by the German Research Foundation (DFG) within the Collaborative Research Centre 901 ''On-The-Fly Computing'' under Project 160364472, and in part by the computing time provided by the Paderborn Center for Parallel Computing (PC^2).

ABSTRACT Verification of software and processor hardware usually proceeds separately, software analysis relying on the correctness of processors executing machine instructions. This assumption is valid as long as the software runs on standard CPUs that have been extensively validated and are in wide use. However, for processors exploiting custom instruction set extensions to meet performance and energy constraints the validation might be less extensive, challenging the correctness assumption. In this paper we present a novel formal approach for hardware/software co-verification targeting processors with custom instruction set extensions. We detail two different approaches for checking whether the hardware fulfills the requirements expected by the software analysis. The approaches are designed to explore a trade-off between generality of the verification and computational effort. Then, we describe the integration of software and hardware analyses for both techniques and describe a fully automated tool chain implementing the approaches. Finally, we demonstrate and compare the two approaches on example source code with custom instructions, using state-of-the-art software analysis and hardware verification techniques.

INDEX TERMS Software analysis, abstract interpretation, custom instruction, hardware verification.

I. INTRODUCTION

Today, software verification has reached industrial size code, and annual competitions on software verification [1] demonstrate the continuing progress. This success is due to recent advances in the software analysis techniques themselves as well as in the underlying SMT (Satisfiability modulo theories) solvers [2], [3]. In general, software analyses rely on the correctness of the processor hardware executing the program. More specifically, strongest postcondition computation used to determine the successor state of a given state for a program statement assumes that the processor correctly implements the statement's semantics.

The assumption of correct hardware is certainly valid for standard processors, since they undergo extensive simulation, testing and partly also formal verification processes [4]. However, during the last years processors with so-called custom instruction set extensions became popular [5], [6], which challenge this correctness assumption. Customized instructions map a part of an application's data flow graph to specialized functional units in the processor pipeline in order to improve performance and/or energy-efficiency [6].

In this paper, we present a novel formal approach for software/hardware co-verification, in particular for processors with custom instruction set extensions. From the software analysis, we derive requirements on the hardware, which then need to be validated in order for the software analysis to produce trustworthy results.

We detail two different approaches for integrating software and hardware analyses that differ in what needs to be verified on the hardware side. Our first approach proves behavioral equivalence between the specification and the implementation of a custom instruction, e.g., that an integer adder is actually adding integer values. While proving equivalence is potentially the most runtime-consuming approach, it is also the most powerful, as it inherently covers all behavioral properties of the custom instruction on which software analyses could rely. Our second approach ties together software and

The associate editor coordinating the review of this manuscript and approving it for publication was Roberto Nardone.

hardware analyses more closely by exploiting the abstract state space of the program generated during verification to identify the specific properties of the individual program statements the software analysis has actually used during verification. These properties become *requirements* on the hardware. We thereby tailor the hardware verification exactly to the needs of the software analysis, hoping to avoid unnecessarily complex and runtime-consuming hardware verification.

In summary, our paper makes the following contributions:

- We present two approaches for software/hardware co-verification that differ in their level of integration between software and hardware analyses and potentially allow for a trade-off between generality and computational effort.
- We describe a tool chain automating all steps required for the two co-verification approaches.
- We evaluate, discuss, and compare our two approaches based on a large set of case studies.

We have presented initial ideas on our co-verification approach in [7]. This paper extends previous work by an elaborate description of the different steps of our methods, a presentation of a tool chain providing full automation of all steps, and a significantly extended evaluation based on 14 custom instructions and 7 analyses on 244 case studies.

The paper is structured as follows: section II introduces to basics on custom instruction set extension as well as on software and hardware analyses. Then, section III discusses the interface between software and hardware analyses. We present our tool chain in section IV. Sections V and VI report on our experimental setup and discuss results of experiments. Finally, section VII reviews related work and section VIII concludes the paper.

II. BACKGROUND

In this section, we provide background information for custom instruction set extension as well as software and hardware analyses techniques employed in this paper.

A. CUSTOM INSTRUCTION SET EXTENSION

The motivation for customizing instruction sets is to improve processor performance and/or energy-efficiency, while keeping the cost as low as possible [8]. There are several approaches to custom instruction set extension. The original static approach analyzes a set of targeted applications to identify runtime intense portions of the applications' data flow graphs. These subgraphs are then turned into custom instructions and mapped to specialized hardware in form of functional units (FU) accelerating the code. These specialized FUs are then integrated into a processor pipeline and a so-called application-specific instruction set processor (ASIP) is being fabricated.

Since the cost of designing a new processor is immense, the dynamic approach to instruction set extension proposes a flexible interface between the processor pipeline and a runtime reconfigurable fabric added as reconfigurable functional unit (RFU) to a processor pipeline. Runtime reconfigurability helps not only lower design cost but also increases flexibility, because the reconfigurable fabric can accommodate different custom instructions that can be switched on demand during runtime. While typically the RFUs are programmed with pre-generated configurations, the most sophisticated approaches currently studied in research, such as Dynamic Instruction Merging (DIM) [9], even shift the tasks of identifying and generating custom instructions to runtime with the goal to achieve transparent just-in-time acceleration.

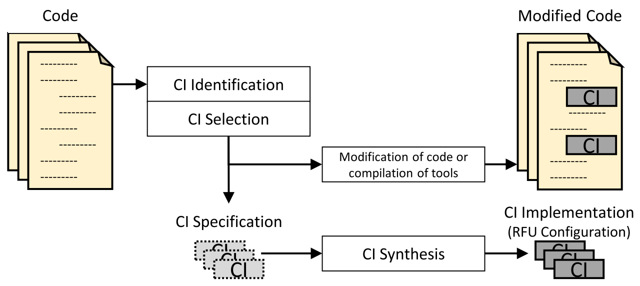


FIGURE 1. Design process for custom instruction (CI) set extensions.

Figure 1 displays the design process for custom instruction set extensions. Based on an analysis of application code, potential custom instructions are identified by exploring the design space. In a second step, the most promising custom instructions are selected applying cost functions. Then, custom instruction synthesis is used to generate the configurations for the RFUs and the code is modified to include the custom instructions. The custom instruction set extension problem is well-studied, for a general survey see [5] and for a recent one highlighting the potential benefits in performance and energy consumption see [6]; current examples of processor architectures with runtime reconfigurable RFUs can be found, e.g, in [10]–[12].

In this paper, we do not consider the impact that introducing such dynamic approaches to custom instruction set extensions can have on the non-functional properties of a processor, such as its energy-efficiency, but rather focus on verifying the functional correctness of the extensions. The effort that can be spent for these verifications in such dynamic scenarios is presumably much lower than for standard processor designs; this issue is particularly emphasized for just-in-time acceleration. It has to be noted that the circuit structures of the underlying reconfigurable fabrics, e.g., the RFUs, are indeed well-tested. What creates the verification challenge is the correctness of the RFU configurations shown in Figure 1.

Throughout this paper, we use the program `FKT` shown in Figure 2 as running example. The program employs $z = (x_1 * x_2) + x_3$ as custom instruction implemented in hardware, combining two arithmetic operations into one instruction. For our experimental evaluation we have chosen a range of different custom instructions, many combining base instructions, as in the case of `FKT`, but some also

```
1  N=100;
2  x=0;
3  while(x<=N){
4    y=2*x+8;
5    printf("f(%d)=%d\n",x,y);
6    assert(y!=0 && y<=900);
7    printf("Inverted: %f",1/y);
8    x+=10;
9  }
```

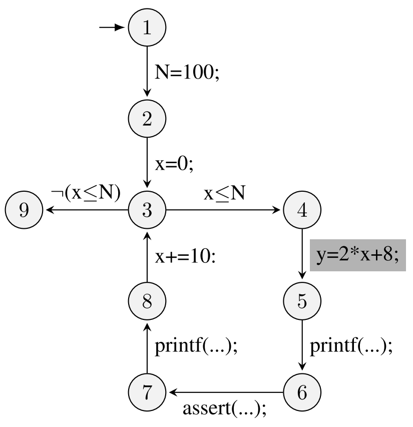


FIGURE 2. **Example program** `FKT` **(left) and its control-flow automation (right).**

offering special functionality, such as using saturation instead of modular arithmetic, so results do not overflow and wrap around, but instead saturate at the extreme values of the range. Saturating arithmetic is often used in signal or image processing and thus can be found in instruction set extensions such as Intel's SSE2 (streaming SIMD extensions).

B. SOFTWARE ANALYSIS

The programs that we consider in our software analyses are written in C. Following the notation of Beyer *et al.* [13], we model a program as a *control-flow automaton* (CFA). A CFA $P = (L, G, l_0)$ consists of a set L of locations, a set of control-flow edges $G \subseteq L \times Ops \times L$ and a program entry location $l_0 \in L$. The set Ops contains all operations, e.g., assignments, function calls and (negated) conditions[1] such as those arising of an if or while statement. Assert statements are used to encode properties to be checked by analyzers. For printing outputs the C `printf` statement is used.

The left of Figure 2 shows our example program `FKT` given in the C programming language. All variables (i.e., `y`, `x` and `N`) are of type `int`. The program shows the calculation of a value table for the function $f(x) = 2 * x + 8$ and its inverse $\frac{1}{f(x)}$ in the range 0 to 100 with step size 10. The assertion in line 6 specifies y to not be zero (such that the following division will not cause a fault because of dividing by zero) and to be less than 900 (for ensuring the value to not exceed a certain range, e.g., for drawing). This is the property to be verified by the software analysis.

The CFA on the right of Figure 2 depicts the program in the form of a control-flow automation. We see that the condition in the while statement has led to two edges in the CFA, one for the condition evaluating to true and one for evaluating to false. The latter is labeled with the negated condition. The statement framed in gray is the location of a custom instruction usage: This statement is executed several times and a specific CI design for statements of the form $z = (x_1 * x_2) + x_3$ could be employed for executing $y = 2 * x + 8$ (using 2 for x_1, x for x_2, 8 for x_3 and y for z).

The software analyses that we perform on the programs are all based on the idea of *abstract interpretation*. Instead of exploring the complete state space of the programs, we only generate the set of states on a specific level of abstraction, called the abstract domain or *analysis domain*. This level fixes what we are interested in with respect to property checking. All our analyses are specified in the *Configurable Program Analysis* (CPA) framework [13] and are performed using the associated tool CPAchecker[2] [14]. The framework allows for the definition of arbitrary abstract interpretation based analyses, which vary in their abstract domain and applied analysis technique. The supported analysis techniques range from dataflow analyses to model checking. To specify such an analysis, we need to configure the analysis technique and define the analysis domain as well as the semantics of program operations on this domain. The semantics is given in terms of a *transfer relation*. In order to provide a sound analysis, the abstract domain and the transfer relation need to provide an overapproximation of the concrete program execution semantics.

In the following, we describe the concept of abstract interpretation on the example of cartesian predicate abstraction [15]. In cartesian predicate abstraction, the abstract domain consists of (conjunctions of) predicates on program variables, like $x \geq 0$. The predicates are often incrementally determined using counterexample-guided abstraction refinement (CEGAR) [16]. For a given set of predicates $P = \{p_1, \ldots, p_n\}$, the transfer relation is determined by (1) computing the strongest postcondition [17] of the currently holding predicates $R \subseteq P$ with respect to the operation op, i.e. $post = sp(op, \bigwedge_{r \in R} r)$, and (2) finding the set of predicates best approximating the strongest postcondition $post$ within in the available predicates P, i.e. $\bigwedge_{p \in P, post \Rightarrow p} p$. Function calls like `printf` or `assert` do not influence predicates. Predicates can be arbitrarily weakened without losing soundness of the analysis. The analysis then computes the abstract state space of the program using predicate *true* for the initial location and determines new states according

[1]Negated conditions start with the negation symbol ¬.

[2]http://cpachecker.sosy-lab.org

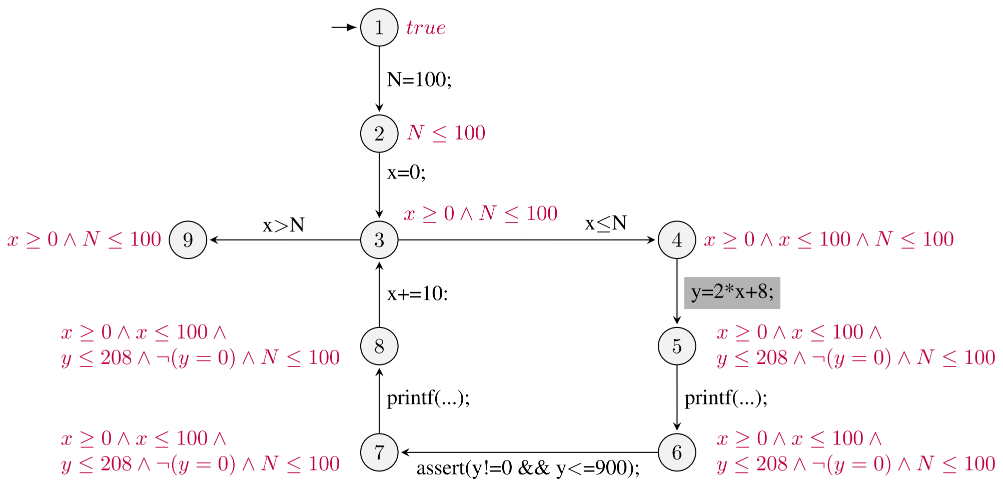

1 true
N=100;
2 $N \leq 100$
x=0;
3 $x \geq 0 \wedge N \leq 100$
x>N
9 $x \geq 0 \wedge N \leq 100$
x≤N
4 $x \geq 0 \wedge x \leq 100 \wedge N \leq 100$
y=2*x+8;
5 $x \geq 0 \wedge x \leq 100 \wedge y \leq 208 \wedge \neg(y = 0) \wedge N \leq 100$
printf(...);
6 $x \geq 0 \wedge x \leq 100 \wedge y \leq 208 \wedge \neg(y = 0) \wedge N \leq 100$
assert(y!=0 && y<=900);
7 $x \geq 0 \wedge x \leq 100 \wedge y \leq 208 \wedge \neg(y = 0) \wedge N \leq 100$
printf(...);
8 $x \geq 0 \wedge x \leq 100 \wedge y \leq 208 \wedge \neg(y = 0) \wedge N \leq 100$
x+=10:

FIGURE 3. **Abstract state space of program** `FKT` **using predicate abstraction.**

to the CFA and the statements on its control-flow edges. The computation of the transfer relation is done with the help of SMT solvers. The precision of the computed abstract state space depends on the analysis technique, which may or may not merge different abstract states during state space construction.

As an example consider the abstract state space of program `FKT` as given in Figure 3 in the form of an *abstract reachability graph* (ARG). The set of predicates used in the analysis is

$$P = \{x \geq 0, x \leq 100, N \leq 100, \neg(y = 0), y \leq 208\}$$

The analysis determines the predicate $N \leq 100$ and $x \geq 0$ to hold after the initial assignments to N and x, respectively, and then throughout the entire remaining program. When entering the loop, this gives us $x \leq 100$ which enables the analysis to deduce predicate $\neg(y = 0)$, i.e., y is not 0, and predicate $y \leq 208$ after the custom instruction, and thus that the assertion can never fail. The result of the software analysis on `FKT` thus yields the result TRUE (property as encoded in assertion holds).

1) COMPLEXITY OF THE ANALYSES

In general, all sorts of non-trivial problems in software analysis are undecidable due to Rice's theorem. Analyses therefore concentrate on over- or underapproximating a certain analysis problem instead of computing an exact solution. In our experiments, we will employ overapproximating analyses that rely on abstract interpretation and analysis techniques like dataflow analyses and model checking. Some of these analyses – for instance analyses based on predicate abstraction – involve CEGAR loops in which the level of abstraction of the analysis is incrementally determined. These techniques might actually not terminate on some programs, thereby manifesting their undecidability. For analyses that employ the dataflow analysis technique, termination depends on the analysis' abstract domain. Domains that consider lattices with infinite heights like the interval abstraction we use require widening operators in order to ensure termination for arbitrary programs. Finally, our simplest analyses (analyses based on sign abstraction) are polynomial in the size of the program. Since the complexity is highly dependent on the program at hand and its requirement, we decided to study the analysis performance experimentally and make the comparison of our approaches on case studies.

C. HARDWARE ANALYSIS

The task of the hardware analysis within this work is to formally verify the validity of certain correctness requirements for custom instructions. These requirements are used by the software analysis implicitly or explicitly during the analysis. We aim for automated or automatable processes and thus employ model checking for this purpose, and we therefore require the model of the custom instruction depicted in Figure 4 (i.e., the verification environment), in order to apply the methods. Figure 4 depicts the general structure of our hardware analysis model for a combinational custom instruction circuit. As verification environment, we expand the implementation of the custom instruction, i.e., its RFU configuration, $I(\underline{in})$ with a property checking circuit $P(\underline{in}, \underline{out})$, where $\underline{in}$ is the set of primary inputs of the custom instruction and $\underline{out}$ the set of primary outputs, respectively. The output of the property checker is an error flag $error = P(\underline{in}, \underline{out}) = P(\underline{in}, I(\underline{in}))$, which is set iff the properties encoded in $P(\underline{in}, \underline{out})$ are violated for the given input stimuli $\underline{in}$. To show that the encoded properties for the implementation of the custom instruction actually hold, it is thus sufficient to prove that the error flag is never set under all possible input stimuli. Our main tasks are thus to encode the requirements posed by the software analysis into a suitable property checker, i.e., into a circuit description in a hardware description language (HDL) such as Verilog, and to prove the unsatisfiability of the resulting model. While the details of the verification model depend on the choice between the approaches we will present in section III, its general structure is always as depicted in Figure 4.

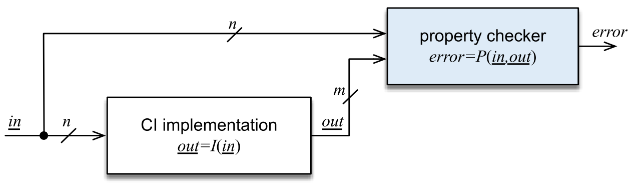


FIGURE 4. **General structure for the hardware analysis. A property checker encodes the requirements posed by software analysis for the implementation of the custom instruction (CI) with *n* bits input and *m* bits output.**

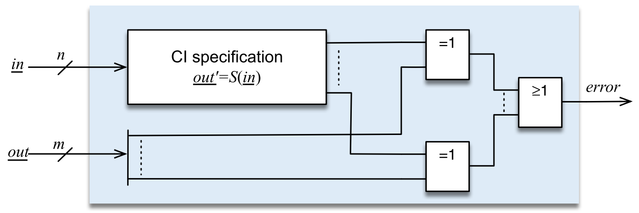


FIGURE 5. **Property checker for proving functional equivalence between an implementation of a custom instruction and its specification, both with *n* bits input and *m* bits output.**

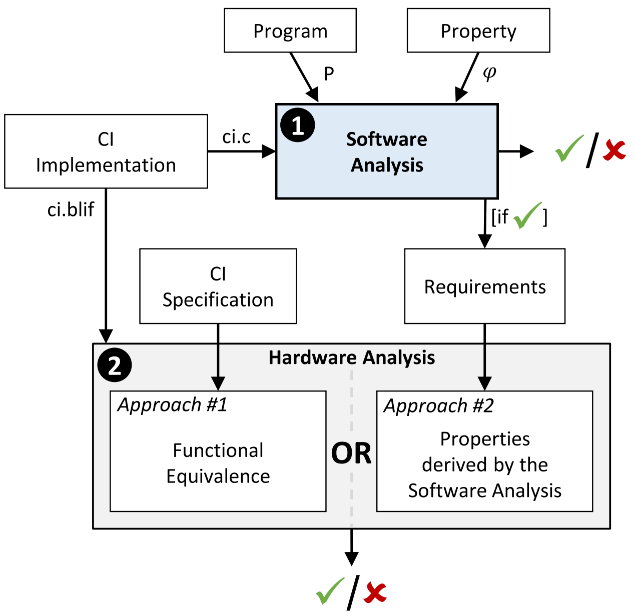


FIGURE 6. **Illustration of the link between software analysis (❶) and both hardware analysis approaches (❷: #1 and #2).**

The actual internal design of the property checker is highly dependent on the requirements placed by the software analysis on the custom instruction. The analysis of one specific CI circuit often requires several independent requirements to be verified. In that case we can devise several property sub-checkers and simply form the error flag as the disjunction of the outputs of these sub-checkers or as the negated conjunction of the underlying assertions, respectively, or we can simply run one verification per property to obtain several hardware certificates. In subsection IV-B, we will present a method to fully automate the property checker generation.

To prove full functional equivalence between an implementation of a custom instruction and its behavioral specification, we need to construct a circuit for a function that is commonly called a miter [18]. A miter comprises the implementation $I(\underline{in})$ and the specification $S(\underline{in})$, both of which receive the same inputs $\underline{in}$. The outputs of the implementation, $\underline{out}$, and the specification, $\underline{out}'$, are pairwise XOR-ed, and the disjunction of the results forms the error flag. The specification and implementation are equivalent if the outputs are identical for any input. Figure 5 sketches the resulting property checker for functional equivalence.

Note that checking for functional equivalence can thus be considered a special case of property checking, and both verifications therefore involve proving the unsatisfiability of a property checker applied to a custom instruction. Since solving SAT is NP-complete, the worst-case runtime of the hardware analysis is hence always exponential in the size of its input, i.e., the CNF formula generated from the check. The main difference between the general and special case lies in the minimization potential of that input size through a smaller checker and simplifications in the pre-processing steps before generating the CNF formula: When checking for functional equivalence, two full copies of the CI will be compared to one another and the resulting Boolean formula will not be reduced by much, but if we could find small and simple rule sets instead that suffice to show the functional correctness, then the resulting formula could be very small after the simplifications. This could then significantly speed up the hardware analysis, although it would not change the worst-case runtime behavior, since we would still need to employ SAT solving.

In the following sections, we present a novel integrated software/hardware co-verification approach, where software analyses such as the one reviewed in subsection II-B work together with the verification of RFU configurations. We detail two alternatives for the integration of software and hardware analyses designed to explore the trade-off between generality and efficiency.

III. LINKING SOFTWARE AND HARDWARE ANALYSES

We strive for establishing trust into the correctness of custom instructions *as far as needed by the software employing the CIs*, more accurately trust into the configurations for reconfigurable functional units. To this end, we employ formal hardware verification, which we need to properly link to the employed state-of-the-art software analysis, as depicted in Figure 6. Here, we introduce two approaches, #1 and #2, for analyzing if the hardware implementation is in line with the requirements of the software analysis. Approach #1 performs ❶ software and ❷ hardware analysis rather independently. It basically checks whether the custom instruction design is *functionally equivalent* to the intended behavior, as specified in a hardware description language. Approach #2 closely ties the requirements to be checked on the CI to the software analysis carried out on the specific program.

Linking both analyses as shown in Figure 6, i.e., first executing ❶ to verify if program P upholds property φ, and then performing either of the alternatives from ❷, will yield two certificates, one for the software and one for the hardware. Together, these two guarantee the adherence to the verified property φ for executions of the software program P when using the given hardware implementation of the custom instruction. The role of the hardware certificate is thus to prove the soundness of the assumptions on which the software certificate is based.

A. APPROACH #1: FUNCTIONAL EQUIVALENCE

The most general requirement for the functional verification of hardware is full functional equivalence of the implementation I to the desired behavioral specification S. For our example, the custom instruction $z = (x_1 * x_2) + x_3$, we thus need to show that it carries out the correct arithmetic operations, by comparing the actual low-level hardware description I, e.g., a technology mapped placed and routed netlist, to a high-level behavioral description of the desired custom instruction S, usually given in a hardware description language (HDL) such as Verilog. If functional equivalence can be shown, then all software analysis results will automatically hold for programs running on processor hardware using the custom instruction. For functional equivalence, we have to prove for every possible input $\underline{x}$, which is the vector of bits which results from mapping the program variables to the hardware input signals, that the output of the implementation $I(\underline{x})$ must match the output of the behavioral specification $S(\underline{x})$, or short: $\forall \underline{x} : S(\underline{x}) = I(\underline{x})$. Checking for full functional equivalence is done by most of the current hardware verification approaches (see e.g., [19]) and is also the basic strategy of some coupled hardware/software verification techniques [20], [21]. The differences to our approach are a) that we require a golden model for the instruction's behavior, i.e., specification S must be guaranteed to realize the original intent of the C instructions that the CI replaces and the software analysis is using, and b) that we actually might verify more than the software analysis needs to know, and thus might have an unnecessarily high effort. We thus pay for the generality of this approach with a higher risk of running into runtime or memory limits for the verifications due to the state explosion problem [22], and we will likely encounter problems whenever hardware limitations prevent us from realizing the original intent as CI, for instance due to limited bit-widths.

B. APPROACH #2: REQUIREMENTS OF THE SPECIFIC ANALYSIS

Our second approach presents a tighter integration of software analysis and hardware verification. Here, we extract the requirements for the hardware from the specific analysis result represented by the abstract reachability graph. The abstract reachability graph for a program as constructed by CPAchecker exactly tells us what properties the software analysis has used. This is typically much less than the full functional equivalence required by the first approach. In the extreme case, the behavior of the custom instruction does not influence the validity of the property at all and we need not check anything on the hardware.

In the following, we describe this approach for our analysis of program `FKT`. We only give an informal description of the approach; a formalization can be found in [23], [24] within the context of approximate computing and the validity of software analysis results for approximate hardware. The approach consists of the following steps:

CI localization Given a number of custom instructions to be employed for program execution, their usage needs to be located within the abstract reachability graph.

Pre- and postcondition extraction For every CI found in the abstract reachability graph, we extract the abstract values in the states directly before and directly after the CI in the form of a pair $(pre, post)$.

Variable replacement In every $(pre, post)$ pair, we replace the *program* variables by the corresponding CI variables. Note that the replacement can be different for the precondition and the postcondition. Furthermore, we equate CI variables with constant values as employed in the usage of the CI.

Pruning pre- and postconditions Pre- and postconditions often contain variables that do not occur in the CI, e.g., variable N in our example (Figure 3). Sometimes, there are transitively related CI variables, for example in formula $x < u \wedge u = z$ variable u relates x and z. For each pre- and postcondition pair, we determine the set V_{ntr} of all non-CI variables that do not transitively relate CI variables. Then, we remove all constraints from the pre- and postcondition that only contain variables from V_{ntr} and that only restrict the state.

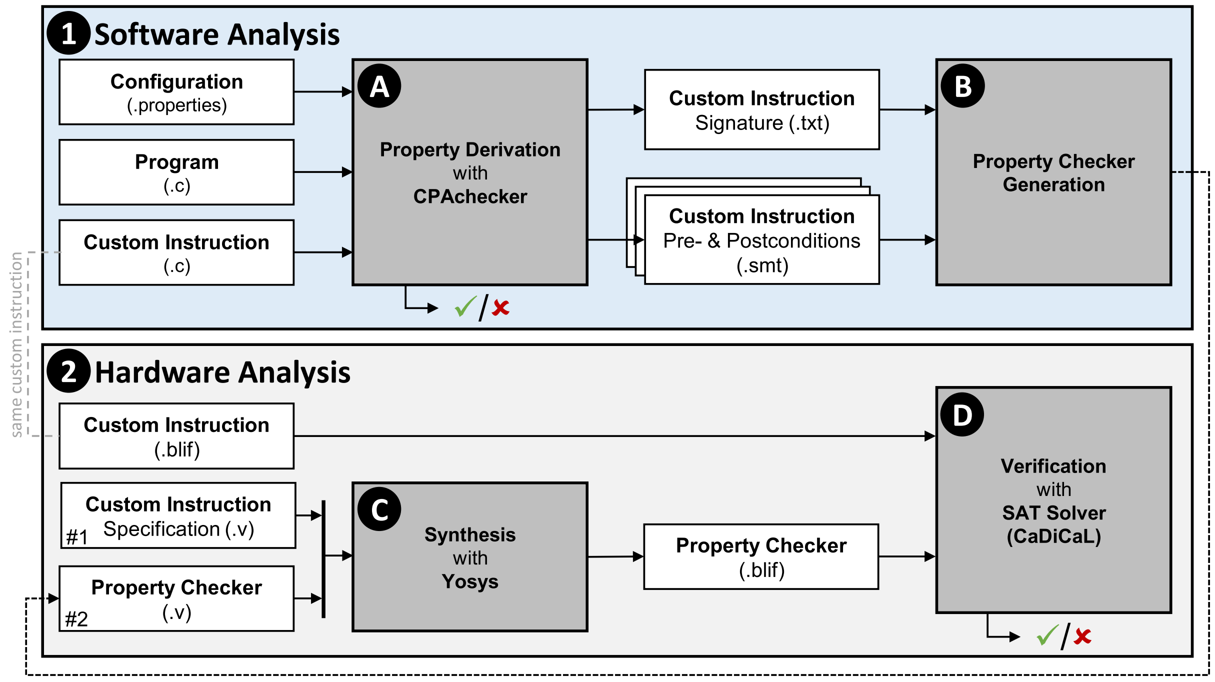
For program `FKT` these steps yield the following results: The CI localization finds the CI $z = (x_1 * x_2) + x_3$ on the edge from location 4 to 5 only. We thus extract the following $(pre, post)$ pair for this CI:

$$\begin{aligned}&(x \geq 0 \wedge x \leq 100 \wedge N \leq 100,\\ &\qquad x \geq 0 \wedge x \leq 100 \wedge y \leq 208 \wedge \neg(y = 0) \wedge N \leq 100)\end{aligned}$$

Next, we need to map program variables to CI variables (i.e., y to z and x to x_2), carry out an appropriate replacement in the pre- and postcondition and equate CI variables x_1 and x_3 with the constants 2 and 8, respectively. The pruning removes the constraint on N. As a result we obtain the following requirement to be checked on the hardware implementation of the CI:

$$\begin{aligned}&x_2 \geq 0 \wedge x_2 \leq 100 \wedge x_1 = 2 \wedge x_3 = 8\\ &\qquad \xrightarrow{CI} x_2 \geq 0 \wedge x_2 \leq 100 \wedge z \leq 208 \wedge \neg(z = 0)\end{aligned}$$

This requirement basically states that whenever the precondition is true, then the result of the execution of $(2 * x_2) + 8$ *when employing the CI design for this calculation* is not zero and less or equal to 208. This is the only requirement which needs to be checked on the CI implementation to guarantee



1 Software Analysis
Configuration (.properties)
Program (.c)
Custom Instruction (.c)
A Property Derivation with CPAchecker
✓/✗
Custom Instruction Signature (.txt)
Custom Instruction Pre- & Postconditions (.smt)
B Property Checker Generation
same custom instruction
2 Hardware Analysis
Custom Instruction (.blif)
#1 Custom Instruction Specification (.v)
#2 Property Checker (.v)
C Synthesis with Yosys
Property Checker (.blif)
D Verification with SAT Solver (CaDiCaL)
✓/✗

FIGURE 7. Tool chain for approaches #1 and #2.

that the software analysis result remains when using our example CI.

IV. TOOL CHAIN

We have built a tool chain automating all steps from Figure 6. Since the software analysis ❶ is not directly influencing the hardware analysis ❷ in the first approach, both can be performed independently of each other, with the added requirement that the specification S needs to be functionally equivalent to the C implementation of the custom instruction. This prerequisite has to be ensured with other means of verification, e.g., by having a knowledgeable designer implementing both in exactly the same way. For Approach #2, both analyses are actually linked in a chain via the assumptions of the software analysis.

We have defined the tool chain depicted in Figure 7 to link the software analysis ❶ to the hardware analysis ❷ without intermediate user interaction for both approaches. The only inputs which must be provided in order to start our co-verification approach are:

1) a configuration that specifies the software analysis to be used and the property (φ) to be evaluated,
2) the targeted C program (P), which uses custom instructions,
3) two implementations of each involved custom instruction – one in C to be used by the software analysis and another one as synthesized version in Berkeley Logic Interchange Format (BLIF) for the hardware analysis, and
4) for Approach #1 additionally specifications of the custom instructions as Verilog files.

The final outputs at the end of this fully automated tool chain are a software certificate that shows the validity of the specified property φ for program P, and a hardware certificate showing that the software certificate is based on valid assumptions about the hardware functionality. The software and hardware analysis including all four intermediate steps (see Ⓐ, Ⓑ, Ⓒ, Ⓓ in Figure 7) are described in the following. The automated linking step (Ⓑ) is obviously only needed and executed for Approach #2.

A. SOFTWARE ANALYSIS WITH CI REQUIREMENT EXTRACTION

The first step in Approach #2, which checks the custom instruction against requirements from the software analysis, performs the software analysis and the extraction of the requirements from the software analysis result (the ARG). We have decided to use the software analysis framework CPAchecker [14] to perform the analysis because it allows us to configure and automatically run a large set of analyses. However, to fully automatize the first step of Approach #2, we also need to integrate the requirement extraction into CPAchecker.

Figure 8 gives an overview of the components taking part in the process of extracting CI requirements from the software analysis. Next to the obvious components, the software analysis and the requirement extraction, there exists a component that identifies the custom instructions used in the program and a component that translates the requirements into standard SMT-LIB format [25],[3] the format employed

[3]http://smtlib.cs.uiowa.edu

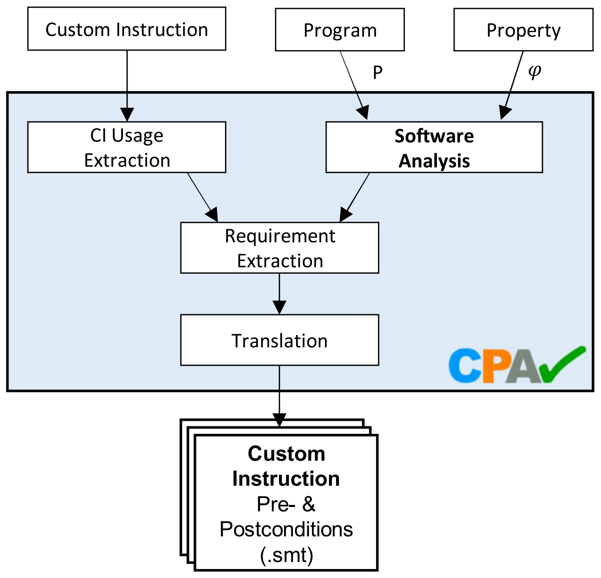


FIGURE 8. Overview on the process of CI requirement extraction from the software analysis.

by SMT solvers. The component identifying custom instruction usages in the program simplifies later extraction of the requirements and the translation component ensures that the requirements are not domain-specific abstract values, but are represented in a common representation.

To realize the process shown in Figure 8, we have integrated the identification of custom instruction usages as well as the requirement extraction and translation utilities into CPAchecker. Furthermore, we have implemented an algorithm that first calls the custom instructions usage extraction utility and then runs the software analysis. If the software analysis proves the property to hold, the algorithm will call the requirement extraction functionality to extract the requirements from the ARG and then translates the requirements into SMT-LIB format. Next, we describe the utilities that we have integrated into CPAchecker to realize requirement extraction in more detail.

1) EXTRACTING CUSTOM INSTRUCTION USAGES FROM PROGRAM

To extract the requirements on a custom instruction, we need to identify its usage in the ARG. Instead of identifying the CI usages in the ARG directly, we have decided to identify them in the program and use their start and end location in the program to find them in the ARG. The reasons for our decision are manifold. Most importantly, our predicate analysis uses adjustable block encoding [26] and only abstracts at specific program locations like loop heads. To get coarse requirements, we also need the predicate analysis to abstract at the beginning and end of custom instructions. Hence, we already need to know at analysis time where the custom instructions are to be found in the program. Furthermore, the identification in the ARG is more difficult. Due to additional analysis information, the CI in the ARG might not be syntactically identical to the CI in the program. For example, the ARG excludes paths that are known to be infeasible. Finally, a custom instruction usage in the program may occur multiple times in the ARG. Hence, it is analyzed multiple times, which is an unnecessary waste of resources.

To identify the custom instructions in the program, we get the custom instruction (prototype).[4] In our example, we get $z = (x_1 * x_2) + x_3$. First, we identify the signature of the custom instruction prototype, which is $(x_1, x_2, x_3) \mapsto z$ for our example. Then, we iterate overall program locations and check if they are start nodes of a custom instruction usage.[5] The check tries to find a structural match between the custom instruction prototype and the code starting at the respective location. To this end, it performs an exploration of the prototype and the code and checks that the two at most differ in variable naming or the code uses a constant instead of a variable. Additionally, the check examines whether the differences are consistent, i.e., whether exactly either one variable name or one constant is used for each variable in the prototype. If the structure matches consistently, a custom instruction is found and the information of the exploration is used to determine the end location(s) and the signature of the custom instruction. In our example, we identify a custom instruction that starts at location 4, ends at location 5, and has signature $(2, x, 8) \mapsto y$. At the end of this process, we have a set of custom instructions specified by their signature, start and end location.

2) REQUIREMENT EXTRACTION FROM ARG

The requirement extraction locates the custom instructions in the ARG (CI localization) and extracts the pre- and postcondition for each of those CIs. Localization is simple. We previously already identified the custom instructions in the program and, thus, know the starting locations of the CIs. Since we assume that the software analysis is at least flow-sensitive, i.e., abstract states (ARG nodes) are related to program locations, we only need to pass through the ARG nodes and identify those ARG nodes that refer to a program location that is the beginning of a CI. The identified nodes are the preconditions for the CIs and each node can be related to a particular CI in the program. To find the postconditions, we start at the identified ARG nodes and use the corresponding custom instruction *ci* in the program to follow all ARG paths until they end at a node that refers to a program location that is an end location in custom instruction *ci*. The set of all these ARG nodes form the postcondition.[6] As an optimization, we finally sort out all pre-/postcondition

[4] Technically, we encode the CI in a separate C function and use C labels `start_ci`, `end_ci_`*i* to describe the start and end of the custom instruction.

[5] To speed up the process, we tell the custom instruction process where custom instruction usages start.

[6] Note that although the CI itself has only one end location, the postcondition can still contain multiple ARG nodes. One reason can be that the CI's control flow is not a single path, e.g., it contains an if-statement, and the analysis does not join different paths.

```
(declare-fun x() Int)
(declare-fun y@1() Int)

;Custom Instruction
(define-fun ci() Bool (and (= 2 0)(and (= x 0)(and (= 8 0) (= y@1 0)))))

;Pre and Post Conditions
(define-fun pre() Bool  (and (>= x 0) (<= x 100)))
(define-fun post() Bool (and (>= x 0) (and and (<= x 100)
                        (and (!= y@1 0) (<= y@1 208)))))
```

FIGURE 9. SMT-LIB representation of the requirement for the custom instruction extracted from the predicate analysis result of our example program `FKT`.

pairs with postcondition that do not impose constraints on the CI.

3) TRANSLATING ABSTRACT STATES TO SMT-LIB FORMAT

After requirement extraction, pre- and postconditions are in form of (sets of) abstract states. Typically, the software analysis uses domain-specific representations for the abstract states. In our example, abstract states are represented by sets of predicates. However, to further automatically process the extracted requirements, they should be represented in a common format. We have decided to use SMT-LIB formulae [25] because it is a well-known standard and one can express many constraints on data states. Thus, we expect that most of the abstract states can be expressed in SMT-LIB. For all abstract domains we considered in our experiments, we are able to do the translation.

Figure 9 shows the SMT-LIB representation for the requirement from our example. The SMT-LIB representation starts with the declaration of the used variables and optional helper functions. Remember that the pre- and postcondition of our example refers to variables x and y. In contrast, Figure 9 declares variables x and $y@1$. We use $y@1$ to refer to variable y after it has been modified by the custom instruction. If the precondition had referred to variable y, we would have declared variable y, too, to refer to y before its modification. Generally, we use var and $var@1$ to distinguish between variables that are modified by the custom instruction. The values of variables that are not modified by the custom instruction are identical before and after the execution of the custom instruction. For these variables, we avoid to introduce two versions, for which we needed to explicitly state their identical values, and, thus, keep the representation more simple. The declarations are produced when translating the abstract states of a pre-/postcondition pair. The translation of each abstract state produces declarations. All these declarations are merged and duplicates are removed, before writing them to the SMT-LIB representation of the requirement.

After the declarations, the SMT-LIB representation of a requirement describes the corresponding custom instruction. We only need the custom instruction description to map it to the custom instruction prototype, which is $z = (x_1 * x_2) + x_3$ for our example. Since the structure of the custom instruction never deviates from its prototype, we only encode the signature of the custom instruction, i.e., inputs and outputs. The signature of our example is $(x_1, x_2, x_3) \mapsto z$, which we cannot express in SMT-LIB directly. Therefore, the signature is written as a conjunction of predicates using predicate $(= par\ 0)$[7] for each input or output par. The predicates in the conjunction are ordered such that we can relate them to the variables of the CI's prototype. In our example, the first predicate $(= 2\ 0)$ describes that $x_1 \mapsto 2$. The second predicate $(= x\ 0)$ states that $x_2 \mapsto x$. The third predicate $(= 8\ 0)$ encodes that $x_3 \mapsto 8$ and the last predicate $(= y@1\ 0)$ relates z to $y@1$.

The last part of the SMT-LIB representation is the pair of pre- and postcondition. The precondition results from the translation of the abstract state in the precondition. The postcondition is the disjunction of all translations of abstract states in the postcondition. Next, we describe how we translate abstract states to SMT-LIB. We describe the translation for predicate, value, sign, and interval states, the abstract states we use in our experiments.

a: TRANSLATING PREDICATE STATES

Predicate states are represented as Boolean formulae and CPAchecker already provides a converter to SMT-LIB. Translating predicate states is therefore rather simple. We only need to call the converter and split its result into (variable) declarations and formula representation. To properly add the suffix @1 to output variables when translating the post condition states, we use the instantiate method provided by CPAchecker and then call the converter. This method is used to get the static single assignment (SSA) representation of a formulae and adds the suffix @SSAIndex to all variables for which an SSA index is specified. In our case, we use an SSA index 1 for each output variable.

b: TRANSLATING ABSTRACT STATES OF VARIABLE-SEPARATE DOMAINS

Variable-separate domains constrain the value of each variable individually, but do not include constraints relating different variables. To this end, they separately assign abstract values to each variable. The abstract values differ among the variable-separate domains. Examples for abstract values are explicit values, sign values, or intervals.

[7] We cannot simply use par because it is not a Boolean formula.

TABLE 1. Mapping constraints from translating the mapping pair consisting of variable `var` and the respective sign value.

Sign Value	Mapping Constraint
-	(< `var` 0)
0	(= `var` 0)
+	(> `var` 0)
-0	(<= `var` 0)
0+	(>= `var` 0)
-+	(or (< `var` 0) (> `var` 0))

Typically, variable-separate domains represent abstract states as mappings from program variables to abstract values. Program variables that are not constrained, i.e., would be mapped to any value, are often not present in the mapping. However, this is no problem for translation. Not mentioning a variable in a requirement makes it unconstrained. To translate a variable-separate abstract state, it is therefore sufficient to compute a mapping constraint for each pair of variable and abstract value in the mapping and then build the conjunction of all these mapping constraints. If the mapping is empty, we therefore get constraint `true`. Moreover, each variable in the mapping must be covered by a variable declaration[8] at the beginning of the SMT-LIB representation. Next, we describe how to translate pairs of variables and abstract values into mapping constraints. This translation depends on the type of abstract value, i.e., it is domain dependent. We explain the translation for the three types of variable-separate domains we currently support.

Translation of Explicit Values. The explicit value analysis domain stores mappings from variables `var` to explicit values `val`, e.g., integer variables are mapped to integers. This can easily be represented by the SMT-LIB constraint `(= var val)`. Note that if we translate an abstract state of the post condition and `var` is an output variable, i.e., it is modified by the custom instruction, we use `(= var@1 val)`.

Translation of Sign Values. The sign domain stores the sign of a variable, e.g., whether it is positive (+), negative (−) or zero (0). Table 1 shows how to translate a pair of a variable `var` and a sign value. For Table 1, we assume that either we translate the precondition or `var` is not an output variable. If we translate an abstract state of the post condition and `var` is an output variable, the constraints use `var@1` instead of `var`.

Translation of Intervals. The interval domain maps numerical variables to intervals $[a, b]$. Intervals $[a, b]$ either use numerical values for a and b or are unbounded on one side (either $a = -\infty$ or $b = \infty$). In the following, we present the mapping constraints resulting from translating the pair $(var, [a, b])$. Thereby, we assume that either we translate the precondition or `var` is not an output variable. If the assumption is invalid, one needs to use `var@1` instead of var.

Case $a = -\infty, b$ numerical: We only need to constrain the upper value of `var` resulting in the mapping constraint `(<= var b)`.

Case a numerical, $b = \infty$: In this case, we constrain the lower value of `var` resulting in the mapping constraint `(>= var a)`.

Case a, b numerical: We need to constrain upper and lower value of `var`. To this end, we conjunct the constraint for the upper and for the lower value. The mapping constraint gets `(and (>= var a) (<= var b))`.

B. LINKING VIA PROPERTY CHECKER GENERATION

To link the software analysis to the hardware verification all pairs of pre- and postconditions must be translated into a so-called *property checker* in form of a Verilog module (see **B** in Figure 7). For this translation task, we have developed the PropertyCheckerGenerator (PCG). It takes two outputs of the software analysis as input, namely the set of pre- and postcondition pairs ($\mathbb{P} = \{(pre_0, post_0), (pre_1, post_1), \ldots, (pre_j, post_j)\}$) and the CI's specification. The generator constructs one implication per pre- and postcondition pair. The negation ($\neg$) of the conjunction of all these implications, represents the property checker as depicted in Figure 10. As the figure shows, the property checker takes three inputs (`in`, `out`, `extra`) and computes one output (`error`). Input `in` represents all inputs that are also provided to the CI – `out` refers to the associated output. `extra` holds variables which are used in a pre- or postcondition although they do not appear in the CI's specification. While `in`, `out` and `extra` hold n, m and k integer values respectively, `error` represents the Boolean (1-bit) error flag. Considering $\mathbb{P}$ with $|\mathbb{P}| = j$, the property checker implements the following formula:

$$\begin{aligned} error \; &= \neg\big((pre_0 \rightarrow post_0) \\ &\quad \wedge \; (pre_1 \rightarrow post_1) \wedge \ldots \wedge (pre_j \rightarrow post_j)\big) \\ &\Leftrightarrow \\ error \; &= \neg\big((\neg pre_0 \vee post_0) \\ &\quad \wedge \; (\neg pre_1 \vee post_1) \wedge \ldots \wedge (\neg pre_j \vee post_j)\big) \end{aligned}$$

With respect to the running example, there is only one precondition (pre_0) and one postcondition ($post_0$), hence, the generated property checker encodes the following formula:

$$\begin{aligned} error \; &= \neg(pre_0 \rightarrow post_0) \\ &\Leftrightarrow \\ error \; &= \neg\Big((x \geq 0 \wedge x \leq 100) \rightarrow (x \geq 0 \wedge x \leq 100 \wedge y \\ &\qquad \leq 208 \wedge \neg(y = 0)) \Big) \end{aligned}$$

Note, the CI's specification (second input to the PCG) is required in order to map the variables used in it to those appearing in any pre- or postcondition. In case of the running example, x and y are mapped this way. A shortened version

[8]In post conditions, output variable names need to be extended by the suffix @1.

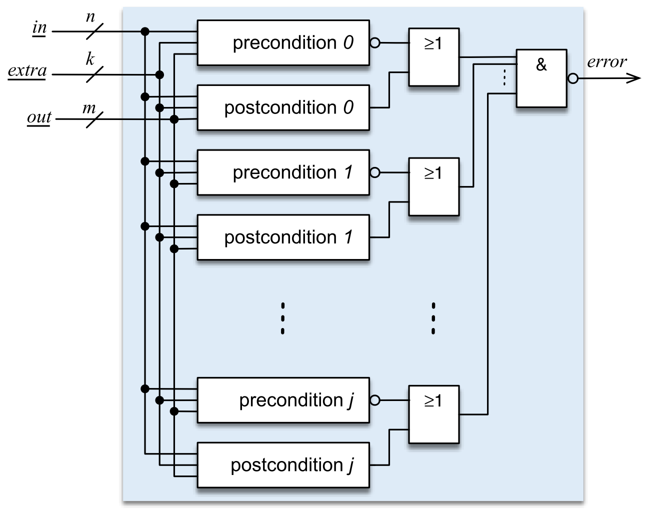


FIGURE 10. **Generated property checker for linking hardware and software analyses.**

of the Verilog module associated with the running example can be found in Appendix A.

C. HARDWARE VERIFICATION TOOL FLOW

To perform the hardware verification required for part ❷ of Figure 7, we use the synthesis suite Yosys [27], the synthesis and verification tool ABC [28], and the SAT solver CaDiCaL [29], which placed first in the SAT track and second overall in the SAT Race 2019. We chose Yosys, because it is a powerful open-source tool that can synthesize behavioral Verilog into a netlist, but in principle any synthesis tool would work that can output netlists in a format that ABC can read. For the actual verification, we take the circuit netlist for a custom instruction and convert it into ABC's natural representation: an And-Inverter-Graph. The property checkers required for the verification model are also supplied as Verilog files, and represent either the specification of the custom instruction, or the automatically generated checker circuit from step 🅑. We synthesize the property checker using Yosys, and then build the entire verification model in And-Inverter-Graph representation. When checking for functional equivalence, we can utilize ABC, which has the capability to automatically transform two circuits into a miter if they use the same number of inputs and outputs. For Approach #2, we use our own tool to combine the representations for the implementation and the property checker into one And-Inverter-Graph, our verification input. Again using ABC, we simplify the graph by removing everything that does not influence the error flag, i.e., is not in its cone-of-influence, and then transform the graph into a large Boolean formula in conjunctive normal form (CNF). We then use CaDiCaL to prove unsatisfiability of the CNF formula and thus the adherence of the CI implementation to the requirements of the software analysis.

As explained above and in subsection II-C, the underlying verification involves SAT solving with CaDiCaL, and hence the worst-case runtime behavior for both approaches is exponential in their input size. Since we can predict neither the quantitative nor qualitative impact of employing Approach #2 over Approach #1 for the subsequent SAT verification, we have conducted a large set of experiments to gain new insights here. The features and trade-offs involved for both approaches will be illustrated and discussed in section VI using the 244 case studies introduced in section V.

V. EXPERIMENTAL SETUP

One goal of our evaluation is to demonstrate the feasibility of our two methods for integrating software and hardware analysis, which guarantees a reliable software analysis despite the use of reconfigurable functional units. Additionally, we aim to show that our methods can be run fully automatically and do not require expert knowledge in verification. Also, we are interested in the additional costs for automation, e.g., the extraction of requirements from the software analysis. Another objective is to compare the two methods and get a better insight when which method should be applied. Especially, we want to investigate whether the custom instruction and the software analysis significantly influence the performance of the methods and how. To answer these questions, we need to evaluate the two methods on a set of verification tasks. More concretely, we require abstraction-based software analyses that verify programs that use custom instructions.

A. APPLIED SOFTWARE ANALYSES

In this experiment, we consider three different software analysis techniques: dataflow analysis (D) [30], [31], model checking (M) [32], and model checking with CEGAR (M+) [16]. All three analysis techniques are parametrized by the abstract domain. We consider four abstract domains: interval abstraction (I) [33], explicit value abstraction (V) [31], [34], sign abstraction (S) [35], and predicate abstraction (P) [26]. We combine the value abstraction with all three analysis techniques. However, we combine the interval abstraction only with the dataflow analysis technique because using interval abstraction with model checking techniques is the same as using model checking with value abstraction. Moreover, we combine the sign abstraction with the dataflow analysis and the model checking technique. As common, the predicate abstraction is only used in model checking with CEGAR. In total, we get 7 different software analyses. All of them are implemented in the software analysis framework CPAchecker [14], in which we have integrated the property derivation of Approach #2 (cf. Figure 7).

B. ANALYZED CUSTOM INSTRUCTIONS

Our custom instructions are specialized operations common in programs like increment, common programming idioms like compare and swap, or combinations of multiple operations, which occurred in benchmark programs. Table 2 lists the 14 custom instructions considered in our experiments.

TABLE 2. Overview on custom instructions used in our experiments.

Name	ID	Description
3-Input adder	`mia`	`r=x+y+z;`
Add an multiply	`am`	`y=(u+v)*x;`
Compare and swap	`cas`	`if (x==y) x=z;`
Complex condition	`cc`	`r=((((((!(x==1)&& (y==1))&& (z==1))&& (t==a))&& (u==b))&& (v==c))&& (w==1));`
Conditional set	`cs`	`if (x>0) y=0; else y=1;`
Increment	`i`	`x=x+1;`
Multiply	`mul`	`z=x*y;`
Multiply and add	`ma`	`y=u*v+x;`
Parallel decrement	`pd`	`x=x-1; y=y-1;`
Parallel saturating decrement	`pd`$_{\text{SAT}}$	`x=x-`$_{\text{SAT}}$`1; y=y-`$_{\text{SAT}}$`1;`
Saturating add	`sa`	`z=x+`$_{\text{SAT}}$`y;`
Subtract and multiply	`sm`	`y=(u-v)*x;`
Swap	`s`	`z=x;x=y;y=z;`
Zero definition	`z`	`x=0;`

For each CI, Table 2 lists its name, its identifier, and a description in program code.

C. PROGRAMS

To apply our methods, especially to verify programs, we require programs with specifications, e.g., assertions, locations that must not be reached, etc. Also, one of our analyses must be able to prove that the program fulfills its specification. Additionally, the programs must use at least one of the CIs from Table 2. In total, we collected 97 programs meeting the above requirements. Some programs we created ourselves, a few programs are taken from research papers [36]–[38], but most of the programs are from existing benchmarks. We considered programs from the NECLA static analysis benchmarks [39] (including our three previous examples [7]), the VeriSec benchmark suite [40], and the SV-COMP benchmark [41].

D. TASK SET

As already mentioned, a program may not use all custom instructions. Furthermore, even if a CI is used by a program, it will not guarantee that it is relevant, i.e., we can extract requirements with postconditions true. Additionally, not every software analysis can verify each of the programs, hardware and software analyzers may disagree on the semantics,[9] and verification capabilities may even depend on the custom instruction.

Our goal was to build a task set that ensures that we consider different analyses for each custom instruction and vice versa, for each analysis different custom instructions. To this end, it is sufficient to consider one custom instruction type at a time and our current automation can only deal with one type. Handling multiple custom instructions is an implementation issue, which we do not need to solve for our experiments.

Table 3 shows which pairs of analysis and custom instruction we managed to cover with our task set. Although the table is densely filled, we also notice that we do not

[9]For example, the software constraints assume mathematical integers or no overflow, while the hardware approach 2-complement integers.

TABLE 3. Overview on which combinations of custom instructions (rows) and software analyses (columns) are covered (✓) by our task set.

	Software analysis						
	Dataflow analysis			Model Checking		Model Checking + CEGAR	
CI	Interval	Sign	Value	Sign	Value	Predicate	Value
`mia`	✓	✓	✓		✓		✓
`am`						✓	✓
`cas`	✓	✓	✓		✓	✓	✓
`cc`					✓	✓	✓
`cs`		✓		✓	✓	✓	✓
`i`	✓	✓	✓	✓			
`mul`		✓	✓		✓		
`ma`						✓	✓
`pd`						✓	
`pd`$_{\text{SAT}}$						✓	
`sa`	✓		✓	✓		✓	
`sm`						✓	✓
`s`		✓	✓	✓	✓	✓	✓
`z`	✓		✓	✓	✓	✓	✓

cover all pairs. Furthermore, note that we created our task set by looking at the pairs of software analysis and custom instruction independently and trying to find programs that use the particular custom instruction and that can be verified by the analysis. Table 5 in Appendix B gives an overview on our task set, which consists of 244 tasks. The sparse table describes for each pair of program and analysis (i.e., cell in the table) which custom instructions are considered.

E. CUSTOM INSTRUCTION IMPLEMENTATION

To prepare the benchmark circuits for the evaluation, we require an implemented version for an RFU. To this end, we employ the academic tool flow VTR [42] and the synthesis suite Yosys [27], both of which are open-source. We specify the custom instructions in Verilog, (manually) making sure that we implement the same functionality as in the C files, which the software analysis uses. We implement the data type used in the software analysis, which is a signed integer of unspecified bit width, using 64-bit binary numbers encoded in two's complement, and then map the CIs to circuits using Yosys for hardware synthesis, followed by ABC (which is used by both, Yosys and VTR) for technology mapping. The circuits are technology-mapped to a generic FPGA architecture that is defined using VTR's architecture

TABLE 4. Runtimes for the software analysis, averaged over all 243 case studies.

		Max (s)	Min (s)	Average (s)	Median (s)
1.	Analysis with CI extraction	752.06	0.05	11.88	2.64
2.	Analysis without CI extraction	751.88	0.03	11.78	2.58
3.	CPACHECKER	761.64	2.86	22.30	10.02
4.	PCG	76.84	0.22	0.89	0.34
		Max (%)	**Min (%)**	**Average (%)**	**Median (%)**
5.	CI extraction (Difference between 1. and 2.)*	*48*	0	**7**	**3**
6.	Relative runtime of PCG (4. compared to 3.)**	*309*	0	**6**	**3**

*: There are only 35 of 243 cases where the derivation is $> 10\%$ and considering all these cases the runtime is less than $1s$.
**: There are only 11 of 243 cases where the derivation is $> 10\%$

description mechanism and is comprised of logic blocks with 6-input lookup tables. For a real customizable processor, the resulting circuit netlist would be placed and routed for the reconfigurable fabric of an RFU and the configuration bitstream would be generated. For the sake of simplicity, these steps are omitted in our current work. Expanding our tool flow to cover also these steps is part of future work and will allow us to catch not only design errors and errors introduced by hardware synthesis and technology mapping, but also errors due to low-level FPGA implementation tools. We store the resulting circuit representation generated by ABC as a BLIF file and use this as the final implemented hardware version of the custom instruction, i.e., as input for part ❷ in Figure 7.

VI. EXPERIMENTAL RESULTS AND DISCUSSION

We have used the tool chain from section IV to run all 244 experiments described in section V. The software side has been run on a Debian 9 (Stretch) virtual machine with Java 11 (OpenJDK 11.0.5) installed. Two cores of the host's Intel Xeon E5-2695v3@2.30GHz were assigned to the guest machine. The memory provided to the analysis tools was limited to a maximum of 32 GiB. For the hardware analysis, we have used a compute cluster with a per-experiment time limit of seven days and memory limit of 5 GiB. The cluster nodes run Scientific Linux 7.2 (Nitrogen) on an Intel Xeon E5-2670@2.6 GHz with 16 cores; however, the current tool chain is single-threaded.

Considering the software side, we compared the runtimes of the analyses executed through CPAchecker with respect to Approach #1 and #2. In both scenarios, the same analyses are executed but considering #2 the pre- and postconditions must be extracted. In addition, the property checker must be built when dealing with Approach #2. Table 4 shows the maximum, minimum, average and median of all runtimes measured. In the first two rows, the runtimes of the employed software analyses *with* and *without* CI extraction are summarized. The next two rows titled with CPAchecker and PCG deal with the overall runtime of both tools. When comparing the runtimes with and without CI extraction (1. and 2. in Table 4) the difference is always negligibly low (only 3% wrt. the median—see 5. in Table 4). Also negligibly low (3% wrt. the median—see 6.) is the time required by PCG (4.) in comparison to the time consumed by CPAchecker (3.). In a single case, the runtime of PCG is greater than the runtime of CPAchecker (309%), since CPAchecker extracts 479 pre- and postcondition pairs in this case—way more than in any other case (30 on average). Apart from this case, PCG's runtime is always lower ($\leq 64\%$). In summary, the runtime required by the software side is almost the same regarding Approach #1 and #2. Consequently, we only discuss the runtimes of the hardware verification in detail in the following.

To evaluate these verification times, we executed our complete tool chain, depicted in Figure 7, for all 244 case studies. One of the most important results from this series of experiments was the fact that it uncovered several tasks in which the software analysis actually had posed unrealistic assumptions on the C code, which, when translated to requirements on the CI's circuit, were consequently not fulfilled, leading to failed verifications for Approach #2. These requirements are mostly due to over-simplifying assumptions designed to increase the performance of the software analysis in cases where the exact physical limitations of the executed commands, such as bit widths of variables, are not actually needed. Approach #1 does not expose this issue, since it is unaware of and unaffected by the preceding software analysis, and can thus regard neither the resulting requirements, nor the context in which the CI will be executed for its hardware verification. For Approach #2, however, each assumption of the software analysis is translated into a requirement for the employed CI, disregarding any such incompatibilities. Using this approach, a verification engineer can thus be made aware of any model mismatches that happen during this translation from software to hardware verification.

For our concrete benchmark set, the software analysis' ignorance of the specific bit widths of the signals lead to requirements that are impossible to satisfy in nine cases. The verification had simplified the software's structure with unbounded integers for its analysis on sign abstraction in eight cases, and in two cases on predicate abstraction, which resulted in violating cases in the hardware when the signal overflows, underflows, or satisfies at the extreme values in the operation. We have encountered this issue with the following benchmarks and CIs: *ex19* with $\texttt{pd}_{SAT}$, *ex49* with `i`, *factorial* with `i` and `mul`, *first_binomial* with `mia` and `mul`, *Problem14_l35* with `sm`, and *resize* with `mia`.

Of the same nature but with reversed roles was the failure of one benchmark, *lock-impl-s* with `cas`, where the software

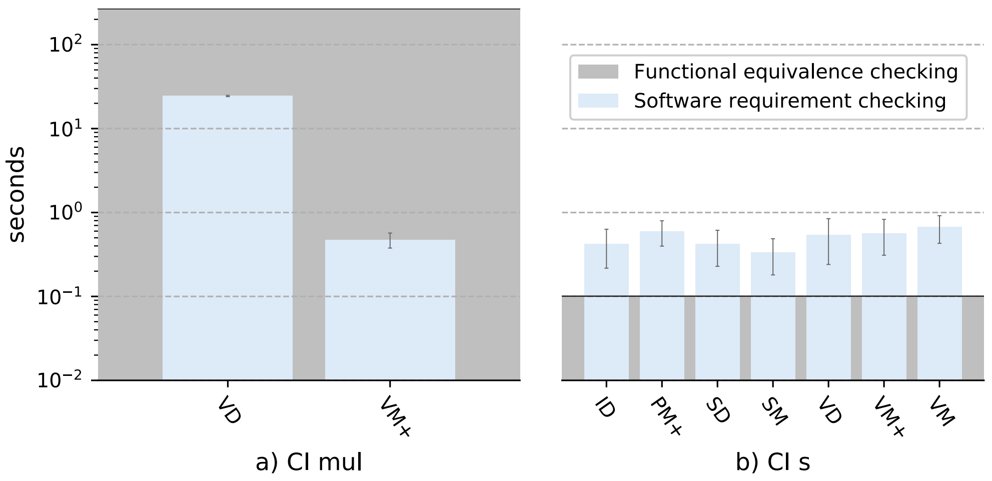


FIGURE 11. Runtimes for the hardware analysis in seconds, grouped by custom instruction and averaged over 100 runs for each of the case studies using the CI `mul` (a) or `s` (b). The labels indicate the employed software analysis.

analysis had deduced the possible states of two variables to be binary and had therefore concluded that they are Boolean variables in the context of the program, while the CI working on them was realizing the full precision of the underlying C variables. The analysis then internally exploited the duality of the two possible variable states in its assumptions, which then later failed as requirements for the hardware CI, whose possible state space was not limited similarly.

All of these cases combined thus effectively show that traditional software analyses tend to neglect the limitations of the underlying hardware, especially when bit widths are concerned. This mismatch between both worlds is the exact reason for our work in this paper, where we aim to provide a verification for the soundness of the software analysis despite these differences. Their detection through our flow hence proves the necessity of such a coupled verification as well as the adequacy of our method to uncover such issues. The successful verifications of each of these cases using Approach #1 furthermore highlights how easy it is to miss these broken requirements when employing a traditional, uncoupled verification style.

We will now discuss the 235 case studies that were successfully verified with both approaches. Figure 11 shows the runtimes for the hardware verification for the case studies using the custom instructions `mul` and `s`, i.e., the time in seconds required to perform a cone-of-influence reduction with ABC to cut away all logic that does not contribute to the satisfiability of the property checker, and to prove the unsatisfiability of the verification model with CaDiCaL. The figure groups the experiments by their custom instruction, with Figure 11 a) showing CI Multiply and b) CI Swap (cp. Table 2). Each blue bar represents the average runtime of one of the 235 successfully verified benchmarks for Approach #2, checking the requirement-based properties, while the label at the axis denotes the employed software analysis for that task. The gray bars depict runtimes for Approach #1, checking the functional equivalence, and since this has to be performed only once per CI, its runtime is depicted spanning each subfigure, with the dark line denoting the average runtime over a 100 runs per involved benchmark.

For CI `mul` in Subfigure a), the hardware verification runtimes for Approach #2 are orders of magnitude faster than verifying the full functional equivalence in Approach #1, although differences in the requirements generated for the involved software analyses also lead to significant changes in the runtime of Approach #2. Custom instructions like `mul` thus clearly indicate that the second approach can save significant verification effort, when the functional equivalence checking encounters structures that trigger state explosions, such as multiplications, especially when the employed software analysis only requires a partial verification of the circuit. For CI `s` (Subfigure b) all verification times are in the range of less than one second for both approaches, i.e., very small. This shows that the resulting combinational circuits of custom instructions might often times be quite easy to verify, enabling SAT solvers to prove even full functional equivalence of specification and implementation quickly, and that proving the extracted properties is neither faster nor much slower in these cases.

Figure 12 shows all benchmarks for the remaining custom instructions, who mostly underline the above observations: For many CIs, e.g., `mia`, `sa`, `i`, `cs`, both approaches are very fast and do not differ by much, such that either approach is well suited for verifying these benchmarks. And similar to CI `mul`, other CIs that lead to hard verification problems for full functional verification, i.e., `am`, `ma`, and `sm`, can greatly benefit from Approach #2 when it enables a much faster hardware verification. For our benchmark set, we have observed differences for these types of CIs of under one second to about 1000 seconds of verification runtime.

There is, however, also a third type of CI which our experiments clearly show with CI `pd`, for which the role

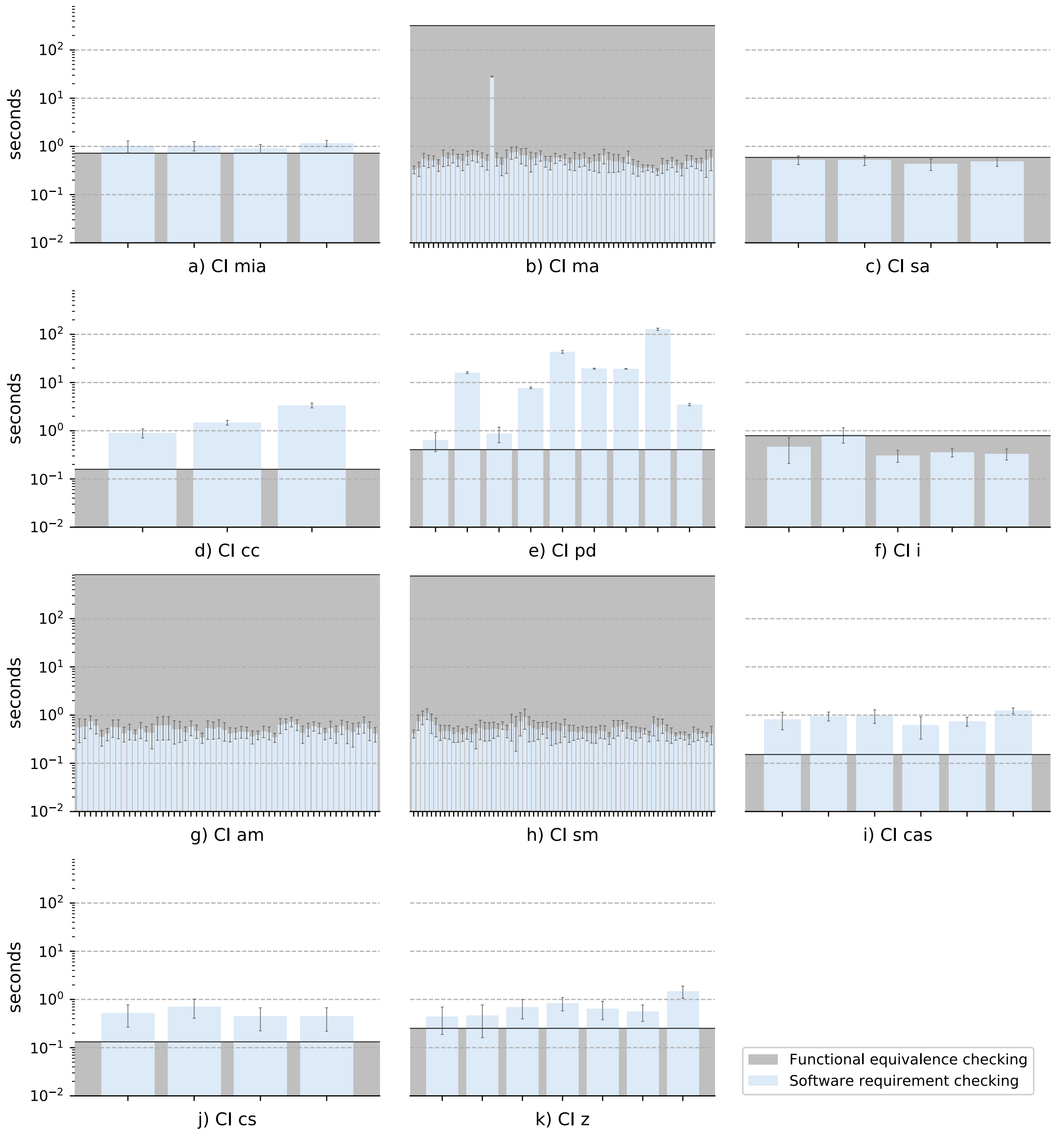


FIGURE 12. Runtimes for the hardware analysis for the remaining instructions in seconds, grouped by CI. Each blue bar represents one of the 235 benchmarks, averaged over 100 runs.

of the approaches is reversed. The special structure of this CI, which performs a parallel decrement of two operands, allows for a very efficient hardware verification, since the verification problem basically collapses into two very easy disjoint verifications of one subtraction circuit each. The extracted requirements from the software analysis, however, correlate the two inputs and also the outputs with each other, since they co-exist with a known difference in the abstract state conditions. The hardware verification for Approach #2 thus does not lead to two mostly disjoint subproblems, but remains entangled and thus more complex to solve for the verification engine.

Figure 13 presents an overview of the same runtimes of the 235 case studies grouped by employed software analysis

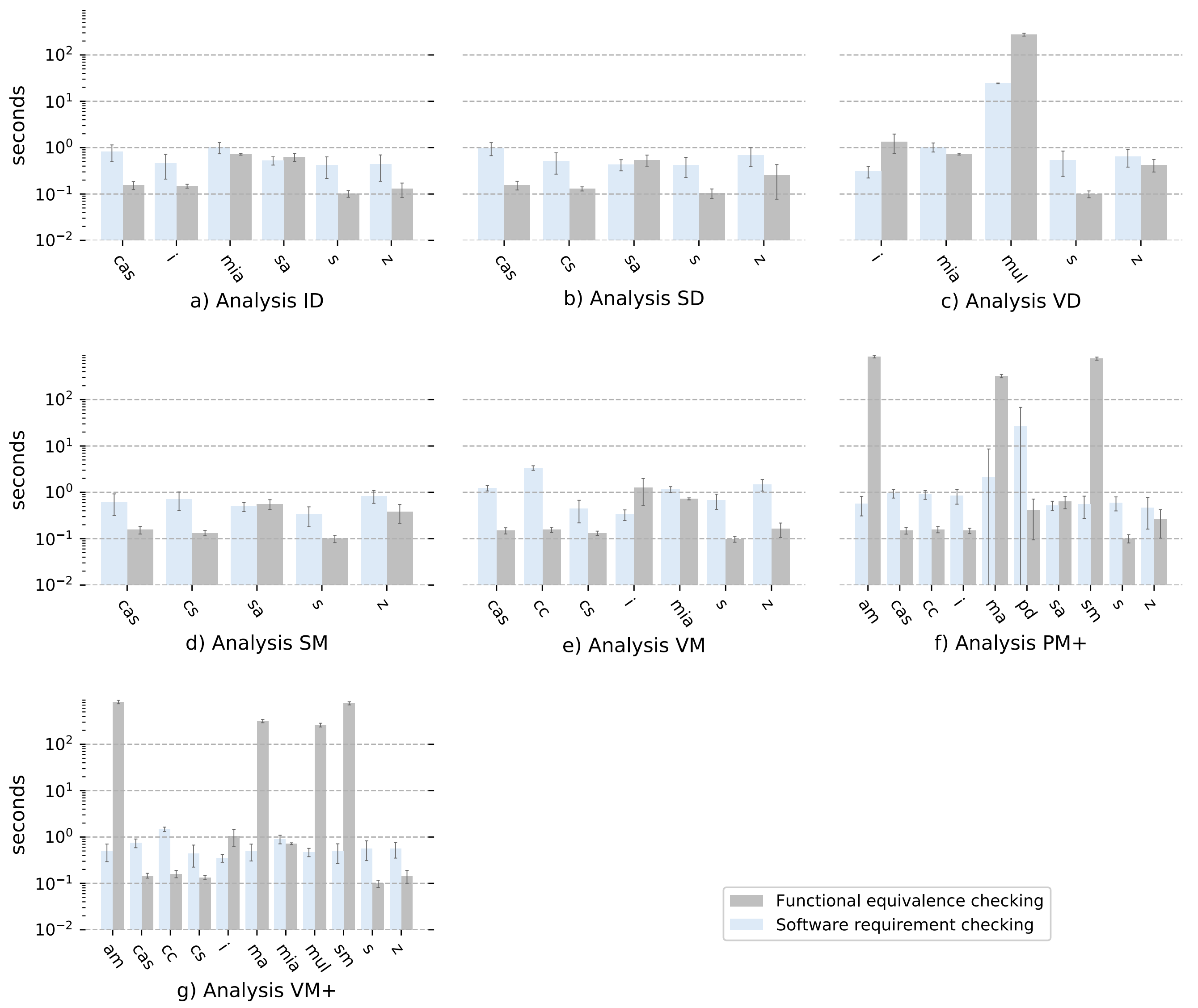


FIGURE 13. Runtimes for the hardware analysis in seconds, grouped by software analysis and averaged per custom instruction and 100 runs for each of the 235 case studies. The labels indicate the employed custom instruction.

(cp. Table 3). Observe that while in Figure 11 and 12 the runtime relation between the approaches does not change over the performed experiments for most of the presented custom instructions, it differs significantly for the software analyses in Figure 13, where we can still clearly identify the harder custom instructions. This effect is mainly due to the runtime relations between the approaches and the CIs and analyses, where the internal structure of the CI determines the complexity of the formal equivalence checking, i.e., the gray bars in Figures 11 through 13, whereas the employed software analysis is responsible for the complexity of the derived properties that have to be checked for Approach #2.

On the other hand, there are some software analyses such as VM+, which corresponds to the last column of Table 3, who seem to generally lead to easily verifiable requirements on the hardware, even if the involved CI itself is harder to verify, since these requirements only demand the correct functionality of the CI for a very narrow value range. This enables the SAT solver to break down the complex verification of the multipliers involved in the CIs `am`, `ma`, `mul`, and `sm` to just a few relevant states, thus avoiding the usual state explosion.

We therefore conclude that whether it is advantageous to check the assumptions instead of the equivalence seems to be predominantly determined by the combination of the hardware circuit's verification complexity versus the complexity of the requirements derived by the employed software analysis. The differences between the blue bars of the subfigures of Figure 13 reveal, however, that the overall hardware verification time for Approach #2 is furthermore influenced by the employed CI, software property, and

```
module Propertychecker(in, out, error);
        parameter Nin = 64; parameter Nout = 64;
        input [3*Nin-1:0] in; input [1*Nout-1:0] out;
        output error;

        wire signed [Nin-1:0] x1_0; ...; wire signed [Nout-1:0] z_0;
        wire term_9; ...; wire term_23;
        wire post_0;
        wire pre_0; wire pre_gen_0; wire pre_gen_1;

        assign x1_0 = in[1*Nin-1:0]; assign x2_0 = in[2*Nin-1:1*Nin]; ...; assign z_0 =
            → out[1*Nout-1:0];

        assign term_9 = (x2_0 >= 0);
        assign term_10 = (x2_0 <= 100);
        assign term_11 = (term_9 && term_10);
        assign pre_0 = term_11;
        ...;
        assign post_0 = term_23;
        assign pre_gen_0 = (x1_0 == 2);
        assign pre_gen_1 = (x3_0 == 8);

        assign error = !( ( !(pre_0 && pre_gen_0 && pre_gen_1) || post_0) );
endmodule
```

FIGURE 14. **Property Checker generated for our example program** FKT **(Shortened for legibility).**

program, albeit on a much smaller scale. In summary, these 235 case studies support our expectation that exploiting more knowledge about the specific analysis can lead to a more efficient verification.

VII. RELATED WORK

There are a number of approaches for software/hardware co-verification. Several techniques exist ([43]–[50]) which first compute a joint model of hardware and software and then check properties on that model. Other approaches [51], [52] apply compositional reasoning. Xie *et al.* [51] check component-based systems and present a scenario–system development–in which requirements for subcomponents, hardware or software, are created. Whenever a component is partitioned into subcomponents, the requirements for the subcomponents are formulated and the component is verified based on the assumption that the subcomponents' requirements hold. Although the requirements are used in verification, they result from partitioning and not from verification as in our second approach. Loitz *et al.* [52] first verify the hardware assuming certain constraints on the environment and then check that the software respects these constraints. In a sense, they aim at the same goal as our work does, but start from the opposite side. The disadvantage of this technique is that it might produce overly complex constraints, whose validation is not needed for trustworthiness. Again, other works ([20], [21]) check that the hardware is equivalent to a specification, which is somewhat similar to the hardware side of our first approach. For example, Clarke *et al.* [20] verify Verilog against a C specification and Erkok *et al.* [21] verify a low level circuit implementation, e.g., in form of a netlist, based on a cryptol specification.

Besides these co-verification approaches, there are further integrations of hardware and software level validation techniques (e.g., [53]–[56]) using simulation, testing or some form of monitoring. However, none of these techniques targets custom instruction set extensions and derives hardware properties to be validated from a software analysis step.

The influence of hardware correctness on software verification has also been investigated in the area of *approximate computing* [57] (AC), where computation precision of hardware components is sacrificed for reducing energy consumption. Approaches for the correctness of software in the context of approximate computing have for instance studied quantitative reliability, i.e., the probability that outputs of functions have correct values [58], [59] or values within a tolerated quality of service band [60], or differences between approximate and precise executions [61]. Furthermore, PAC [62] computes for each instruction the required degree of accuracy, e.g., allowed number of incorrect bits, when a given accuracy of the output. Closest to our own work is that presented by Isenberg *et al.* [23], [24]. While they also use the abstract reachability graph for extraction of constraints on the behavior of hardware, they focus on approximate circuits instead of processors with custom instruction set extensions and experimentally validate their technique on a number of approximate adders [24].

VIII. CONCLUSION

In this paper, we have proposed a new technique for software/hardware co-verification for processors with custom instruction set extensions. We have detailed two approaches tailoring the hardware verification to the needs of the software analysis to different extents, thus potentially allowing for a trade-off between generality and computational effort.

TABLE 5. Overview on task set. The columns fix the software analyses and rows the program. A cell contains the ID of the custom instructions considered for the respective pair of analysis and program. (*: am, ma, sm).

	Software analysis						
	Dataflow analysis			Model Check		Model Check + CEGAR	
Program	Interval	Sign	Value	Sign	Value	Predicate	Value
abs-diff						s	
array-sort-type							mia
evensum				sa			
count-npos		cs					
count-npos-rnd					cs, mia		cs
factorial		i, mul					
fibonacci		s		s			
fibonacci-call	s						s
first-binomial					mia, mul		
fkt						ma	
grey	mia						
lock-impl						cas	
lock-impl-s		cas	cas				
maximum						pd	
minimum						pd	
resize		mia					
ring-buffer	cas						
safe-div					cas		
			(*: am, ma, sm)				
subtract						pd	
subtract2						pd	
sum	sa					sa	
triangle			mia, mul				
octagon						i	
spp2015			i				
PfPb2			s				
PfPb					s		
ex19						pd, pd$_{\text{SAT}}$	
ex39						pd	
ex49		z, sa		i			
inf6				z, cs		z	
apache-gettag					i		
encode-ie-iproc							i
down						pd	
fragtest-simple						pd	
jm2006						pd	
kbfiltr-s1					z		
n.c11	i, z						
Problem01-l00-10					cc		
Problem01-l11-19,31,34,36,39							cc
Problem01-l22-30						cc	
Problem10-l03						sm	*
Problem10-l04						ma	*
Problem10-l05						ma, sm	*
Problem10-l06							*
Problem10-l07						*	*
Problem10-l08						ma, sm	*
Problem10-l09						ma, sm	*
Problem10-l10						ma, sm	*
Problem10-l14						sm	sm
Problem10-l19						sm	*
Problem10-l20						*	*

Further, we have presented a fully automated tool chain and reported on extensive experiments.

As a main result from our experimentation, we can conclude that while in general tailoring the hardware verification more to the concrete needs of the software analysis indeed results in lower computational effort, neither approach is superior for all cases. Moreover, we have been able to identify clear recommendations for each of the two

TABLE 5. ***(Continued.)*** **Overview on task set. The columns fix the software analyses and rows the program. A cell contains the ID of the custom instructions considered for the respective pair of analysis and program. (*: `am`, `ma`, `sm`).**

Problem10-l21			ma	*
Problem10-l22				*
Problem10-l31				*
Problem10-l32			ma, sm	*
Problem10-l33			sm	*
Problem10-l34			sm	*
Problem10-l35				*
Problem10-l36			ma, sm	*
Problem10-l37				*
Problem10-l38			ma, sm	*
Problem10-l51			sm	*
Problem10-l52				*
Problem10-l53				*
Problem10-l54			ma	*
Problem14-l20			am	am, sm
Problem14-l25				am, sm
Problem14-l30			sm	am, sm
Problem14-l35			sm	am, sm
Problem14-l36			sm	
Problem14-l38			sm	
Problem16-l23				*
Problem16-l24				*
Problem16-l25				*
Problem16-l26	(*: am, ma, sm)			*
Problem16-l40			ma	
Problem16-l47				*
Problem16-l48				*
Problem16-l49				*
Problem16-l50				*
Problem16-l55			ma	*
Problem16-l56			ma	*
Problem16-l57			am, ma	*
Problem16-l58				*
Problem16-l59			am	*
Problem18-l41				*
Problem18-l43				*
Problem18-l46				*
Problem18-l48				*
Problem18-l51				*
Problem18-l53				*
Problem18-l56				*
Problem18-l58				*
s3-clnt-1		cas		
s3-srvr-1				cas, mul
test-locks5				z
while-inf-3	z (*: am, ma, sm)			

approaches, depending on the difference between the inherent verification complexity of a CI's hardware implementation and the resulting complexity of the actual requirements extracted from the employed software analysis.

Our case studies have furthermore revealed the gap between the semantics of software and hardware verification, where the former sometimes all but ignores the physical limitations of the underlying processing units to increase the verification performance. This leads to an interesting line of future work that couples software and hardware analyses even closer, by bridging this gap in a way that a) does not impede the software analysis, e.g., by forcing it indiscriminately to bit-vector accuracy, b) allows for successful hardware verifications of custom instructions that realize the required functionality within the physically feasible limitations, and c) does not sacrifice the benefits of our coupling approach that enables us to verify software and hardware successively without a joint co-verification model. Building on such a joint model would require users to perform a full re-verification of the whole system for each new combination of software and custom instruction, therefore limiting the dynamic usage of automatically generated CIs. In contrast, our current approaches can enable the dynamic usage of automatically generated CIs, as the hardware verification in our approaches serves as legitimization of a previous software analysis' result.

Going one step further, this method could allow us to extend our current coupled verification scheme into a full software/hardware co-certification, integrating proof-carrying code [63] and proof carrying hardware [64], such that a user could replace their own verifications with a validation of the accompanying proofs of the software program and the RFU configuration, leading to the same level of trust with only a fraction of the computational effort.

APPENDIX A VERILOG PROPERTY CHECKER MODULE

Figure 14 shows a shortened version of the Verilog module that implements the property checker associated with the paper's running example. It illustrates the conversion of variables into Verilog wires based on the mapping generated between the CI's specification and variables used in pre- and postconditions (Lines 2-13). The conditions themselves and the output error flag can be found in Lines 15-26.

APPENDIX B BENCHMARK TASKS

See Table 5.

REFERENCES

[1] D. Beyer, "Software verification: 10th comparative evaluation (SV-COMP 2021)," in *Tools and Algorithms for the Construction and Analysis of Systems* (Lecture Notes in Computer Science), vol. 12652. Cham, Switzerland: Springer, 2021, pp. 401–422.

[2] L. De Moura and N. Bjørner, "Satisfiability modulo theories: Introduction and applications," *Commun. ACM*, vol. 54, no. 9, pp. 69–77, Sep. 2011.

[3] C. Barrett and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Model Checking*, M. C. Edmund, T. A. Henzinger, H. Veith, and R. Bloem, Eds. Cham, Switzerland: Springer, 2018, pp. 305–343.

[4] I. Wagner and V. Bertacco, *Post-Silicon Runtime Verification for Modern Processors*. Boston, MA, USA: Springer, 2011.

[5] C. Galuzzi and K. Bertels, "The instruction-set extension problem: A survey," *ACM Trans. Reconfigurable Technol.*, vol. 4, no. 2, pp. 1–28, 2011.

[6] N. Paulino, J. C. Ferreira, and J. M. P. Cardoso, "Improving performance and energy consumption in embedded systems via binary acceleration: A survey," *ACM Comput. Surv.*, vol. 53, no. 1, pp. 1–36, May 2020.

[7] M. Jakobs, M. Platzner, H. Wehrheim, and T. Wiersema, "Integrating software and hardware verification," in *Integrated Formal Methods* (Lecture Notes in Computer Science), vol. 8739. Cham, Switzerland: Springer, 2014, pp. 307–322.

[8] N. Clark, J. Blome, M. Chu, S. Mahlke, S. Biles, and K. Flautner, "An architecture framework for transparent instruction set customization in embedded processors," *ACM SIGARCH Comput. Archit. News*, vol. 33, no. 2, pp. 272–283, May 2005.

[9] A. C. S. Beck, M. B. Rutzig, and L. Carro, "A transparent and adaptive reconfigurable system," *Microprocessors Microsyst.*, vol. 38, no. 5, pp. 509–524, Jul. 2014.

[10] J. D. Souza, L. Anderson Sartor, L. Carro, M. B. Rutzig, S. Wong, and C. S. A. Beck, "DIM-VEX: Exploiting design time configurability and runtime reconfigurability," in *Applied Reconfigurable Computing Architectures, Tools, and Applications* (Lecture Notes in Computer Science), vol. 10824. Cham, Switzerland: Springer, 2018, pp. 367–378.

[11] M. Brandalero, M. Shafique, L. Carro, and A. C. S. Beck, "TransRec: Improving adaptability in single-ISA heterogeneous systems with transparent and reconfigurable acceleration," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2019, pp. 582–585.

[12] M. Damschen, M. Rapp, L. Bauer, and J. Henkel, "I-core: A runtime-reconfigurable processor platform for cyber-physical systems," in *Embedded, Cyber-Physical, IoT Systems: Essays Dedicated to Marilyn Wolf Occasion Her*. Cham, Switzerland: Springer, 2020, pp. 1–36.

[13] D. Beyer, T. Henzinger, and G. Théoduloz, "Configurable software verification: Concretizing the convergence of model checking and program analysis," in *Computer Aided Verification* (Lecture Notes in Computer Science), vol. 4590. Berlin, Germany: Springer, 2007, pp. 504–518.

[14] D. Beyer and M. Erkan Keremoglu, "CPAchecker: A tool for configurable software verification," in *Computer Aided Verification* (Lecture Notes in Computer Science), vol. 6806. Berlin, Germany: Springer, 2011, pp. 184–190.

[15] T. Ball, A. Podelski, and S. K. Rajamani, "Boolean and Cartesian abstraction for model checking C programs," *Int. J. Softw. Tools Technol. Transf.*, vol. 5, no. 1, pp. 49–58, Nov. 2003.

[16] E. Clarke, "Counterexample-guided abstraction refinement," in *Proc. 10th Int. Symp. Temporal Represent. Reasoning*, 2000, pp. 154–169.

[17] W. E. Dijkstra and S. Carel Scholten, "Predicate calculus program semantics," in *Texts and Monographs in Computer Science*. New York, NY, USA: Springer, 1990.

[18] D. Brand, "Verification of large synthesized designs," in *Proc. CAD*, Nov. 1993, pp. 534–537.

[19] A. Mishchenko, S. Chatterjee, K. Robert Brayton, and N. Eén, "Improvements to combinational equivalence checking," in *Proc. ICCAD*, 2006, pp. 836–843.

[20] E. Clarke and D. Kroening, "Hardware verification using ANSI-C programs as a reference," in *Proc. ASP-DAC*, 2003, pp. 308–311.

[21] L. Erkök, M. Carlsson, and A. Wick, "Hardware/software co-verification of cryptographic algorithms using cryptol," in *Proc. FMCAD*, 2009, pp. 188–191.

[22] K. McMillan, "Symbolic model checking: An approach to state explosion problem," Ph.D. dissertation, Dept. Comput. Sci., Carnegie Mellon Univ., Pittsburgh, PA, USA, May 1992.

[23] T. Isenberg, M.-C. Jakobs, F. Pauck, and H. Wehrheim, "Validity of software verification results on approximate hardware," *IEEE Embedded Syst. Lett.*, vol. 10, no. 1, pp. 22–25, Mar. 2018.

[24] T. Isenberg, M. Jakobs, F. Pauck, and H. Wehrheim, "When are software verification results valid for approximate hardware?" in *Tests and Proofs* (Lecture Notes in Computer Science), vol. 11823. Cham, Switzerland: Springer, 2019, pp. 3–20.

[25] C. Barrett, P. Fontaine, and C. Tinelli, "The SMT-LIB standard: Version 2.6," Dept. Comput. Sci., The Univ. Iowa, Iowa City, IA, USA, Tech. Rep., 2021. [Online]. Available: https://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-r2021-05-12.pdf

[26] D. Beyer, M. E. Keremoglu, and A. Wendler, "Predicate abstraction with adjustable-block encoding," in *Proc. FMCAD*, 2010, pp. 189–197.

[27] C. Wolf. *Yosys Open SYnthesis Suite*. [Online]. Available: https://yosyshq.net/yosys/

[28] K. Robert Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Computer Aided Verification* (Lecture Notes in Computer Science), vol. 6174. Berlin, Germany: Springer, 2010, pp. 24–40.

[29] A. Biere, "CaDiCaL at the SAT race 2019," in *Proc. SAT RACE*, 2019, pp. 8–9.

[30] G. A. Kildall, "A unified approach to global program optimization," in *Proc. 1st Annu. ACM SIGACT-SIGPLAN Symp. Princ. Program. Lang.*, 1973, pp. 194–206.

[31] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Berlin, Germany: Springer, 1999.

[32] V. D'Silva, D. Kroening, and G. Weissenbacher, "A survey of automated techniques for formal software verification," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 27, no. 7, pp. 1165–1178, Jul. 2008.

[33] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proc. 4th ACM SIGACT-SIGPLAN Symp. Princ. Program. Lang.*, 1977, pp. 238–252.

[34] D. Beyer and S. Löwe, "Explicit-state software model checking based on CEGAR and interpolation," in *Fundamental Approaches to Software Engineering* (Lecture Notes in Computer Science), vol. 7793. Berlin, Germany: Springer, 2013, pp. 146–162.

[35] P. Cousot and R. Cousot, "Systematic design of program analysis frameworks," in *Proc. 6th ACM SIGACT-SIGPLAN Symp. Princ. Program. Lang.*, 1979, pp. 269–282.

[36] A. Miné, "The octagon abstract domain," *Higher-Order Symbolic Comput.*, vol. 19, no. 1, pp. 31–100, Mar. 2006.

[37] D. Beyer, S. Löwe, and P. Wendler, "Sliced path prefixes: An effective method to enable refinement selection," in *Formal Techniques for Distributed Objects, Components, and Systems* (Lecture Notes in Computer Science), vol. 9039. Cham, Switzerland: Springer, 2015, pp. 228–243.

[38] M. Jakobs and H. Wehrheim, "Programs from proofs of predicated dataflow analyses," in *Proc. SAC*, 2015, pp. 1729–1736.

[39] S. Sankaranarayanan and F. Ivancic. *NECLA Static Analysis Benchmarks (Necla-Static-Small) V1.1*. Accessed: Nov. 30, 2013. [Online]. Available: http://www.nec-labs.com/research/system/systems_SAV-website/small_static_bench-v1.1.tar.gz

[40] K. Ku, E. Thomas Hart, M. Chechik, and D. Lie, "A buffer overflow benchmark for software model checkers," in *Proc. ASE*, 2007, pp. 389–392.
[41] D. Beyer, "Software verification with validation of results," in *Tools and Algorithms for the Construction and Analysis of Systems* (Lecture Notes in Computer Science), vol. 10206. Berlin, Germany: Springer, 2017, pp. 331–349.
[42] K. E. Murray, O. Petelin, S. Zhong, J. M. Wang, M. Eldafrawy, and J.-P. Legault, "VTR 8: High performance CAD and customizable FPGA architecture modelling," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 13, no. 2, pp. 1–55, May 2020.
[43] R. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenigun, "Verifying hardware in its software context," in *Proc. CAD*, 1997, pp. 742–749.
[44] J. Li, F. Xie, T. Ball, V. Levin, and C. McGarvey, "An automata-theoretic approach to hardware/software co-verification," in *Fundamental Approaches to Software Engineering* (Lecture Notes in Computer Science), vol. 6013. Berlin, Germany: Springer, 2010, pp. 248–262.
[45] D. M. Nguyen, "Formal hardware/software co-verification by interval property checking with abstraction," in *Proc. DAC*, 2011, pp. 510–515.
[46] A. Horn, M. Tautschnig, C. Val, L. Liang, T. Melham, J. Grundy, and D. Kroening, "Formal co-validation of low-level hardware/software interfaces," in *Proc. FMCAD*, Oct. 2013, pp. 121–128.
[47] B. Schmidt, C. Villarraga, J. Bormann, D. Stoffel, M. Wedler, and W. Kunz, "A computational model for SAT-based verification of hardware-dependent low-level embedded system software," in *Proc. 18th Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Jan. 2013, pp. 711–716.
[48] R. Mukherjee, M. Purandare, R. Polig, and D. Kroening, "Formal techniques for effective co-verification of hardware/software co-designs," in *Proc. DAC*, Jun. 2017, pp. 1–6.
[49] M. Schwarz, R. Stahl, D. Müller-Gritschneder, U. Schlichtmann, D. Stoffel, and W. Kunz, "ACCESS: HW/SW co-equivalence checking for firmware optimization," in *Proc. 56th Annu. Design Autom. Conf.*, Jun. 2019, pp. 1–6.
[50] R. Mukherjee, S. Joshi, J. O'Leary, D. Kroening, and T. Melham, "Hardware/software co-verification using path-based symbolic execution," *CoRR*, vol. abs/2001.01324, pp. 1–6, Jan. 2020.
[51] F. Xie, G. Yang, and X. Song, "Component-based hardware/software co-verification for building trustworthy embedded systems," *J. Syst. Softw.*, vol. 80, no. 5, pp. 643–654, May 2007.
[52] S. Loitz, M. Wedler, C. Brehm, T. Vogt, N. Wehn, and W. Kunz, "Proving functional correctness of weakly programmable IPs case study with formal property checking," in *Proc. SASP*, Jun. 2008, pp. 48–54.
[53] D. Groíe, U. Káhne, and R. Drechsler, "HW/SW co-verification of embedded systems using bounded model checking," in *Proc. 16th ACM Great Lakes Symp. VLSI*, 2006, pp. 43–48.
[54] M. H. Wu, P. C. Wang, C. Y. Fu, and R. S. Tsay, "An extended system C framework for efficient HW/SW co-simulation," *ACM Trans. Des. Automat. Electron. Syst.*, vol. 17, no. 2, pp. 1–16, 2012.
[55] P. Herber and S. Glesner, "A HW/SW co-verification framework for SystemC," *ACM Trans. Embedded Comput. Syst.*, vol. 12, no. 1s, pp. 1–23, Mar. 2013.
[56] B. Chen, K. Cong, Z. Yang, Q. Wang, J. Wang, L. Lei, and F. Xie, "End-to-end concolic testing for hardware/software co-validation," in *Proc. IEEE Int. Conf. Embedded Softw. Syst. (ICESS)*, Jun. 2019, pp. 1–8.
[57] J. Han and M. Orshansky, "Approximate computing: An emerging paradigm for energy-efficient design," in *Proc. 18TH IEEE Eur. TEST Symp. (ETS)*, May 2013, pp. 1–6.
[58] M. Carbin, S. Misailovic, and M. C. Rinard, "Verifying quantitative reliability for programs that execute on unreliable hardware," in *Proc. ACM SIGPLAN Int. Conf. Object Oriented Program. Syst. Lang. Appl.*, Oct. 2013, pp. 33–52.
[59] V. Fernando, K. Joshi, and S. Misailovic, "Verifying safety and accuracy of approximate parallel programs via canonical sequentialization," *Proc. ACM Program. Lang.*, vol. 3, pp. 1–29, Oct. 2019.
[60] B. Nongpoh, R. Ray, S. Dutta, and A. Banerjee, "AutoSense: A framework for automated sensitivity analysis of program data," *IEEE Trans. Softw. Eng.*, vol. 43, no. 12, pp. 1110–1124, Dec. 2017.
[61] S. He, S. K. Lahiri, and Z. Rakamarié, "Verifying relative safety, accuracy, and termination for program approximations," *J. Automated Reasoning*, vol. 60, no. 1, pp. 23–42, Jan. 2018.
[62] P. Roy, J. Wang, and W. F. Wong, "PAC: Program analysis for approximation-aware compilation," in *Proc. Int. Conf. Compil., Archit. Synth. Embedded Syst. (CASES)*, Oct. 2015, pp. 69–78.
[63] C. G. Necula, "Proof-carrying code," in *Proc. POPL*, 1997, pp. 106–119.
[64] S. Drzevitzky, U. Kastens, and M. Platzner, "Proof-carrying hardware: Towards runtime verification of reconfigurable modules," in *Proc. Int. Conf. Reconfigurable Comput. FPGAs*, Dec. 2009, pp. 189–194.

MARIE-CHRISTINE JAKOBS received the B.Sc., M.Sc., and Ph.D. degrees in computer science from Paderborn University, Germany, in 2009, 2012, and 2017, respectively.

From 2017 to 2019, she was a Research Assistant at the LMU Munich, Germany. Since 2019, she has been an Assistant Professor at the Technical University of Darmstadt, Germany. Her research interests include formal methods for automatic software verification, in particular incremental verification, cooperative verification, and validation of verification results.

FELIX PAUCK received the M.Sc. degree in computer science from Paderborn University, in 2017, where he is currently pursuing the Ph.D. degree. He started his work on the cooperative analysis of android apps by finishing his master's thesis. Currently, his main research interests include software analysis and benchmarks.

MARCO PLATZNER (Senior Member, IEEE) received the Diploma and Ph.D. degrees in telematics from the Graz University of Technology, Austria, in 1991 and 1996, respectively, and the Habilitation degree in hardware-software codesign from ETH Zürich, Switzerland, in 2002. He is currently a Professor for computer engineering at Paderborn University. Previously, he held research positions at the Computer Engineering and Networks Laboratory, ETH Zürich; the Computer Systems Laboratory, Stanford University, USA; the GMD—Research Center for Information Technology (now Fraunhofer IAIS), Sankt Augustin, Germany; and the Graz University of Technology. His research interests include reconfigurable computing, hardware-software codesign, and parallel architectures.

HEIKE WEHRHEIM received the Diploma degree in computer science from the University of Bonn, Germany, in 1992, the Ph.D. degree in computer science from the University of Hildesheim, in 1996, and the Habilitation degree from the University of Oldenburg, Germany, in 2002.

From 2004 to 2021, she was first as an Associate and then a Full Professor at Paderborn University, Germany. Since April 2021, she has been a Full Professor at the University of Oldenburg. She has published over 100 articles in journals and conferences. Her research interests include formal methods and software analysis, in particular the verification of concurrent data structures and software transactional memory algorithms on weak memory models.

Prof. Wehrheim is a member of the Gesellschaft für Informatik (GI) and the IFIP working group 6.1. She is on the editorial board of the journal *Formal Aspects of Computing*.

TOBIAS WIERSEMA received the M.Sc. and Ph.D. degrees in computer science from Paderborn University. He works as a Postdoctoral Researcher with the Computer Engineering Group, Paderborn University. His research interests include provably trustworthy remote verification of reconfigurable hardware circuits and FPGA overlays.