Systematic RISC-V based Firmware Design*

Vladimir Herdt[1] Daniel Große[1,2] Rolf Drechsler[1,2]
Christoph Gerum[3] Alexander Jung[3] Joscha-Joel Benz[3] Oliver Bringmann[3]
Michael Schwarz[4] Dominik Stoffel[4] Wolfgang Kunz[4]

[1]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
[3]Embedded Systems Department, Eberhard-Karls-University of Tuebingen, 72076 Tuebingen, Germany
[4]Department of Electrical and Computer Engineering, University of Kaiserslautern, 67663 Kaiserslautern, Germany

{vherdt, grosse, drechsle}@informatik.uni-bremen.de
{gerum, junga, benz, bringmann}@informatik.uni-tuebingen.de
{schwarz, stoffel, kunz}@eit.uni-kl.de

***Abstract*—Small embedded devices are highly specialized platforms that integrate several peripherals alongside the CPU core. Embedded devices extensively rely on *Firmware* (FW) to control and access the peripherals as well as other important functionality. This poses challenges to FW development since the FW must be adapted to each specific device configuration. Besides ensuring functional correctness to avoid errors and security vulnerabilities, an important design factor today is the control and adaptivity of a system with respect to non-functional properties, like for example application-specific timing budgets. Furthermore, optimizations of the FW and HW/SW interface play a very important role due to the tight resource constraints of small embedded devices. To satisfy these requirements new FW design methods are needed targeting FW generation, FW verification and FW optimization.**

This paper presents such new methods to enable an early, efficient and systematic FW design taking the underlying HW architecture into account. We use the RISC-V *Instruction Set Architecture* (ISA) as a case study to demonstrate our methods.

I. Introduction

Small embedded devices serve as basis to create *Internet-of-Things* (IoT) as well as automotive applications which are prevalent nowadays. Embedded devices integrate several peripherals alongside the CPU core and extensively rely on *Firmware* (FW) to configure, control and access the peripherals. Design flows for small embedded systems build on highly configurable platforms in order to minimize design costs and satisfy application specific requirements. This poses challenges to FW development since the FW must be adapted to each specific configuration. Therefore, automated FW-based methodologies are on the rise to enable generation, verification and optimization of FW. An important design factor today is the control and adaptivity of a system with respect to non-functional properties, like for example application-specific timing budgets. This requires appropriate timing models and needs to be considered by a FW generation approach. Furthermore, optimizations of the FW and HW/SW interface play a very important role due to the tight resource constraints of small embedded devices. Finally, FW verification is crucial to avoid errors and security vulnerabilities as well as to ensure that optimizations did not introduce any unwanted side effects.

*This contribution is funded as part of the CONFIRM project (project labels 16ES0565, 16ES0567 and 16ES0564K) within the research program ICT 2020 by the German Federal Ministry of Education and Research (BMBF) and supported by the industrial partners Infineon Technologies AG, Robert Bosch GmbH, Intel Deutschland AG, and Mentor Graphics GmbH.

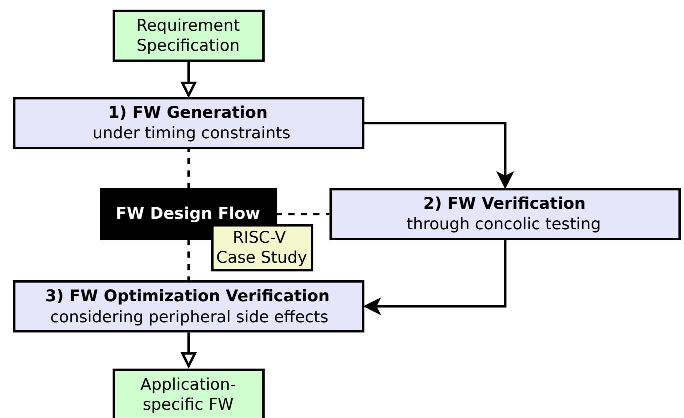

Fig. 1. Overview on our three approaches and their integration into a FW design flow.

Contribution: This paper showcases new automated methods towards enabling an early, efficient and systematic FW design which also takes the underlying HW architecture into account. In particular, we discuss three approaches that consider 1) generation, 2) verification and 3) optimization aspects of FW running on RISC-V based systems[1]. Fig. 1 shows an overview on how our approaches contribute to a FW-based design flow. In the following we discuss our three approaches in more detail.

1) We start in Section II by reviewing an efficient timing model for RI5CY processor cores implementing the RISC-V instruction set. An accurate timing model allows to obtain an accurate performance estimation and thus can be used to evaluate the quality of FW performance optimizations and enable FW generation under specific timing constraints. The timing model is based on a context-sensitive pipeline execution graph with integrated value analysis and considers properties of the underlying micro architecture. The approach is designed to work with both static and dynamic timing analysis approaches and is applied to FW timing simulation

[1]RISC-V is an open and free *Instruction Set Architecture* (ISA) that recently gained huge popularity for embedded systems. In particular, for IoT processors RISC-V is a game changer and meanwhile big companies start to adopt RISC-V and contribute to its steadily growing ecosystem. RISC-V is designed in a very modular and extensible way, supporting different register bitwidths and configurable instruction set extensions. For a comprehensive description of RISC-V please refer to the official specifications [1], [2].

at source and binary level. Furthermore, the impact of context-sensitive timing models using pipeline execution graphs is discussed in the context of an automated FW generation flow.

2) Then, in Section III, we present a novel case study on FW verification through concolic testing for RISC-V systems with peripherals and discuss the bugs we have found. Essentially, concolic testing successively explores new paths through the FW by solving symbolic constraints that are tracked alongside the concrete execution. This combination of concrete with symbolic execution enables an efficient exploration of a large set of different program paths and hence enables comprehensive testing of the FW. Based on the results of the FW verification case study, we discuss the open challenges and provide concrete next steps.

3) Optimization of the FW and HW/SW interface frequently alter the communication between the FW and the peripheral devices. In Section IV we review a designated HW/SW co-equivalence checking technique, *Access Compatibility Checking via Equivalent State Subsets* (ACCESS), tailored to ensure functional correctness of the optimized FW. Furthermore, certain FW optimizations can cause unwanted and subtle side effects in the accessed peripheral that are not immediately visible to the SW. We present new results on how to apply ACCESS for verifying correct I/O behavior of peripherals when both the FW and the HW peripheral are jointly optimized.

Related Work: Performance models are a fundamental part of most automated embedded SW generation flows designed to optimize performance or meet real-time requirements. While there are many performance models for other architectures at source level [3] [4] or binary level [5], [6] these are currently missing for the RISC-V architecture. In this paper we present a context-sensitive performance model for the RISC-V architecture that can be used for both source and binary code level performance estimations of firmware code.

Concolic testing has been shown very effective in the SW domain [7], [8], [9], [10], [11]. In addition, several extensions have been proposed to enable analysis of complex HW/SW interactions by integrating peripheral models (e.g. by using QEMU, Verilog or SystemC models) into the analysis [12], [13], [14]. This extensions are very important to enable FW verification, since embedded devices extensively rely on FW to configure, control and access the peripherals. We present a novel case study on FW verification through concolic testing for RISC-V systems with peripherals and discuss our findings.

FW optimizations often result in modifications of the HW/SW interface which may affect the FW as well as the HW in subtle and often unexpected ways. These effects may only result from the interaction between HW and SW. Hence, verification methods considering only the SW [15], [16] are insufficient. Methods using abstract HW models derived from RTL code [13], [17] may require extensive knowledge about the HW peripheral and may still miss subtle side effects.

II. A Context-sensitive Timing Model for Automated Firmware Generation onto RISC-V-based Microprocessor Platforms

In this section we present an accurate timing model that allows to obtain an accurate performance estimation and thus can be used to evaluate the quality of FW performance optimizations and enable FW generation under specific timing constraints.

This section is structured as follows. In Section II-A we give an introduction in our context sensitive timing and value analysis. Section II-B describes the application of our timing model to performance analysis of embedded FW, and in Section II-C we discuss the integration of the timing models in an automated FW design flow. Section II-D discusses the limitations of the current model and gives an overview of our current work.

A. Timing Model Generation

Our static timing analysis models a RISC-V system based on the *RI5CY* open source processor core integrated in a *Pulpino* based *System-on-a-Chip* (SoC) design [18]. This system has the following properties:

- 4-stage inorder pipeline
- 3-entry instruction fetch buffer
- 32-bit instruction set architecture
- static branch prediction
- data dependent instruction runtimes e.g. division and remainder operations take 2-32 cycles.
- single-cycle memory access

The timing model is based on a pipeline execution graph model. In Fig. 2 we show an example of a pipeline execution graph for four instructions on the RI5CY pipeline. In the pipeline execution graph based model the flow of each instruction through the 4-stage pipeline is represented, by the four nodes of the pipeline execution graph on the vertical axis. The latencies inherent to the pipelined execution of each instruction on the pipeline are shown by black edges. As the first instruction in the instruction flow is a variable latency instruction the pipeline analysis would assume a pessimistic timing, which is a 32 cycle delay from between the start of execute and the start of writeback. All other instructions have a single cycle delay between the pipeline stages (for brevity we omit +1 labels in Fig. 2).

Resource dependencies between instructions are modeled by the vertical grey edges in Fig. 2. As RI5CY uses an inorder pipeline, these edges are always to the node of the next instruction in program order, and have a delay of one cycle. As the pipeline employs forwarding of operation results from the output of execute stage to the decode stage, true data dependencies are modeled by the red edges from the execute stage to the decode stage of the dependent instruction. The latencies are reduced by one compared to the actual latencies, to account for the time spent processing in instruction decode.

The green edges are used to model the limited size of the instruction fetch buffer and mispredicted branches are modeled by the blue edges from the execute stage of a taken branch to instruction at the branch target.

Timing analysis is then carried out on the binary level control flow graph, of the application under investigation. The analysis builds pipeline execution graphs for each pair of basic blocks, and initializes each start time of each node to zero. To obtain the final execution time the model then iterates over each state in the pipeline execution graph and calculates the earliest start times for each node by the maximum over the start times of the predecessors added with the latencies on the corresponding edges. The fixpoint iteration is finished when the start times for each state do not change anymore.

To better cover value dependent instruction timings we integrated a context sensitive value analysis into our timing model. The analysis is based on *abstract interpretation* of the binary code. The analysis uses an interval abstract domain implementation [19] and calculates value ranges for the RISC-V architecture, by implementing the semantics of the RISC-V integer instructions on abstract intervals instead of concrete values. The results of the timing analysis can then be used during the analysis of the pipeline

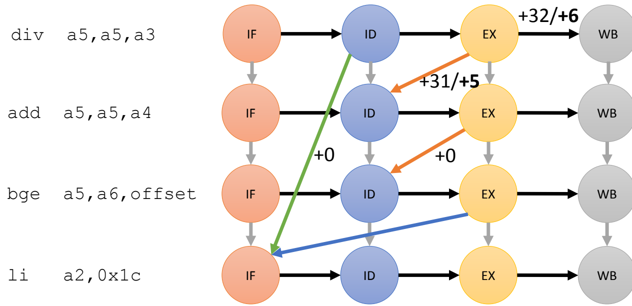

Fig. 2. Example of the pipeline execution graph based analysis (default latencies/**value constrained latencies**)

execution graphs to generate tighter bounds for the execute nodes if the analysis found a bound for the register used as divisor, as shown by the numbers in bold font in Fig. 2. Memory accesses are currently analysed using a fixed delay per memory region.

B. Performance Analysis

The performance models can then be used to evaluate the performance of embedded FW designs at multiple abstraction levels. We currently support two analysis types.

- **Binary-Level Dynamic Analysis:** As the performance model provides context sensitive timings for each binary level basic block the integration of timing models in a functional *Instruction Set Simulator* (ISS) is straight forward. The timing simulator simply needs to simulate the current callstack and query the timing model at each basic block boundary to accumulate the basic block timings to the final execution time of a program or SW component.
- **Source-Level Dynamic Analysis:** As the performance model provides context-sensitive timings at the binary level the performance estimation at the source level needs to relate source level execution paths with the binary level execution paths. For this purpose we implemented a source to binary matching heuristic according to [20] and use it for our source level dynamic analysis of the target specific execution times. The structure of the source level performance simulation is as follows:
  1) The source code of an application is first translated by a compiler for the RISC-V architecture.
  2) Then we construct a control flow graph from the source code as well as from the binary code.
  3) Both control flow graphs are used to match the corresponding basic blocks at source and binary level.
  4) The results of the binary to source matching and the timing model from the previous section are used to create a version of the original source code, that contains additional function calls to simulate the paths through the binary level CFG. These annotations enable a fast simulation of the performance behavior of an embedded application, through execution of annotated source code on an arbitrary host machine.

We evaluated the results of the binary- and source-level dynamic analyses on benchmarks from the *malardalen* benchmark suite [21].

All benchmarks were compiled with the official *GCC* compiler toolchain for RISC-V version 5.2 with optimization level *-Os*. As reference timings, we used the execution times of the application binaries on a *Register-Transfer-Level* (RTL) simulation of a Pulpino-SoC. Table I shows a comparison of the execution time estimates of on the source and binary performance analyses, to the actual exection times from the RTL simulations. The results without a value analysis are shown in Column 2 for the binary level performance analysis and Column 4 for the source level performance analysis. Columns 3 and 5 show the corresponding results with value analysis.

TABLE I
EXPERIMENTAL RESULTS: DEVIATION OF ESTIMATED EXECUTION TIMES FROM THE RTL SIMULATION (NV=NO VALUE ANALYSIS, V=VALUE ANALYSIS)

Benchmark	Binary Level nv	Binary Level v	Source Level nv	Source Level v
insertsort	00.13%	00.13%	00.13%	00.13%
fdct	$<$ 00.01%	$<$ 00.01%	$<$ 00.01%	$<$ 00.01%
bsort100	$<$ 00.01%	$<$ 00.01%	$<$ 00.01%	$<$ 00.01%
jfdctint	32.92%	$<$ 00.01%	32.92%	$<$ 00.01%
cnt	24.70%	$<$ 00.01%	32.75%	08.75%
matmult	07.85%	00.22%	09.19%	01.22%

The benchmarks in the upper half (*insertsort*, *fdct*, *bsort100*) of the table do not contain any instructions with data dependent execution timings. On these benchmarks we achieve nearly cycle accurate performance estimations in all analyses. While the benchmarks in the lower half of the table (*jfdctint*, *cnt*, *matmult*) contain variable instruction latencies. In these cases the pipeline execution graph based performance model benefits from our value analysis. In the binary level analysis we also achieve nearly cycle accurate performance estimations, while the performance estimations using source level analysis contain still higher errors in case of *cnt* and *matmult*. These errors can be attributed to an incomplete source to binary matching.

C. Firmware Generation under Timing Constraints

Our timing model and dynamic analysis is currently integrated in an automated FW design workflow similar to [22]. The automated design flow performs four major steps.

First a data-flow-graph based model of the SW is transformed to C source code, of the embedded FW.

Next, an analysis step is executed. This step consists of timing simulation of the executed FW using the presented timing model as well as power simulation. The timing analysis interfaces with the ISS to get the binary basic block execution order. Hence, the start address of each executed basic block is passed to the analysis for an online context-sensitive timing analysis. In addition to the obligatory total simulated execution time of the FW, timing analysis optionally yields a report, containing an evaluation of specified timing constraints.

The analysis results are then passed to an optimizer to optimize the mapping of data to memory banks and to insert transitions to low-power memory modes for specific memory banks.

In the final step, the resulting, optimized binary is executed using the dynamic binary instrumentation to re-run the power and timing simulation. The resulting timing and power information could then be used to further improve the design in another iteration of generation and optimization steps or for further manual optimizations of the resulting FW.

D. Next Steps: Context-Sensitive Timing Model

We have presented a context sensitive timing model for *RI5CY* based SoCs. The timing model reaches high accuracies when combined with a binary level functional simulation and can also reach high accuracies when using source level simulations, although the reconstruction of binary paths on the source level might introduce some divergencies from the actual timing. Our future steps are:

1) Since the RISC-V architecture can be easily extended, we also want to increase the extensibility of our performance model. In particular, we want to create possibilities for simple modeling of domain-specific instruction set extensions.
2) We would like to further improve, the implementation of our source level timing annotation to make the source level performance estimations to reduce the remaining gap with the binary level performance estimations.
3) Improve the integration of the timing models in the FW generation flow to directly guide optimized code generation and optimizations.
4) Additionally, we intend to use our timing models in a case study on FW generation for a RISC-V based low power smart speaker application.

III. FW Verification trough Concolic Testing for RISC-V Systems

In this section we present a case study on FW verification through concolic testing for RISC-V systems with peripherals. We employ the concolic testing approach from [14] which we briefly review in Section III-A. Then, we present the results of our case study (Section III-B) and discuss limitations and possible extensions (Section III-C and Section III-D).

A. Approach Overview: Review on Concolic Testing for RISC-V based Systems with Peripherals

Fig. 3 shows an overview. The approach is centered around a *Concolic Testing Engine* (CTE) that enables concolic testing of RISC-V binaries (middle of Fig. 3). The RISC-V binary is obtained by compiling and linking the SW/FW application together with a designated CTE-interface library (left side of Fig. 3). Variables, representing input data, are marked to be symbolic (using the CTE-interface – by leveraging RISC-V system calls). The exploration engine starts by assigning each symbolic variable a random value (first input) and then successively generates new inputs (based on the observed execution constraints) to explore different paths through the RISC-V binary (right side of Fig. 3). For each input the RISC-V binary is executed on a *Virtual Prototype* (VP). The VP tracks symbolic constraints (i.e. branch conditions, assumptions and assertions) alongside the concrete execution in order to generate new inputs in the exploration engine. The VP essentially consists of an *Instruction Set Simulator* (ISS), a memory, and a bus system. Additional peripherals are integrated through a SW library (that is compiled alongside the SW/FW application and also executed on the VP). Essentially, peripherals are registered by their address range on the VPs bus and hence the VP can automatically route memory access operations accordingly. To call a function in a SW model the VP sets the program counter to the functions address and restores it again after the function has finished. Arguments and results are passed through registers and designated memory regions.

The exploration engine continues until all inputs have been processed (i.e. all feasible paths through the RISC-V binary have been explored) or a runtime check fails (e.g. SW assertion violation

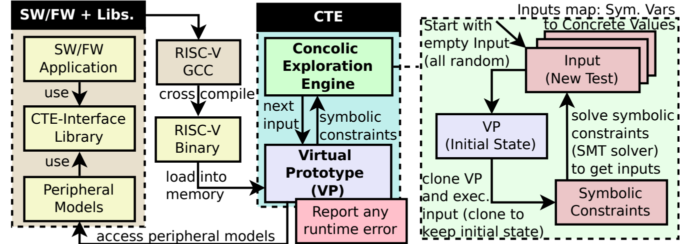

Fig. 3. Overview on Concolic Testing of Embedded RISC-V Binaries with Virtual Prototypes as shown in [14]

or generic memory error). Symbolic constraints are solved using an SMT solver.

B. Case Study on FW Verification using Concolic Testing: Description and Results

We have implemented the concolic testing approach for RISC-V binaries in a tool called CTE and used CTE for testing different FW. CTE is based on our open-source RISC-V VP[2]. All experiments are performed on an Ubuntu 16.04 Linux system with an Intel Core i5-7200U processor with 2.5 GHz. We use KLEE [7] v1.4.0 with STP [27] solver v2.3.1 as symbolic backend in CTE. As a case study we consider three different scenarios:

1) FW that copies data from one device to another (*uart-copy*).
2) FW that logs a warning on specific sensor values to an output device (*sensor-log*).
3) FW that periodically queries a sensor to control a fan (*fan-control*).

We found a bug in each FW using CTE and then fixed the bug. Table II shows a summary of the verification results. The columns show: the scenario (i.e. application FW), the number of executed instructions (*#instr*), lines of code in C and assembly (ASM), overall execution time (*time* in seconds), solver time (*s-time* in seconds), number of concolic execution paths (*#paths*), and number of solver queries (*#s-queries*). For each scenario we report results for the buggy (name: *bug) and fixed FW (name: *ok) version. In case of a bug, CTE stops and reports a counterexample (i.e. concrete inputs for the symbolic variables) to reproduce the bug. Otherwise (i.e. no bug is detected), CTE performs a comprehensive state space exploration based on the symbolic inputs. It can be observed that CTE is very effective for both use cases, bug finding as well as exhaustive exploration. In the following we provide more details on the three scenarios that we consider and the bugs we have found in the FW.

Scenario 1) *uart-copy*: In the first scenario we consider a FW driver that copies data from one device to another through an internal ring buffer (size=32). In particular, we consider two UART devices, one as the source and the other as target, respectively. The FW code can be triggered by an interrupt when new data is available for copying. The FW iterates until all data has been copied, i.e. the source UART returns that no more elements are available. In each step the ring buffer of the FW is filled by querying the source UART and then the ring buffer content is transferred to the target UART. The FW also supports the *copy_size*

[2]The VP is implemented in standard-compliant SystemC and TLM-2.0 [23], [24], is designed as extensible and configurable platform with a generic bus system [25], and the ISS has been verified thoroughly [26]. The VP is available under MIT licence. Visit http://www.systemc-verification.org for our most recent VP-based approaches.

TABLE II

EXPERIMENT RESULTS - USING CTE TO ANALYZE FW OF RISC-V SYSTEMS. IN CASE OF A BUG (*BUG) CTE STOPS THE ANALYSIS AND REPORTS A COUNTEREXAMPLE. OTHERWISE (*OK), CTE PERFORMS AN EXHAUSTIVE CONCOLIC EXECUTION BASED ON THE SYMBOLIC INPUTS TO THE FW.

Scenario (App. FW)	#instr	C	ASM	time (sec.)	s-time (sec.)	#paths	#s-queries
1) uart-copy bug	1,541,315	269	949	72.34	65.26	34	8721
1) uart-copy ok	1,529,477	269	950	74.85	67.74	32	8713
2) sensor-log bug	4,490	261	869	0.59	0.43	15	28
2) sensor-log ok	616,404	261	869	78.47	62.51	1021	1787
3) fan-control bug	2,638	295	1019	0.28	0.15	12	16
3) fan-control ok	63,070,763	295	1019	1196.12	616.98	37666	73486

options to configure the maximum number of elements copied in each step.

We use a testbench that checks that the data received at the target UART corresponds to the data from the source UART (same data received in the same order). The source UART returns elements from an input array with 256 symbolic elements one after another when queried. Afterwards, the source UART stops returning data. Once the target UART has received all 256 elements, the testbench terminates the execution.

We found a bug in the ring buffer implementation of the FW. The function that checks if the buffer is full did not work correctly for the case when the writing position of the ring buffer is set on the last element. Thus, the FW did erroneously overwrite active elements in the ring buffer in this case.

Scenario 2) *sensor-log*: The second scenario considers a FW that logs specific sensor data to an output device. The sensor generates periodic interrupts to notify the FW that a new data frame is available. A data frame consists of new 8 values representing the measurements for the last time interval. The FW reads the sensor frame into an internal buffer and then computes the maximum of the values. In case the maximum value is above a pre-defined treshold, then the FW writes the maximum value alongside a warning to the output device. The output device buffer is flushed by the FW after the write process finished and when the device buffer is full. The size of the device buffer is obtained by the FW during initialization.

We have created a testbench that simulates a single time interval and returns a symbolic data frame, constrained to valid sensor values, when the sensor is queried. We have added assertions that check that the sum and maximum value of the data frame elements is within a valid range. By applying CTE, we detected a bug where the FW writes beyond the output device buffer, hence causing a memory error in the device. The reason is that the FW initialization code obtained the device buffer size not correctly (the size is encoded in the lower 8 bit of the 32 bit result, but the FW used the upper 8 bit). Depending on the selected threshold only a very specific combination of sensor values triggers the bug. Concolic testing (or symbolic execution in general) is very well suited to detect such bugs.

Scenario 3) *fan-control*: The third scenario considers a FW that controls a fan based on a sensor temperature value. Similar to Scenario 2, the sensor works by periodically triggering an interrupt and providing a new data frame in each step. The FW is triggered at each sensor interrupt. It first copies the sensor data frame (i.e. temperature values observed in the last time interval) into an internal buffer. Then, the FW computes the average value of the values stored in the buffer. The fan speed (3=high to 0=off) is adjusted based on the current average temperature value in combination with the two last observed values. The high speed setting is immediately selected in case a high temperature value is observed. Lower speed settings (off in particular) require that the last two observed temperature values were also within the lower range.

We created a testbench and added assertions to check that the average value stays within a valid range (between the minimum and maximum sensor temperature value) and we capture the last sensor frame in the testbench to assert that the FW computed fan speed setting does not violate the last sensor measurement (i.e. a high speed setting is selected for a large value and the off setting is not used in case the value is above a certain threshold). The sensor is configured to return a data frame of symbolic elements with each element constrained to stay within a valid temperature range. In addition, the last and second last observed average temperature values are also set to symbolic values. We have set the number of simulated time intervals to four to ensure that the execution terminates and enough time intervals are executed (since the fan control depends on multiple time intervals).

We found a bug in the FW. The loop that computes the average temperature value reads one element beyond the internal buffer (due to an off-by-one loop condition). This results in a read of an uninitialized variable on the stack after the buffer. Hence, our CTE assumes that the uninitialized stack variable can have an arbitrary value. This results in a temperature value overflow (i.e. even though the sensor returned a high value, the FW computes a negative value) and thus the fan is erroneously not activated.

C. Discussion: Buffer Overflow Detection

Please note, that in Scenario 3 CTE did not report the buffer overflow immediately in the FW but only found the error because the assertion failed in the testbench checking code. The reason is that we perform a binary analysis, which albeit being precise (the binary is the final code that will be deployed) lacks information that the actual source code provides. At the source level a buffer and a subsequent variable are placed on the stack. At the binary level the stack pointer is simply adjusted to allocate stack space, hence the information what objects are allocated are lost. Therefore, the buffer overflow is not immediately detected because it is a valid access inside the allocated stack space.

One possible solution to detect buffer overflows at binary level is to instrument the binary with source level information during compilation. GCC and LLVM provide several *sanitizer* to perform such an instrumentation in order to e.g. detect memory leaks and buffer overflows. Sanitzers are complementary to concolic testing (for example we can run CTE to generate a comprehensive testset and then re-run every test with different sanitizers). However, sanitizers are not always available for every architecture (e.g. RISC-V is not yet supported), in particular for new architectures or architectures targeting embedded systems.

Another possible solution that is particularly suited for heap buffer overflow detection and works without extensive compiler

support is to wrap existing memory allocation functions (i.e. malloc and free). Many embedded systems do not rely on the standard C library implementation but provide custom memory management functions instead. GCC for example provides linker flags that allow to automatically redirect calls to existing functions to a custom user defined function (*Wl,–wrap* flag). The custom malloc function allocates a larger memory block than has been requested by adding extra bytes (protected zones) before and after the actual requested memory block. These protected zones are then registered in the verifier, as supported by CTE, which will monitor every memory access and report a buffer overflow error in case of a read or write access into a protected zone. The custom free function unregisters the protected zones and calls the real free function afterwards. In addition, it checks for double free and non-allocated blocks.

D. Next Steps: Improve Concolic Testing of FW

CTE is already very effective for testing RISC-V based FW. To further improve it we plan to consider three different directions:

1) Add support for the 64 bit RISC-V ISA and additional ISA extensions (e.g. floating point instructions) to support a broader range of RISC-V systems.
2) Incorporate further state-of-the-art symbolic exploration techniques to speed-up the verification process. In particular, we plan to integrate sophisticated state space exploration heuristics which are very important to speed up the bug finding proces in very large state spaces. In addition, we want to integrate dynamic state merging techniques to alleviate the state space explosion problem.
3) Integrate the RISC-V processor timing model (see Section II) to obtain accurate timing results for systems with peripherals and enable checking timing related properties alongside the concolic testing.
4) Incorporate dynamic binary translation techniques to significantly boost the execution performance with native execution [28] and consider advanced symbolic state subsumption techniques to detect re-exploration of symbolic states [29].

IV. Checking for Peripheral Device Side Effects in Firmware Variants

In this section we present a case study on formal verification of FW variants resulting from optimizations of the HW/SW interface. The case study extends the application of ACCESS over [30] to cases where not only the FW but also the HW peripherals are modified by the optimization process.

A. Optimizations of the HW/SW Interface

Load/store instructions are, compared with arithmetic instructions, expensive in terms of energy and execution time. They always involve some action by secondary HW (e.g. bus systems) contributing to latency and power consumption. Additionally, a large portion of an embedded system's chip area is occupied by memory.

Optimizations that change the layout of bit fields and registers in the system's HW/SW interface may lead to fewer accesses and reduce the FW's memory footprint. This may have noticeable impact on the system's HW cost and power consumption. We can increase code sharing between API functions by unifying device access patterns. However, this may change the I/O behavior of the FW w.r.t. a particular peripheral device because the new API function uses a different number and/or ordering of load/store accesses.

In general, a compiler is unable to perform such optimizations, due to its lack of detailed knowledge about the system's HW. In FW, accesses to HW are usually done via volatile pointers. These are excluded from compiler optimization. Instead, the FW developer or HW designer makes such adjustments manually when customizing the platform, or by additional tools such as code generators.

Such optimizations are typically restricted to local modifications of the I/O behavior of the program but do not change its global control flow. Yet, they pose significant challenges to designers and SW developers. They may easily compromise the functional correctness of the I/O behavior of the embedded system, e.g., by triggering unanticipated side effects in the HW.

B. ACCESS Verification Method

The goal of *ACCESS* is to formally prove functional equivalence between two variants of a HW/SW system, i.e., certify that both variants exhibit equivalent I/O behavior w.r.t. the system environment.

Common notions of equivalence like *sequential HW equivalence* or (unrestricted) *HW/SW co-equivalence modulo latency* are either too restrictive, or computationally too complex for the optimizations set out in Section IV-A. In order to make computational complexity tractable, ACCESS partitions the FW variants into segments such that a bijective mapping between segments exists and each pair of corresponding segments is expected to produce equivalent I/O behavior. This allows us to prove the equivalence between two mapped segments within a relatively small number clock cycles using the miter structure shown in Fig. 4. It is, essentially, an unrolling of the peripheral device logic into a number of time frames (similar as in *Bounded Model Checking* (BMC)) such that different access patterns can be applied and evaluated.

Each FW variant is modeled by an instance of a *Program Netlist* (PN) [31], i.e., a combinational circuit representing all possible execution paths in the FW. For each pair of mapped segments the accessed peripheral is unrolled from an arbitrary, but identical, initial state. The length of the unrolling depends on the number of accesses in the mapped segments, the maximum time interval allowed between accesses and a "grace period" allowed for reaching an equivalent state. Equivalence of two mapped segments is shown if both unrollings lead to identical HW states at some point within the grace period while producing identical I/O behavior with the environment at all considered time points.

The key idea of ACCESS lies in how the HW/SW segments are created: by over-approximating the initial state of the HW we ensure that each proof is independent of prior HW/SW interactions outside a considered segment. Every HW state resulting from such an interaction is already contained in the over-approximation. At the same time, each unrolled HW segment is conjoined with the *complete PN*. This ensures that the proof for each cycle-accurate HW segment is performed for the right access sequences and takes into account the precise context of the executing FW. In this way, the HW/SW equivalence of the entire system can be proven step by step by proving HW/SW equivalence for the individual segments. In the following case study the method is applied to a system in which both sides of the HW/SW interface have been modified.

C. Case Study: HW/SW optimized Soft-SPI Implementation

We present results of a case study in checking HW/SW co-equivalence between variants of an interrupt driven *Software-implemented Serial Peripheral Interface* (Soft-SPI) slave for the RISC-V based PULPino platform. All experiments were run on an

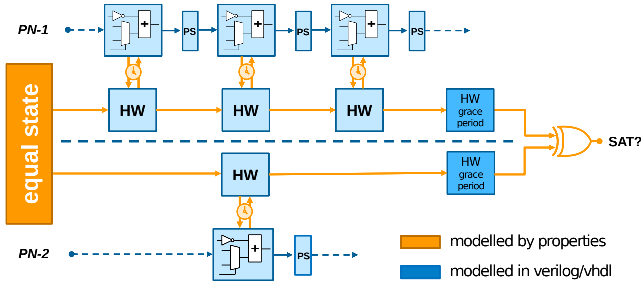

Fig. 4. ACCESS computational model to prove equivalence of two I/O sequences accessing a peripheral as shown in [30]

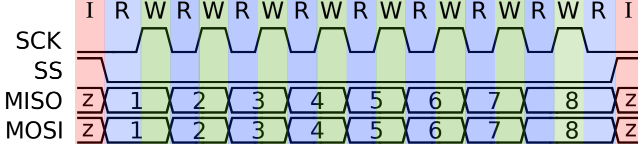

Fig. 5. SPI Protocol Timing Diagram. Colors highlight different phases as used in the Soft-SPI.

Intel® Core™ i7 @ 3.40 GHz × 8 with 32 GB of RAM, running an Ubuntu 16.04 Linux system and the OneSpin® 360 Design Verification tool.

The Soft-SPI emulates a SPI slave peripheral via "bit banging" on the PULPino *General Purpose Input/Output* (GPIO) peripheral. The protocol requires four dedicated I/O pins: *Synchronous Clock* (SCK), *Slave Select* (SS), *Master-In-Slave-Out* (MISO) and *Master-Out-Slave-In* (MOSI). Fig. 5 shows the timing diagram of the protocol for an 8-bit duplex transmission as well as the three phases defined by the Soft-SPI slave: *Idle* (I), *Prepare-for-Read* (R) and *Prepare-for-Write* (W).

The functions of the GPIO pads are defined by four control registers. Each control register holds four bits that are each associated with one of the four GPIO pads. Each pad can be configured independently by setting the bits associated with it in the four registers. Register *PADDIR* sets a pad's function as input or output, register *INTEN* enables interrupts, and registers *INTTYPE0* and *INTTYPE1* control the interrupt triggering behavior. A GPIO pad needs to be initialized after reset and/or reconfigured according to the current phase of the Soft-SPI. Table III shows the configurations for each pad in each protocol phase.

TABLE III
GPIO PAD CONFIGURATIONS FOR THE SOFT-SPI PHASES

Phase	Pad	PADDIR	INTEN	Trigger*
Idle (I)	SCK	IN	disabled	high
	SS	IN	enabled	low
	MISO	IN	disabled	-
	MOSI	IN	disabled	-
Prepare-for-Read (R)	SCK	IN	enabled	high
	SS	IN	enabled	high
	MISO	OUT	disabled	-
	MOSI	IN	disabled	-
Prepare-for-Write (W)	SCK	IN	enabled	low
	SS	IN	enabled	high
	MISO	OUT	disabled	-
	MOSI	IN	disabled	-

*high: INTTYPE0=0, INTTYPE1=1, low: INTTYPE0=1, INTTYPE1=1

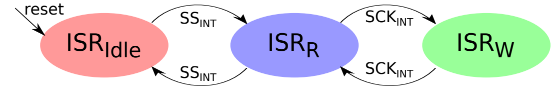

Fig. 6. Soft-SPI call graph

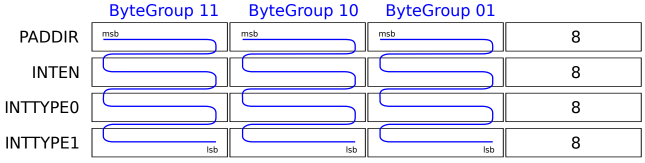

Fig. 7. Optional GPIO control register addressing scheme after HW modification.

The Soft-SPI defines one *Interrupt Service Routine* (ISR) for each phase, ISR_{idle}, ISR_R and ISR_W. Besides servicing interrupts, ISR_{idle} is also used for initialization of the peripheral. ISR_R is called for two different interrupt events, depending on the driver state. Fig. 6 shows the resulting FW call graph.

We examined three variants of the Soft-SPI: *unpacked*, *SW-packed* and *HW/SW-packed*. All use the same set of GPIO pads. Because the software uses only a subset of the available pads, each write access to a register must be preceded by a read access and by creation of an appropriate bit-mask for selecting the used pads. This prevents the write accesses from affecting pads that are possibly used by other applications. In the sequel, we will refer to these two steps as a joined *Read-Modify-Write* (RMW) operation. All variants require one RMW to write data to MISO in ISR_R, a normal read access to MOSI in ISR_W as well as a read access of the interrupt status register at each interrupt. These are not included in the following descriptions.

The first variant, *unpacked*, uses the GPIO driver provided with the PULPino platform, in which each bit in a GPIO register must be set by its own RMW. This requires at least 12 RMW operations in ISR_{idle}, three in ISR_R and one in ISR_W.

The next variant *SW-packed* exploits the fact that the FW can simultaneously configure the same control parameter for multiple pads in a single RMW operation. Hence, the RMW operations in ISR_{idle} are reduced to 4 (one for each control register). ISR_R and ISR_W remain unaffected because the accesses are all distributed over different registers, leaving no room for optimizations.

While in variant *SW-packed* only the FW was optimized, both the FW and the HW are modified in variant *HW/SW-packed*. The optimization achieved in variant *HW/SW-packed* is possible due to changes in the GPIO's addressing scheme. In the original addressing scheme the two least significant address bits are not used. We utilized these to introduce a new optional addressing mode in the peripheral's HW. It is without effect when both bits are 0, but otherwise allows to simultaneously access one byte of each of the four control registers. The accessed byte is defined by the values of the address bits. The new scheme is conceptually presented in Fig. 7. Due to the new addressing scheme, all pads of the same byte can be completely configured using only a single RMW. Hence, only one RMW is required by this variant in each ISR.

Fig. 8 shows the normalized effect of the optimization on the amount of accesses required to reconfigure the GPIO in order to transmit a certain number of bits. We can see that both optimized variants have a significant effect on the number of required I/O accesses, especially for a low number of transmitted bits.

In order to verify the equivalence of the variants with ACCESS,

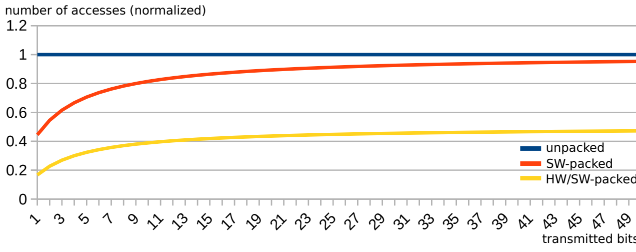

Fig. 8. Normalized number of configuration accesses to the GPIO by the Soft-SPI variants to transmit x data bits.

TABLE IV
EXPERIMENT RESULTS USING THE ACCESS VERIFICATION METHOD PROVING EQUIVALENCE BETWEEN SOFT-SPI VARIANTS

variant combination	segment	time	memory	result
unpacked	ISR_{idle}	21 min 28 sec	2881 MB	hold
⇔	ISR_{R}	3 min 07 sec	2148 MB	hold
SW-packed	ISR_{W}	2 min 08 sec	1347 MB	hold
unpacked	ISR_{idle}	12 min 07 sec	2539 MB	hold
⇔	ISR_{R}	1 min 58 sec	1794 MB	hold
HW/SW-packed	ISR_{W}	35 sec	1357 MB	hold
SW-packed	ISR_{idle}	3 min 15 sec	1939 MB	hold
⇔	ISR_{R}	2 min 14 sec	1793 MB	hold
HW/SW-packed	ISR_{W}	33 sec	1357 MB	hold

the division of the code into ISRs provides a natural segment mapping. During PN generation the maximum number of bits per transmission was set to one. This is sufficient to exhaustively explore all transitions in the FW call graph. The HW modifications to the GPIO do not add or remove any of HW ports or registers. Hence, no additional action has to be taken to map HW states.

All variants could be proven to be equivalent to each other under the equivalence notion of ACCESS. Proofs were performed assuming a maximum delay of 4 cycles per access and a grace period of 5 cycles. Table IV shows more detailed data on the individual proofs.

Our results show the potential of application-specific optimizations of the HW/SW interface of peripherals. Low-level FW optimizations such as re-structuring the register interfaces and the access patterns of FW, as they are often practiced by FW developers and HW designers, typically involve only local modifications of FW code and HW structures. At the same time, however, they require a global understanding of the FW behavior. The ACCESS approach provides a new instrument for encouraging and systematically verifying such optimizations.

V. CONCLUSION

Automated FW-based methodologies are very important to cope with the rising complexity of the design flow for embedded systems. In this paper we showcased new automated methods towards enabling an early, efficient and systematic FW design which also takes the underlying HW architecture into account. In particular, we discussed three approaches that consider generation, verification and optimization aspects of FW. Our evaluations, using the RISC-V ISA as a case study, demonstrated the efficiency and applicability of our methods.

REFERENCES

[1] A. Waterman and K. Asanović, *The RISC-V Instruction Set Manual; Volume I: User-Level ISA*, SiFive Inc. and CS Division, EECS Department, University of California, Berkeley, 2017.

[2] A. Waterman and K. Asanović, *The RISC-V Instruction Set Manual; Volume II: Privileged Architecture*, SiFive Inc. and CS Division, EECS Department, University of California, Berkeley, 2017.

[3] S. Ottlik, C. Gerum, A. Viehl, W. Rosenstiel, and O. Bringmann, "Context-sensitive timing automata for fast source level simulation," in *DATE*, 2017.

[4] Z. Zhao, A. Gerstlauer, and L. K. John, "Source-level performance, energy, reliability, power and thermal (PERPT) simulation," in *TCAD*, 2017.

[5] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," in *SIGARCH Comp. Arch. News*, 2011.

[6] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *TACO*, vol. 11, no. 3, pp. 28:1–28:25, 2014.

[7] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*, 2008, pp. 209–224.

[8] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *NDSS*, 2008.

[9] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *IEEE S & P*, 2012, pp. 380–394.

[10] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: a platform for in-vivo multi-path analysis of software systems," in *ASPLOS*, 2011, pp. 265–278.

[11] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Krügel, and G. Vigna, "SOK: (state of) the art of war: Offensive techniques in binary analysis," in *IEEE S & P*, 2016, pp. 138–157.

[12] S. Ahn and S. Malik, "Automated firmware testing using firmware-hardware interaction patterns," in *CODES+ISSS*, 2014, pp. 25:1–25:10.

[13] R. Mukherjee, M. Purandare, R. Polig, and D. Kroening, "Formal techniques for effective co-verification of hardware/software co-designs," in *DAC*, 2017, pp. 35:1–35:6.

[14] V. Herdt, D. Große, H. M. Le, and R. Drechsler, "Early concolic testing of embedded binaries with virtual prototypes: A RISC-V case study," in *DAC*, 2019, pp. 188:1–188:6.

[15] D. Currie, X. Feng, M. Fujita, A. J. Hu, M. Kwan, and S. Rajan, "Embedded software verification using symbolic execution and uninterpreted functions," *Int. J. Parallel Program.*, vol. 34, pp. 61–91, 2006.

[16] C. Villarraga, B. Schmidt, C. Bartsch, J. Bormann, D. Stoffel, and W. Kunz, "An equivalence checker for hardware-dependent software," in *MEMOCODE*, 2013, pp. 119–128.

[17] 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 *FMCAD*, Oct 2013, pp. 121–128.

[18] A. Traber, F. Zaruba, S. Stucki, A. Pullini, G. Haugou, E. Flamand, F. K. Gurkaynak, and L. Benini, "PULPino: a small single-core RISC-V SoC," in *RISCV Workshop*, 2016.

[19] B. Jeannet and A. Miné, "Apron: A library of numerical abstract domains for static analysis," in *CAV*, 2009.

[20] S. Stattelmann, O. Bringmann, and W. Rosenstiel, "Dominator homomorphism based code matching for source-level simulation of embedded software," in *DAC*, 2011.

[21] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The mälardalen WCET benchmarks: Past, present and future," in *WCET*, 2010.

[22] J. Rudolf, M. Strobel, J.-J. Benz, C. Haubelt, M. Radetzki, and O. Bringmann, "Automated sensor firmware development - generation, optimization, and analysis," in *MBMV*, 2019.

[23] IEEE, *IEEE Standard SystemC Language Reference Manual*, IEEE Std. 1666, 2011.

[24] D. Große and R. Drechsler, *Quality-Driven SystemC Design*. Springer, 2010.

[25] V. Herdt, D. Große, H. M. Le, and R. Drechsler, "Extensible and configurable RISC-V based virtual prototype," in *FDL*, 2018, pp. 5–16.

[26] V. Herdt, D. Große, H. M. Le, and R. Drechsler, "Verifying instruction set simulators using coverage-guided fuzzing," in *DATE*, 2019, pp. 360–365.

[27] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *CAV*, 2007, pp. 519–531.

[28] V. Herdt, H. M. Le, D. Große, and R. Drechsler, "Compiled symbolic simulation for SystemC," in *ICCAD*, 2016, pp. 52:1–52:8.

[29] V. Herdt, H. M. Le, D. Große, and R. Drechsler, "Verifying SystemC using intermediate verification language and stateful symbolic simulation," *TCAD*, vol. 38, no. 7, pp. 1359–1372, July 2019.

[30] M. Schwarz, R. Stahl, D. Mueller-Gritschneder, U. Schlichtmann, D. Stoffel, and W. Kunz, "Access: HW/SW Co-Equivalence Checking for Firmware Optimization," in *DAC*, 2019.

[31] B. Schmidt, C. Villarraga, T. Fehmel, J. Bormann, M. Wedler, M. Nguyen, D. Stoffel, and W. Kunz, "A new formal verification approach for hardware-dependent embedded system software," *T-LSDM (ASP-DAC 2013 Special Issue)*, vol. 6, pp. 135–145, 2013.