Formal Co-Validation of Low-Level Hardware/Software Interfaces

Alex Horn*, Michael Tautschnig*, Celina Val†, Lihao Liang*,
Tom Melham*, Jim Grundy‡, Daniel Kroening*
*University of Oxford †University of British Columbia ‡Intel Corporation

***Abstract*—Today's microelectronics industry is increasingly confronted with the challenge of developing and validating software that closely interacts with hardware. These interactions make it difficult to design and validate the hardware and software separately; instead, a verifiable co-design is required that takes them into account. This paper demonstrates a new approach to co-validation of hardware/software interfaces by formal, symbolic co-execution of an executable hardware model combined with the software that interacts with it. We illustrate and evaluate our technique on three realistic benchmarks in which software I/O is subject to hardware-specific protocol rules: a real-time clock, a temperature sensor on an I^2C bus, and an Ethernet MAC. We provide experimental results that show our approach is both feasible as a bug-finding technique and scales to handle a significant degree of concurrency in the combined hardware/software model.**

I. Introduction

A growing problem for today's microelectronics industry is co-design of hardware alongside embedded, low-level software that closely interacts with it. In particular, semiconductor designs are witnessing an increased use of on-chip microcontrollers running firmware to implement functionality that would formerly have been implemented in hardware. This trend is driven by factors that include the following:

- Extracting the control of complex devices and implementing it in firmware can cut development schedules while adding flexibility and survivability.
- By making on-chip devices more capable, work can be shifted away from the CPU, where performance is increasingly hard-won. The richer control required for more capable devices further drives the trend.

The sorts of devices that are typically integrated on-chip are controllers for power management, hardware with sequestered functionality for remote management or secure content, and increasingly capable graphics processors.

Co-design and validation of such devices together with their firmware, with the predictability needed to schedule fabrication and hit market windows, has become an acute challenge. Similar challenges arise in developing firmware for systems on chip (SoCs) and general embedded systems [1].

Firmware is just hardware-specific software. One might therefore expect that the problem can be addressed by some combination of today's separate techniques for design and verification of hardware and software. But the results of this approach are disappointing. The problem is the complex nature of the interactions at the hardware/software interfaces. All large systems are structured into subsystems, but the interfaces between hardware and software subsystems are more problematic than those in a homogeneous system.

- In a homogeneous design, the documentation of interfaces (say by header files) is both understandable by developers and processable by tools. A compiler, for example, can guarantee some consistency in how two modules view a shared interface. But hardware and software are typically described in different languages, processed by separate tool chains. And the hardware and software design teams have their own descriptions of the interfaces they share, with little to ensure the two are consistent.
- The mechanisms for invoking functionality and sharing data among modules in a homogeneous system, particularly in software, are relatively few. In contrast, the means of passing information between hardware and software are varied and built up from nonstandard primitives that may include interrupts, memory-mapped I/O, and special-purpose registers. The situation is analogous to software before procedure-calling conventions were standardized.
- The hardware/software interface also marks a boundary between different threads of concurrent execution. Without the shared understanding of synchronization that follows from a common language and library, concurrency at the hardware/software interface needs special treatment.
- Finally, a hardware/software interface almost always marks a boundary between different teams, working in different parts of a company and having different educational backgrounds and skills. The scope for misunderstanding is greater than usual.

The challenges faced by those implementing the two sides of a hardware/software interface are high—but so is the need to get it right. Building a system with a new interface and then testing it to find and remove bugs is a perilous practice. When designers move the control from a complex hardware device into firmware, the stripped-down hardware can be difficult to test without a means to run the firmware. Testing the hardware and firmware together is difficult before silicon is fabricated: simulators are slow, emulators expensive, and FPGAs limited in capacity. Delaying extensive testing until silicon is available is unacceptable as it serializes the development of hardware and software to the point where it can be difficult to meet the

Supported by ERC project 280053 and EPSRC project EP/H017585/1.

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 2013: Formal Methods in Computer-Aided Design: Portland, Oregon, USA, 20-23 October 2013*, edited by B. Jobstmann and S. Ray (IEEE Computer Society, 2013), pp. 121–128. The final publication is available at https://doi.org/10.1109/FMCAD.2013.6679400

project schedule. And, of course, any hardware bugs found at this stage will be expensive to fix.

An increasingly common approach to designing embedded software is to employ *virtual prototyping*, so that coding can begin before silicon is available. A software model (in C or SystemC) serves as a proxy for the hardware on which the embedded firmware code can be developed and tested. In our research, we leverage this trend and propose a new approach to co-validation of hardware/software interfaces by *formal, symbolic co-execution* of two combined pieces of code: a software model of the hardware, and the real embedded firmware that interacts with this hardware. When necessary, we capture the parallelism between firmware and hardware with a modelling approach that employs software concurrency in the form of asynchronous threads.

We demonstrate our idea with three realistic benchmarks, all publicly available on the web.[1] These are constructed by combining hardware models adapted from a virtual machine emulator and software taken from a general-purpose OS kernel. The interactions at the hardware/software interfaces of these benchmarks are characteristic of devices of interest to our industry partner—including low-level software, akin to a driver, executing on a separate on-chip microcontroller. Our experimental results show the approach both is feasible as a bug-finding technique and scales to handle a significant degree of hardware/software concurrency.

A further, methodological, contribution of our work is the identification of the open-source QEMU [2] code base, together with Linux device drivers [3], as a rich source of characteristic examples to drive research in this domain. The correctness properties we have devised also exemplify validation problems that are typical of low-level hardware/software interfaces. Co-validation poses distinctive challenges and has growing industrial importance; it is our hope that this work will encourage other formal verification researchers to engage with this important problem in contemporary hardware design.

II. Validation Aims and Technical Approach

We have designed and experimentally evaluated a method for semi-automatically searching for bugs in the interactions at hardware/software interfaces. Our aim is to find potential violations of certain correctness properties as the hardware interacts with the low-level software under scrutiny. Such violations yield counterexample traces that can expose bugs both in the low-level software and in the hardware model. Our method targets early and relatively high-level *software* models of hardware, which need not be cycle accurate.

For our work, the purpose of hardware models is to capture the hardware side of interactions that occur at nonstandard hardware/software interfaces. This includes modelling complex, ah-hoc side-effects characteristic of low-level devices. For example, when the software reads a hardware register, this may cause other changes to the visible state at the interface.

[1] http://www.cprover.org/firmware

In general, these interactions are expected to conform to a protocol that can be articulated in terms of pre- and post-conditions, which we refer to as *properties*. We formalize these as runtime assertions within the executable hardware model itself. Specifications of interface protocols are usually articulated separately, in logic. Our approach, however, is designed to appeal to practicing engineers who use virtual platforms to test their intuition through co-simulation, and so we embed specifications within executable models—which are given formal meaning through a bit-precise execution semantics [4].

A. Modelling Approach and Concurrency

In the conventional approach to formal hardware verification, hardware models are essentially state-transition systems: a formal model is given that determines or constrains the next state of registers/memory in relation to the current state and inputs. In this work, we propose a different approach that need not be cycle accurate and provides a higher-level, event-driven software abstraction of the hardware, focussed on interactions at its interface with low-level software.

In some hardware/software systems, the speed of the hardware and low latency of the interaction mechanism, relative to software execution, imply that we do not need to model hardware/software concurrency. An example is the RTC benchmark in Section IV. In these cases, our combined code, comprising the firmware plus the hardware model, can simply represent interaction by procedure calls into the hardware model from the software. This simplification encodes an assumption that hardware response is effectively instantaneous.

More interesting is our handling of hardware/software concurrency in the cases where it matters. Here, we model the hardware by asynchronous software threads invoked by the low-level code that interacts with the hardware. When this code engages in an interaction with the hardware—writing to a register, say—a concurrent, asynchronous thread is spawned whose sole purpose is to call a function in the hardware model that initiates an execution to model the hardware's response. Once the function returns, the thread terminates. This event-driven abstraction enables us to find concurrency bugs, as illustrated by the Ethernet MAC benchmark in Section VI. It also maps well onto the early-stage hardware modelling activity that our validation method targets.

In the specific context of our benchmarks drawn from QEMU, we are able justify making threads of the hardware model atomic with respect to each other and the low-level software. This reduces the complexity of our analysis. We recognize, however, this will not always be possible.

The low-level software's response to hardware *interrupts* are also modelled by asynchronous threads, this time created at run-time by the hardware model—and not atomic. This allows us to capture concurrency bugs in interrupt handling, a prominent potential source of errors in the industrial systems we have in mind.

B. Validation Technology and Concurrency Encoding

Our validation method is designed to leverage today's highly optimized SMT/SAT solvers. Where our combined code is purely sequential, we can analyse it with any software analysis tool that gives a bit-precise semantics to C and can check our embedded run-time assertions. In Section VII, we report experimental results using CBMC and, for comparison, also using KLEE—a pathwise testing tool that achieves high coverage through symbolic execution [5].

Our main approach, however, is to analyse our coalesced code by symbolic execution with aggressive path-merging, as exemplified by CBMC [6]. This yields a mathematical formula that encodes multiple execution paths up to a certain depth, which is then checked by a SAT/SMT solver. This approach maximizes the exploitation of today's optimised solvers and avoids the potentially exponential number of execution paths explored by path-wise enumeration [7], [5].

To handle concurrency, we exploit a recent encoding of concurrent software execution in CBMC that uses *partial orders* to constrain the relative timing of events that access shared state [8]. Roughly speaking, it works as follows. Accesses to shared state by separate concurrent threads are first decoupled by being given distinct symbolic references. An integer *clock* is introduced for each access to shared state, and a partial order is given among the clocks that encompasses all feasible interleavings of these accesses to shared state. Finally, order-dependent equality relationships are established among the values named by the decoupled references, making connections between the state values 'seen' by the hardware model and low-level software. All this is efficiently encoded in a quantifier-free formula whose size is cubic in the maximum number of shared state accesses. Any satisfying assignment found by a SAT/SMT solver corresponds to a property violation.

Encoding concurrency by partial orders side-steps having to deal explicitly with the complexity of interleavings, and produces a highly competitive analysis for concurrent software [8], which we exploit in this work. It also allows the ordering of accesses to be more loosely specified than in conventional sequential consistency. The latter property is used in [8] to capture the complex semantics of weak memory models in modern multi-core architectures, but is not (yet) exploited in our work on hardware/software co-validation.

III. Hardware Models from QEMU Devices

To evaluate our method, we extract hardware models from the open-source QEMU virtual machine emulator [2] and combine them with Linux device drivers [3]. Our idea is to leverage the rich collection of hardware models that QEMU provides, in combination with real OS driver code. This strengthens the objectivity and realism of our experiments, since they retain essential characteristics of production code. This includes a specific division into hardware models and low-level software, which moreover originate from separate developer communities.

QEMU was designed for hardware virtualization, not experiments in formal co-validation, and extracting usable stand-alone hardware models from QEMU code is not entirely straightforward. To give an idea of what is involved, we briefly sketch some aspects of the QEMU architecture, before going on to present our benchmark experiments.

QEMU is written in C. Each QEMU virtual machine is divided into *boards*, each of which consists of *device* and *bus* models. Communication between device models can occur only through bus models. This is the basis for a modular design, implemented through a QEMU-specific factory and service locator pattern known as QDev [9].

QDev organizes hardware models into a dynamic tree data structure that relies on a QEMU-specific object model called QOM [10]. In essence, QOM seeks to extend C with object-oriented programming features. To achieve this, QOM stores information about its internal C structures in `glib` trees and hash tables. An instantiation of such a structure is called an *object*. Function pointers serve as methods. QEMU's physical memory management architecture determines which object methods of a device model are called when memory regions are accessed by the guest operating system [11].

For our benchmarks, we extracted stand-alone C hardware models by excluding all physical memory management code and dependencies on QDev and QOM. This was done through a somewhat laborious process of careful slicing and approximation of essential features.

By default, QEMU accelerates dynamic code translation though just-in-time compilation [12]. For our purposes, this can fortunately be bypassed through an undocumented feature called QTest, a client-server architecture that facilitates testing of hardware models. Few QEMU models currently take advantage of this test harness, but the trend is towards more testing. The benefit for our approach is that these tests can give insight into hardware-specific verification properties, and serve as starting points for symbolic co-execution.

IV. Co-validation of a Real-Time Clock

Our first benchmark is the MC146818 real-time clock (RTC), a low-power CMOS device that provides—among other functionality—a time-of-day clock, a calendar, and programmable timers for periodic interrupts and square-wave generation [13]. One of the stated purposes of the MC146818, which is quite an old device, is to 'relieve the [micro-processor] software of the timekeeping workload'. This motivation also drives today's proliferation of complex on-chip device controllers, themselves running firmware.

The RTC is representative of hardware devices that firmware interacts with through special-purpose registers, a common low-level software/hardware interface idiom [14]. The speed of the RTC device means we can represent interaction with it by procedure calls in the firmware; in essence we can assume, in this benchmark, that both a register write and the hardware's response to it are instantaneous.

In this first benchmark, we focus on only part of the RTC interface: reading and writing the registers that hold the time, date, and alarm data. As will be seen, this is not simply a matter of the firmware executing a 'read' or 'write' instruction, but requires some ancillary manipulation of bits in control

status registers. Our validation task is to check for violation of the protocols that govern this mechanism.

A. Interface Properties

The MC146818 presents its interface as 64 bytes of RAM, addressed `0x00` though `0x3F`. The time, date, and alarm data are held in the first 10 bytes, each of which is essentially a 'register' at a specific address [13]. For example, the byte at `0x09` is the register that holds the year. To access a register, software must execute a sequence of two I/O instructions that access two different memory-mapped hardware registers at addresses `0x70` and `0x71`. The first determines the data register to be accessed, and the second holds the value of this register. It is an error to read or write a data register value at address `0x71` without first setting the register to be accessed:

- ⋆ *Each execution of* `outb 0x71` *or* `inb 0x71` *must be preceded by a unique* `outb 0x70`.

This property alone is insufficient to guarantee safe writes of data. The firmware controlling this device must also correctly manipulate two control bits in 'Register B', one of four other RAM locations whose individual bits monitor and control a diverse assortment of device operations. The two relevant bits are the *SET* bit and the *data mode (DM)* bit.

- ⋆ *The SET bit of Register B must be enabled (have value* 1*) when any of the time, date, or alarm registers are written.*

Once SET is enabled, data can be safely written as either binary or binary-coded decimal. The choice must be made explicit by writing 1 or 0 to the DM bit. In addition, the selected 'data mode cannot be changed without re-initializing the 10 data bytes' [13]. A permissive interpretation of this sentence in the data sheet yields the following two properties:

- ⋆ *The DM bit can be changed only when the SET bit is already enabled, or as the SET bit is also being enabled or disabled when Register B is written.*
- ⋆ *If the DM bit has changed since the SET bit has been or is being enabled, then every time, calendar and alarm register must have been written at the moment when the SET bit is disabled.*

The final property we discuss here says there are no concurrent hardware writes to any of the time, date, or alarm registers while the SET bit is 1. Note that no such guarantee exists once the SET bit is disabled.

- ⋆ *While the SET bit is 1, when data d is written to a time, calendar or alarm register R, a subsequent read of R returns d.*

We have shown only informal statements of our properties to make the exposition accessible. In our actual method, we encode properties as runtime assertions in the RTC hardware model. Several other properties, omitted here for brevity, are included in our experiments. This yields an executable specification, through which we expose a real bug (Section VII).

B. Technical Details of the RTC Benchmark

To illustrate the architecture shared by all our benchmarks and explain how hardware models are extracted from QEMU, we give here some technical details for the RTC benchmark.

The full RTC model in QEMU depends on a large amount of code irrelevant to our properties, and is too complex to analyse formally. We therefore manually removed these dependencies, including the dependency on the i440fx PCI host. We also manually sliced away some code not representing actual hardware, such as QEMU timers. These simplifications, which we would expect in future to mechanize, preserve the core of the RTC model.

There are two loops in the RTC hardware model that are hard for CMBC to handle, but the functions that contain them are for RTC timer functionality that has nothing to do with our interface properties. We could therefore safely remove these function calls without affecting the validation results.

For the low-level software side, we used the dependency tracking capabilities of CBMC to pull together sufficient Linux driver code to exercise the hardware model. This was done in a semi-automated way that produces a coalesced C program containing the model together with a superset of the exact Linux code needed to drive the hardware features covered. The coalesced program has around 49k lines of C code.

Symbolic execution of the coalesced program has to proceed from a main function that actually invokes the driver. For this, we develop scenarios that invoke the driver in various ways. These were specially written for the RTC benchmark, but a merit of our approach in general is that we might instead leverage test cases created by developers, as long as these initialize the hardware model. Our code initializes the RTC with a non-deterministic value representing the time. After initialization, we call the Linux device driver function `get_rtc_time()` to read the time from the RTC. Finally, we call `set_rtc_time()` to write back the time just read. These calls induce state transitions in the hardware model.

A similar approach can be taken to produce benchmarks for stand-alone QEMU hardware models, in isolation from their driver, simply by wrapping each QEMU model with some C code that enacts driver I/O scenarios. In the RTC benchmark, each such scenario includes an initialization step that sets the time in the RTC to a non-deterministic value, represented in binary-coded decimal and constrained to be in the range given in the MC146818 datasheet [13]. For a sanity check of the properties, we created a buggy test case for the stand-alone hardware model that calls `inb(0x71)` before `outb(0x70,*)`, where $*$ is a non-deterministic value.

V. Co-validation of an I²C Temperature Sensor

The second benchmark features a temperature sensor [15], called 'TMP105', that is controlled by software through the I²C bus [16]. This allows us to experiment with properties that go beyond fixed-sized register updates. Our hardware model for the I²C benchmark incorporates the essential interaction constraints for these updates, so the benchmark does not need to include a model of the I²C bus controller.

The temperature sensor has four registers: an 8-bit configuration register, a 16-bit temperature register, and 16-bit lower and upper temperature threshold registers for hysteresis. Reading and, when applicable, writing of these registers is done over the I^2C serial bus. The registers have different sizes, so the number of transmitted bytes varies.

Individual bits in registers must conform to rules similar to those of the RTC. We show a few illustrative properties, again stated informally here but in practice encoded as run-time assertions in the hardware model.

The sensor can be shut down by writing a 1 to the least significant bit of the configuration register. This turns off continuous temperature measurements to save power. While the sensor is in this 'shutdown mode' individual readings can still be triggered by writing a 1 to the most significant bit of the configuration register. This leads to the following property:

⋆ *Each read of the temperature register is preceded by a write of a 1 to the most significant bit of the configuration register if and only if the temperature sensor is in shutdown mode.*

Writing 1 to the most significant bit of the configuration register merely triggers an individual temperature measurement; the bit itself is immutable and not affected by the write.

⋆ *When the most significant bit of the configuration register is read, it is zero regardless of any previous writes to it.*

The next property concerns the configuration register.

⋆ *After writing byte c to the configuration register, the next read gives a byte c' where $c'[i] = c[i]$ for $0 \leq i < 7$.*

That is, all bits of the old and new configuration value are pairwise equal, except perhaps the most significant bit.

Altogether, the bus and register properties amount to around two dozen runtime assertions. It was straightforward to encode these in the TMP105 hardware model extracted from QEMU: the TMP105 internal state is stored in a C structure that has fields to that implement its registers and store control information related to communication through the I^2C protocol.

VI. Co-validation of an Ethernet MAC

Our final benchmark concerns interrupt-driven software for an Ethernet MAC with a direct-memory access (DMA) ring [17]. We concentrate on the functionality of receiving Ethernet frames. Each incoming frame is called an *RX frame*. The Ethernet MAC can be configured to generate a hardware interrupt for each RX frame. We call this 'interrupt mode'. When interrupts are disabled but RX frames should still be processed, the software polls for incoming data.

Hardware/software concurrency is therefore important to model in the Ethernet MAC benchmark, because multiple frames can arrive simultaneously and the software reacts to interrupts generated by the hardware. These are handled using the modelling approach discussed in Section II, and produce a significant degree of concurrency in the coalesced model.

A noteworthy complication is that the software switches between interrupts and polling to improve performance [18]. Similar techniques are used for block devices with high data throughput, such as solid state drives. Switching between polling and interrupt mode is known to be error-prone, so this benchmark is a good exemplar for concurrency bugs due to interrupts in a producer-consumer scenario. This section explains one such bug and how the developers fixed it.

The OpenCores Ethernet MAC features 128 DMA buffer descriptors [17], each of which determines the memory that holds an Ethernet frame. Our benchmark code elides the details of DMA address translation; instead, we focus on how the software and hardware synchronize their updates to the DMA buffer. In the case of RX frames, the software sets bit 15 in a buffer descriptor to 1 when the associated DMA buffer can be overwritten by the hardware. Such a buffer descriptor is said to be 'empty'. The hardware clears bit 15 to signal to the software that the DMA buffer associated with a buffer descriptor contains a new RX frame. Despite its simplicity, this communication protocol is error-prone when interrupts are being re-enabled, as illustrated next.

Suppose there is at least one empty RX buffer descriptor. The software switches from polling to interrupt mode as soon as it detects no new RX frames. To do this, it reads bit 15 of the next available RX buffer descriptor. Suppose the current buffer descriptor is empty and so this bit is still 1. In this case, the version of the software with the bug continues by clearing all RX interrupt sources before re-enabling all RX interrupts.

Unfortunately, this algorithm can result in RX frames being delayed or even dropped. Figure 1 shows an example, in which an RX frame arrives just after the check for new RX frames but before the RX interrupt sources are cleared. This RX frame will not trigger an interrupt until another one arrives. In fact, if there are no other ones, the delayed RX frame is not even promoted to the socket layer. This happens when the driver is stopped, for example due to standby.

The following properties will expose this concurrency bug:

⋆ *When the software enables the MAC receiver, there exists at least one empty RX buffer descriptor.*

⋆ *The software must eventually process every RX frame. At the very latest, when it is stopped, all RX frames must have been processed.*

The crux of these properties is that the driver must detect any potentially lost frames. Figure 2 shows how the developer for the 'ethoc' driver in the Linux 2.6.38 kernel release fixed the bug, ensuring these properties are then satisfied.

These and several other properties were analysed using CBMC and the partial order encoding for concurrency. The scenarios we wrote to exercise these properties asynchronously invoke the hardware model to trigger new RX frames or force the MAC to become busy.

A few simplifications were made to enable analysis within reasonable time and memory bounds. Because the solver has no array logic built in, we had to reduce the maximum number of DMA buffers to eight and shrink their sizes to at most two bytes. For the same reason, the number of buffer descriptors in the hardware model was reduced to eight. Finally, we

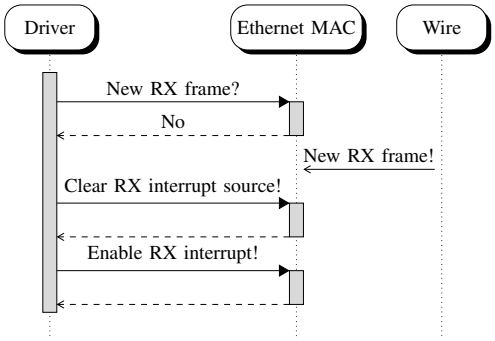

Fig. 1. Incorrect handling of an empty RX buffer descriptor causes potential package loss.

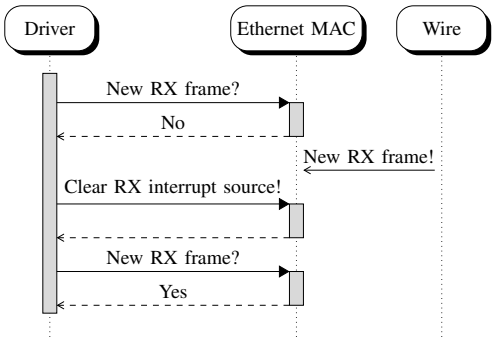

Fig. 2. A second buffer descriptor check, after the RX interrupt sources have been cleared, detects intermittent RX frame arrivals.

suppressed interrupts on changes of the interrupt mask because this functionality appears to be QEMU-specific and not part of the Ethernet MAC itself.

VII. Experimental Results

Table I summarizes our experimental results. We performed all experiments on a 64-bit machine running Linux 3.5.0 with eight Intel Xeon 3.07 GHz cores and 48 GB of main memory.

For the RTC and I^{2}C benchmarks, we employed sequential, multi-path symbolic execution. Loops were unrolled a bounded number of times. This type of symbolic execution generates a Boolean formula that encodes the expected interface properties and all calls to read and write procedures in the hardware model invoked from the low-level software. The formula is then checked with MiniSat 2.2. If the formula is satisfiable, a violation of some property has been found. Otherwise, no decisive conclusion about the validity of the property can be reached.

As part of our RTC experiments, we found a real bug in the QEMU hardware model[2] that causes it to violate property RTC.1 in Table I. The violation is exposed through a test that first writes a time or calendar register and then writes to one of the control registers of the device. For the combined RTC benchmark with the hardware model and driver code, CBMC reports the violation of property RTC.1 in $225.3\,s$, of which

	LOC	#Unroll	#Threads	#Constraints	#Clauses	Sec.
RTC.1	47609	21	1	61314	25,797,536	122.5
RTC.1 (unfixed)	47609	21	1	61065	25,771,226	225.3
RTC.2	47609	21	1	61314	26,430,338	71.7
RTC.3	47609	21	1	60648	25,769,903	68.5
RTC.4	47609	21	1	60766	26,422,852	69.7
RTC.5	47609	21	1	60435	26,425,148	69.5
RTC.6	47609	21	1	60295	8,208,764	54.1
RTC.7	47609	21	1	60394	8,759,757	55.2
RTC.8	47609	21	1	60294	24,491,011	69.1
RTC.9	47609	21	1	60294	24,468,704	67.7
RTC.10	47609	21	1	60294	24,781,142	68.6
RTC.11	47609	21	1	60668	25,231,840	112.8
I2C.1	46609	16	1	159481	20,020,885	803.9
I2C.2	46609	16	1	158391	19,997,082	793.1
I2C.3	46609	16	1	158556	20,006,113	795.8
I2C.4	46609	16	1	158556	20,023,452	787.1
I2C.5	46609	16	1	158556	20,024,494	786.2
I2C.6	46609	16	1	158436	19,998,982	783.4
I2C.7	46609	16	1	158436	20,007,984	786.1
I2C.8	46609	16	1	158436	20,001,601	780.5
I2C.9	46609	16	1	163866	20,118,277	854.4
I2C.10	46609	16	1	164547	20,074,751	841.0
I2C.11	46609	16	1	162381	20,388,706	808.6
I2C.12	46609	16	1	158556	20,160,928	789.4
I2C.13	46609	16	1	158811	20,009,910	804.3
I2C.14	46609	16	1	160596	20,294,436	798.7
I2C.15	46609	16	1	160596	20,295,406	800.8
I2C.16	46609	16	1	158391	19,997,740	788.9
I2C.17	46609	16	1	167481	20,064,678	912.9
ETHOC.1+2	940	2	2	1036+57	336,557	1.7
ETHOC.1+3	940	2	13	4633+1063	8,109,581	46.1
ETHOC.1+4	940	2	17	6073+1300	17,192,339	145.7
ETHOC.5	2097	1	19	16707+20991	250,371,908	1680.7
ETHOC.6	2097	1	19	16683+21034	252,154,414	335.5
ETHOC.7	2097	1	19	16635+20750	239,259,859	219.5
ETHOC.5-seq	2097	1	1	17710	73,388,552	426.8
ETHOC.6-seq	2097	1	1	17686	73,324,230	426.1
ETHOC.7-seq	2097	1	1	17648	71,998,722	435.3

TABLE I
Experimental Results

$177.4\,s$ were spent in MiniSat. The violation of property RTC.1 in the standalone RTC hardware model is found in $50\,s$.

The temperature sensor benchmark also helped to expose a real bug in the QEMU hardware model.[3] The bug causes data on the I^{2}C bus to be lost because of an off-by-one error. In the standalone I^{2}C hardware model, CBMC found a violation of property I2C.10 in $3.6\,s$ and property I2C.18 in $3.4\,s$, of which $1.4\,s$ and $1.3\,s$, respectively, were spent in MiniSat.

We also tried to use CBMC to expose property violations in the temperature sensor by analysing the combined driver code and hardware model. The scenarios developed for this analysis take a brute-force approach, in which we explore all possible sequences, of a fixed length, of non-deterministically chosen driver API function calls. The sequences are encoded symbolically, in a single run of the analysis. The idea was to simulate all possible fixed-length sequences of invocations of the driver API, such as might arise in a typical interrupt-driven setting. Our experiments with a bound of 15 driver API calls in the test harness failed to expose a property violation. CBMC timed out after $1800\,s$ when the number of API function calls was set to 20.

For the ETHOC device benchmark, our tool processes the coalesced code and symbolically encodes concurrent memory accesses as partial orderings, as discussed in Section II. The resulting formula, sent to an SMT/SAT solver, captures all possible execution schedules for the threads that enact invocations of hardware functionality. Interrupts generated

[2]http://git.qemu.org/?p=qemu.git;a=commit;h=02c6ccc6dde90dcbf5975b1c

[3]http://git.qemu.org/?p=qemu.git;a=commit;h=cb5ef3fa1871522a08866270

by the hardware model also introduce concurrency that is included in the encoding, because the software's interrupt handler executes asynchronously from the hardware's point of view.

This modelling approach generates a significant degree of concurrency. Our scenarios result in up to 18 threads being spawned, yet we can validate most of the concurrent scenarios, explained in further detail below, in under 6 minutes (one scenario requires up to 29 minutes). The partial-order based encoding of [8] thus appears to be well suited for efficiently checking our models of combined hardware/software systems.

First, we validated the QEMU hardware model in isolation. In under 1 minute, despite 12 threads being spawned, we showed that our hardware model correctly simulates interrupts to be raised asynchronously. As our symbolic execution is limited to bug-finding, we had added a runtime assertion to state the converse; the counterexample we found constitutes evidence that our interrupt properties are not vacuously true.

With a more than tenfold increase in the number of clauses when analysing the combined hardware/software system, the burden on the underlying decision procedure rises significantly. We therefore experimented with both a purely sequential composition of the systems, corresponding to an assumption of instantaneous hardware operation, and a concurrent version, capturing all behaviour. In just over 7 minutes, our experiments confirm that the sequential version does not exhibit the erroneous behaviour described in Section VI. This failure to detect the bug illustrates how developing software without a realistic hardware model poses risks. The complete, concurrent system spawns 18 threads to simulate all interactions of the software with asynchronous hardware and interrupts. In 29 minutes we were now able to detect a property violation that exposes the presence of the bug in the combined hardware/software system.

Other Experiments using KLEE. We have also analysed all three benchmarks using KLEE [5]. On the RTC benchmark with the combined hardware model and driver software, KLEE times out after $1800\,s$. It also times out on the standalone, corrected RTC hardware model, but it can find the bug in the original model in $1\,s$. The temperature sensor driver code cannot be compiled into LLVM IR, but KLEE can check the corrected hardware model in $1\,s$. When we analyse the original temperature sensor hardware model, KLEE exposes the bug in $1\,s$. In addition, we confirmed that KLEE cannot detect the concurrency bug in the Ethernet MAC driver; with both the corrected and buggy version of the driver, it explores 25 paths in less than $30\,s$ and passes all seven ETHOC properties (whether violated or not).

VIII. Discussion and Future Work

In this section, we indicate some of the limits of our work and discuss some directions for further research.

The hardware models in all our benchmarks serve as an executable specification of the hardware/software interface. This is helpful to engineers, who test their intuition through simulation. An executable hardware model also gives flexibility in expressing properties. But, under symbolic analysis, this can also give rise to a diverse range of logical formulas for checking—making it hard to find an appropriate decision procedure. The problem might be mitigated by adopting an embedded contract language [19].

We found executable hardware models to be essential in discovering key properties of the hardware and software, and to exploring the interfaces between them. Our models are event-driven and phrased at a high level—executable software abstractions that strike a balance between modelling accuracy and tractability. An obvious goal is to verify them formally against a lower-level reference model. Our Ethernet MAC benchmark could be used as a research case study for this, in which executions of threads in the stand-alone QEMU model are compared with the OpenCores RTL Verilog model [17].

Our Ethernet MAC benchmark exemplifies the subtleties of interrupts by explaining a concurrency bug in a driver. For the automatic analysis, we relied on symbolic encoding of interactions between concurrently running model elements. But we have not yet considered nested interrupts, which would require extensions to both our models and the concurrency encoding. In future, we would also like to address interrupt priorities, such as FIQ interrupts on ARM. Our experiments also expose opportunities for automatic slicing algorithms that are aware of concurrency semantics.

Multi-threading in the Ethernet MAC benchmark frames some of our current research on improvements to the partial order concurrency encoding. Our experience suggests there is potential to delegate some of the generation of constraints that constitute the partial order on state accesses to the SMT solver itself, where it could be done incrementally.

IX. Related Work

Most research on verification of low-level software has focused on operating systems and drivers, with some prominent successes [20], [5], [21], [6], [7], [22]. There has also been some work on formal analysis of assembly code [23] and even binary drivers have been analysed [24]. There is, of course, a large body of literature on design and verification of embedded systems at a higher level [25]. In our work, we address *formal co-validation* of complex interactions between low-level software and on-chip hardware, using a novel technique that combines symbolic execution and partial-order encodings [8], bypassing the scalability limitations of partial-order reductions in earlier work (e.g. [26]). Our work represents the first demonstration of this approach to this important problem in contemporary systems design.

There has been some research, with aims similar to ours, using bounded model checking [27] and interval property checking [28]. The emphasis of both these efforts is on machine instructions and cycle-accurate hardware models, while ours aims at early validation before a cycle-accurate model is available. This is reflected in the fact that the work of [28] targets Verilog code, while ours revolves around higher-level models in C. Earlier work also analysed a more non-deterministic C model through abstract interpretation [29],

but with less sophisticated support for concurrency. More recently, an automata-theoretic co-verification technique has been applied to PCI drivers [30].

Other related work has used symbolic simulation and SMT to check equivalence between a software reference model and a system containing (restricted) C code that invokes data computations on reconfigurable streaming hardware modelled in Java [31]. Hardware/software concurrency is not represented; interaction is modelled by synchronous calls from the software into an API that loads and runs the streaming hardware designs. The aim is to establish correctness of the dataflow computations in hardware/software co-designs. By contrast, our work and modelling approach seek to uncover bugs in hardware/software systems that interact through concurrent, imperative modification of shared state.

X. Concluding Remarks

We have described a new approach to co-validation of hardware and low-level software, based on formal co-verification of an executable hardware model together with the software that interacts with it. We articulate key correctness properties that we expect interactions at the hardware/software interface to exhibit, and check these by symbolic execution. As our experiments show, the approach can be adapted to a range of interaction mechanisms—and it can expose bugs.

Systematic experimental research into formal co-verification of hardware and low-level software is hindered by the lack of realistic benchmarks accessible to academic researchers. Our work suggests a way to close this gap. We exploit the availability of well-developed open source virtual machine emulator code, from which one can extract a wide range of typical hardware models. These models can be made to work with Linux drivers, which serve as a proxy for typical firmware code. A practical benefit is that this facilitates collaboration with the systems community, whose insights helped us understand the problem space and expose real bugs. We suggest that a community effort to develop a benchmark suite, following our approach, would produce an invaluable resource to drive further academic research into firmware validation.

XI. Acknowledgements

This work is funded by a donation from Intel Corporation for research on *Effective Validation of Firmware*. We are grateful for illuminating discussions with our project partners: Alan Hu (UBC), Luke Ong (Oxford), Moshe Vardi (Rice), and Sharad Malik (Princeton). We thank Anthony Liguori (IBM), Paolo Bonzini (Redhat), and Andreas Färber (SUSE) for their comments on the QEMU mailing list.

References

[1] J. Teich, "Hardware/software codesign: The past, the present, and predicting the future," *Proceedings of the IEEE*, vol. 100, pp. 1411–1430, May 2012.

[2] A. Liguori, "QEMU emulator user documentation," http://wiki.qemu.org/download/qemu-doc.html, Jan. 2010.

[3] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers*. O'Reilly, 2005.

[4] G. Parthasarathy, M. K. Iyer, K.-T. Cheng, and L.-C. Wang, "An efficient finite-domain constraint solver for circuits," in *Design Automation Conference (DAC)*. ACM, 2004, pp. 212–217.

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

[6] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *TACAS*. Springer, 2004, vol. 2988, pp. 168–176.

[7] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," *SIGPLAN Notices*, vol. 40, pp. 213–223, June 2005.

[8] J. Alglave, D. Kroening, and M. Tautschnig, "Partial orders for efficient Bounded Model Checking of concurrent software," in *Computer-Aided Verification (CAV)*, ser. LNCS, vol. 8044. Springer, 2013, pp. 141–157.

[9] P. Bonzini, "QEMU developer mailing list – qdev for programmers," http://lists.nongnu.org/archive/html/qemu-devel/2011-07/msg00842.html, Jul. 2011.

[10] A. Liguori, "QEMU wiki – QOM," http://wiki.qemu.org/Features/QOM, Jul. 2011.

[11] A. Kivity, "QEMU developer documentation on memory API," http://git.qemu.org/?p=qemu.git;a=blob;f=docs/memory.txt, Jul. 2011.

[12] F. Bellard, "QEMU, a fast and portable dynamic translator," in *ATEC*. USENIX Assoc., 2005, pp. 41–46.

[13] Freescale Semiconductor, *MC146818 – Real-Time Clock Plus RAM (RTC)*, http://www.freescale.com/files/microcontrollers/doc/data_sheet/MC146818.pdf, 1988.

[14] G. Stringham, *Hardware/Firmware Interface Design: Best Practices for Improving Embedded Systems Development*. Elsevier, 2009.

[15] Texas Instruments, *Digital Temperature Sensor with 2-Wire Interface*, http://www.nxp.com/documents/user_manual/UM10204.pdf, Sep. 2011.

[16] NXP Semiconductors, *UM10204 – I^2C – bus specification and user manual*, http://www.nxp.com/documents/user_manual/UM10204.pdf, Oct. 2012.

[17] I. Mohor, "Ethernet MAC 10/100 mbps," http://opencores.org/project,ethmac, Jul. 2011.

[18] J. H. Salim, R. Olsson, and A. Kuznetsov, "Beyond softnet," in *Proceedings of the 5th annual Linux Showcase & Conference*, vol. 5. USENIX Assoc., 2001, pp. 18–26.

[19] M. Fähndrich, M. Barnett, and F. Logozzo, "Embedded contract languages," in *SAC*. ACM, 2010, pp. 2103–2110.

[20] T. Ball, V. Levin, and S. K. Rajamani, "A decade of software model checking with SLAM," *CACM*, vol. 54, no. 7, pp. 68–76, Jul. 2011.

[21] C. Cadar and D. R. Engler, "Execution generated test cases: how to make systems code crash itself," in *SPIN*. Springer, 2005, pp. 2–23.

[22] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: formal verification of an operating-system kernel," *CACM*, vol. 53, no. 6, pp. 107–115, Jun. 2010.

[23] 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, no. 1, pp. 61–91, Feb. 2006.

[24] V. Kuznetsov, V. Chipounov, and G. Candea, "Testing closed-source binary device drivers with DDT," in *USENIXATC*. USENIX Assoc., 2010, pp. 12–12.

[25] M. Loghi, T. Margaria, G. Pravadelli, and B. Steffen, "Dynamic and formal verification of embedded systems: a comparative survey," *Int. J. Parallel Program.*, vol. 33, no. 6, pp. 585–611, Dec. 2005.

[26] R. P. Kurshan, V. Levin, M. Minea, D. Peled, and H. Yenigün, "Combining software and hardware verification techniques," *FMSD*, vol. 21, no. 3, pp. 251–280, Nov. 2002.

[27] D. Große, U. Kühne, and R. Drechsler, "HW/SW co-verification of embedded systems using bounded model checking," in *GLSVLSI*. ACM, 2006, pp. 43–48.

[28] M. D. Nguyen, M. Wedler, D. Stoffel, and W. Kunz, "Formal hardware/software co-verification by interval property checking with abstraction," in *Design Automation Conference (DAC)*. ACM, 2011, pp. 510–515.

[29] D. Monniaux, "Verification of device drivers and intelligent controllers: a case study," in *EMSOFT*. ACM, 2007, pp. 30–36.

[30] J. Li, F. Xie, T. Ball, V. Levin, and C. McGarvey, "An automata-theoretic approach to hardware/software co-verification," in *FASE*. Springer, 2010, pp. 248–262.

[31] T. Todman, P. Boehm, and W. Luk, "Verification of streaming hardware and software codesigns," in *2012 International Conference on Field-Programmable Technology*. IEEE, 2012, pp. 147–150.