VirSoC: Automatic Synthesis of Virtual System-on-Chip Environments

Tashfia Alam, Indira Bhoomareddy Ramaiah, and Sandip Ray *Senior Member, IEEE*

***Abstract*—Modern System-on-Chip functionalities include significant software interacting closely with low-level hardware to realize system functionalities. This software is developed concurrently with the hardware and must be validated before the hardware is fabricated. Current industrial practice depends on the creation of virtual prototyping environments to enable the validation of such software. However, creating such prototypes is complicated, manual, and error-prone. In this paper, we propose a novel infrastructure, VirSoC, for automatically generating virtual prototyping environments. VirSoC includes an architecture and CAD flow to integrate different design blocks available in different abstraction levels to create a coherent, uniform view of SoC functionality suitable for early software validation. We show several case studies illustrating the applicability of VirSoC.**

***Index Terms*—Hybrid Virtual Platform, Hardware-software co-validation, Stub IP**

I. Introduction

Recent years have seen a significant rise in the amount and complexity of embedded software code integrated into System-on-Chip (SoC) designs, often blurring the boundaries between hardware, firmware, and software components. Software is now used to implement a variety of low-level functionalities, such as fine-grained power management, authentication, updates, and protocols for interfacing with off-chip components. A key advantage of realizing a system functionality through software instead of custom hardware is that software can be more easily updated or "patched" in field, in response to bugs or vulnerabilities detected after deployment or to changing security, functionality, or performance requirements during the lifetime of the system. The trend towards moving more and more system functionality to software is anticipated to rise at an even sharper gradient in future as time-to-market schedules continue to shrink, resulting in a consequent shrinkage of verification time and a consequent increase in escapes to deployment.

A key requirement for critical, low-level system software is that they must be functional at the same time that the hardware product is ready to ship. Consequently, the development and validation of such software must proceed concurrently with the hardware development and cannot wait until a mature silicon platform is available. A key challenge in the early validation of such software is the availability of a mature hardware platform on which to run (and test) the software. Note that running the software on a previous generation SoC is *not* a viable approach: software implementing low-level system functionality is generally written to interact with various components of the *targeted* hardware, which may not be available in a platform implementing a previous variant of the same SoC. On the other hand, running software workloads on a Register-Transfer-Level (RTL) simulator is prohibitively expensive (see Section II). Consequently, a fast hardware model on which to run software is not available until the hardware is mature enough for running on emulators or FPGAs (or early silicon), which can only occur late in the design life-cycle.

The current industrial practice to address this problem is using virtual platforms. The *virtual platform* $\mathcal{V}_{\mathcal{P}}$ of a hardware platform $\mathcal{P}$ is an abstract software model implementing the high-level functionality of $\mathcal{P}$. Software intended to run on $\mathcal{P}$ is then developed and validated on $\mathcal{V}_{\mathcal{P}}$. Unfortunately, the application of virtual platforms as a vehicle for developing or validating low-level system software is limited. In particular, since many internal details of $\mathcal{P}$ are abstracted or omitted in $\mathcal{V}_{\mathcal{P}}$, subtle corner cases of hardware-software interactions are missed when running software on $\mathcal{V}_{\mathcal{P}}$. Industry has attempted to address this problem through a variety of *hybrid* virtual platforms, which include some integration of some functionality as hardware models, *e.g.*, as RTL blocks, while the rest of the functionality is in software prototypes. However, putting together a hybrid virtual platform in today's practice is a highly manual and time-consuming exercise, often taking weeks to months. Furthermore, validation of different software requires different hybrid platforms. For instance, if a software functionality involves configuring the cryptographic units for specific operation modes, then it is critical that the cryptographic engine IP is integrated with a low-level (RTL or netlist) implementation in the hybrid platform used for developing and validating the software. On the other hand, if the software is a computer vision implementation, the IPs required with low-level implementation may include DSP accelerators and vector processing units. Developing and maintaining such hybrid virtual platforms represents a complex, expensive, and time-consuming bottleneck for early software development and validation.

In this paper, we address this problem by introducing a novel infrastructure, VirSoC, for automatically creating hybrid virtual platforms integrating components in software and hardware (RTL/netlist) abstractions. We refer to the platform generated by VirSoC as a *virtual SoC environment*. Although the virtual SoC is composed of a collection of heterogeneous components at different levels of abstraction, the heterogeneity

Tashfia Alam and Sandip Ray are with the Department of Electrical and Computer Engineering, University of Florida at Gainesville, Gainesville, FL 32611, USA. Email: tashfia.alam@ufl.edu, sandip@ece.ufl.edu. Indira Bhoomareddy Ramaiah is with Cadence Design Systems, Inc, San Jose, CA 95134, USA. Email: indirar@cadence.com.

The second author contributed to this research while being affiliated to the University of Florida.

of the underlying components is hidden from the user of the platform utilizing it for early software validation: the user is provided with the unified view of an integrated SoC platform on which to run the software targeted for early validation. To our knowledge, VirSoC represents the first automated flow for generating hybrid virtual platforms with heterogeneous IP collateral. The ideology for the automated tool flow is to enable usage of the generated platform by exercising any use case without considering how the hardware components take the inputs. We demonstrate the use of VirSoC with a number of case studies.

The key insight behind VirSoC is the observation that IPs integrated into an SoC generally include validation testbenches in addition to the implementation of the IP functionality. The traditional role of a testbench is to enable smooth and systematic application of test inputs to validate the IP. However, an indirect characteristic of any testbench is the ability to translate inputs typically provided as transaction-level collateral into a form that can exercise the target IP. VirSoC exploits this observation to create an SoC-level testbench from those of the individual IPs that is employed as a conduit for interfacing modules in VP with IPs in RTL.

The paper makes the following important contributions.

- We develop a novel infrastructure VirSoC for configuration and automated generation of hybrid virtual platforms.
- We show how we can use VirSoC to enable early validation of SoC functionalities by providing suitable platforms that help explore critical corner cases.
- We demonstrate the use of VirSoC with several case studies.

The remainder of the paper is organized as follows. Section II discusses the requisite background and provides an overview of existing prototyping solutions. Section III discusses the challenges towards developing hybrid prototyping solutions and how VirSoC addresses those. Section IV illustrates the VirSoC architecture and tool flow. Section V explains the application of VirSoC platforms in hardware-software co-validation, demonstrating several case studies targeting different applications. We discuss the role of VirSoC in reducing the manual effort in creating hybrid platforms along with the overhead and functional efficiency of VirSoC platforms in Section VI. We conclude in Section VII.

II. Background and Related Work

A. Virtual Platforms

Virtual platform (VP) has emerged as a critical infrastructure for enabling early software development for modern SoC designs. The idea is to develop a (virtualized) software abstraction of the underlying hardware platform and use this abstract model in lieu of hardware in early design and validation of software intended to run on the platform. To enable effective abstractions and enable the VP to be ready early in the development cycle, only a part of hardware functionality that is visible to the software running on the system is modeled, *e.g.*, low-level timing and power management functionality are removed, while the functionalities of the interface registers that can be written by software are preserved. VP models for different hardware IPs are generally developed using a high-level language such as SystemC-TLM (Transaction Level Model) [1]. Many commercial CAD vendors include infrastructure for the streamlined implementation of VP models. Some commercial offerings include Virtualizer™ from Synopsys [2], Helium™ from Cadence [3], and Vista™ from Siemens [4]. Wind River Simics [5], [6] enables simulation of high-level behavior of target hardware and exploration of hardware-software interactions. Additionally, there are open-source virtual prototyping infrastructures: a popular infrastructure is *e.g.* QEMU [7], which has been used for a broad spectrum of validation tasks. Obviously, many low-level details of the hardware are abstracted in the VP model, and consequently, the use of VP as a validation vehicle may miss corner cases dependent on such detail. To address such limitations, various *hybrid* virtual platforms (HVP) are architected in current industrial practice, where some hardware blocks are modeled in software, but for others, a hardware model is integrated with the VP. The abstraction level for the integrated hardware model can vary, *e.g.*, there are hybrid platforms with some IPs in RTL model, FPGA, or even a previous silicon implementation of the product. Hybrid VPs can expose system-level corner cases by enabling the user to view low-level details of hardware-software interaction [13]. However, the design of hybrid VPs is challenging in today's practice and can incur extensive human effort (see Section III).

B. Hardware-software Co-validation Basics

SoC designs are created by the integration of a variety of pre-designed hardware and software blocks, — often referred to as "Intellectual Properties" or "IPs", — which coordinate through system-level fabrics to realize the overall system functionality on a single substrate. In modern SoC designs, many key functionalities are implemented through complex hardware/firmware interactions. Consequently, validation of the integrated hardware/firmware/software stack is critical for the correct and trustworthy functionality of the system. The term "hardware-software co-validation" refers to techniques for validating this integrated stack, typically early in the design development prior to the availability of silicon implementation of the hardware. Obviously, verification of software components requires *some* underlying hardware model on which to run the software. Consequently, industry has invested in a variety of platforms for early validation of software (before the hardware is fabricated in silicon). These platforms are diverse and complex and include Emulators, FPGAs, and virtual platforms as well as various hybrids [14], [15], [16], [17], [18], [19], [20], [21]. However, each platform entails a trade-off between various parameters, including simulation speed, design abstraction, modeling effort, and cost. For instance, conventional RTL simulation models provide a detailed model of the hardware, but the simulation speed of RTL is about a billion times slower than the silicon, making it infeasible to explore complex hardware-software use cases [13]. FPGA prototyping [22], [23] provides a higher execution speed that facilitates the exploration of embedded software. However, the

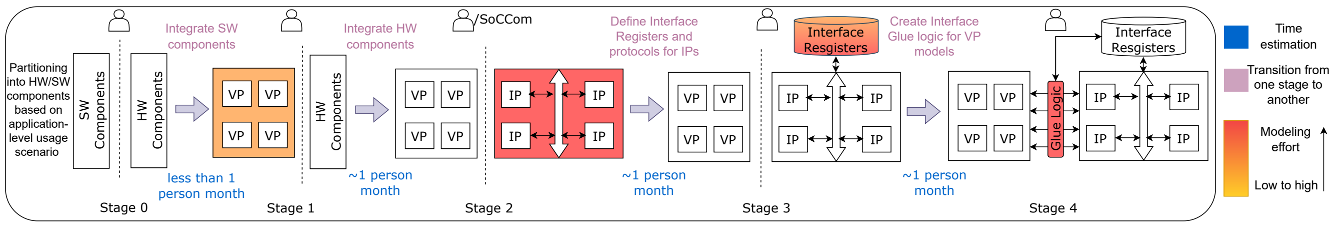


Figure 1. Industrial practice of Hybrid Virtual Prototype Formation. The time estimates for each stage are culled from publications, tutorials, and keynotes of industrial researchers [8], [9], [10], [11], [12] as well as experience of the authors working on hybrid VP design in companies like Intel and Qualcomm.

observability of design internals is limited in FPGAs, making it challenging to perform hardware debugging. Additionally, the number of input/output ports in FPGAs may restrict the scope of SoC implementation, incurring high resource costs. Emulators [24], [25] reproduce the functionality of an IP or SoC and offer reasonably high exploration speed with substantial observability, enabling hardware-software co-validation of SoC designs; however, emulation machines are highly expensive and consequently may not be suitable for IP-level verification. Furthermore, both FPGA and emulation solutions are available only when the hardware design is relatively mature and thereby preclude early validation.

C. HVP-based Hardware-Software Co-validation Practice

Early hardware-software validation using hybrid VP in industry includes actions from a variety of stakeholders and several handoffs. Figure 1 provides a simplified view of the activities involved to provide a flavor of the complexity and handoffs. Roughly, the activities can be divided into two phases: (1) creation of the appropriate hybrid VP that is effective in running the targeted scenarios and tests and (2) execution of the tests and collecting coverage data from the platform (and finding bugs). Note that the hybrid VP must be configured with the targeted tests in mind, *e.g.*, if the test involves interaction of a software application with a cryptographic module, the platform must be assembled to enable executing the software on a VP running the Instruction Set Architecture (ISA) for the microprocessor involved, while the crypto module involved must be integrated as an IP in RTL or gate-level netlist. To comprehend the abstraction level necessary for each component, the usage scenarios to be exercised are first analyzed and partitioned into hardware (RTL or netlist) and software components (Stage 0). The software components are mapped to VP models (Stage 1). The hardware components are integrated with the communication fabric to form the hardware subsystem (Stage 2). For the hardware subsystem, additional interface registers and protocols are defined for each IP to enable interaction with the VPs (Stage 3). Finally, glue logic is created to compose the VP and hardware models into an integrated subsystem (Stage 4).

As shown in Figure 1, the design of a hybrid VP takes several months of manual effort for typical validation teams in companies like Intel and Qualcomm. One key reason for this significant manual effort is the need to reconcile heterogeneity and inconsistency among interfaces in the constituent IPs, as explained in Section III. Furthermore, note that the platform being created is targeted towards the specific usage scenarios to be run and the hardware-software corner cases being explored. Different tests and usage scenarios require different hybrid VPs, *e.g.*, a power management scenario requires a different set of components in RTL vs software than a cryptography use case, even for the same targeted SoC design. Consequently, a large part of the manual effort involved needs to be replicated for each such hybrid VP instance created (for each class of use cases).

D. An SoC Compiler

Our work makes use of an SoC compiler tool SoCCom that was developed in previous work by Deb Nath *et al.* [26]. This tool takes hardware IP blocks (implemented in RTL) together with metadata specifying integration topology and constraints and generates an integrated SoC (Algorithm 1, refer to Table II for a glossary of notations). The tool enables the configuration of design parameters such as bus width, memory size allocation, memory-mapped I/O, and peripheral address size. The SoC design is then implemented by creating a top-level RTL *glue logic* that integrates the constituent IPs and generates interfaces obeying the specified configuration. SoCCom also enables the generation of multi-core SoCs with Network-on-Chip fabrics.

Algorithm 1: SoC formation using SoCCOM

```
Input:
  Configuration file ($C_{FG}$) containing
   (a) SoC specification (connectivity information)
   (b) Pointer to RTL IPs ($R$)
Output: SoC RTL ($S$)
1  Procedure SoCCOM ($C_{FG}, R$)
2      $\langle E_{IO}, E_P, E_H \rangle \leftarrow$ HDL-ANALYZER($C_{FG}, R$)
3      $S \leftarrow$ SoC-GENERATOR($E_{IO}, E_P, E_H$)
4  Procedure HDL-ANALYZER($C_{FG}, R$)
5      $\langle E_{IO} \rangle \leftarrow$ EXTRACT-I/O($C_{FG}, R$)
6      $\langle E_P \rangle \leftarrow$ EXTRACT-PARAM($C_{FG}, R$)
7      $\langle E_H \rangle \leftarrow$ EXTRACT-HEADER($C_{FG}, R$)
8      return $\langle E_{IO}, E_P, E_H \rangle$
9  Procedure SoC-GENERATOR($E_{IO}, E_P, E_H$)
10     CREATE TOP-LEVEL SoC MODULE
11     INCLUDE $E_H$
12     INSTANTIATE FABRIC IP BASED ON $E_P$
13     INSTANTIATE AND MAP $E_{IO}$ TO FABRIC PORTS
14     $S \leftarrow$ FORM TOP-LEVEL RTL
15     return $S$
```

Algorithm 1 shows the System-on-Chip (SoC) formation process using SoCCom. The initial phase in SoC compilation involves the preparation for integration of all IPs and interconnect components. To achieve this, SoCCom employs an HDL-Analyzer module to systematically extract and consolidate IP integration standards, encompassing headers such as define, import, and include for each IP using Extract-Header function, port definition, port types using Extract-I/O function, and width specifications and local or global parameters using Extract-Param function. Using the SoC-generator module, SoCCom then assimilates the metadata obtained from the HDL-Analyzer to produce interfaces that align with the design parameters. The parameters include address bus width, data bus width, memory size allocation, memory-mapped I/O, and peripheral address size. They are employed to create a glue logic to integrate the constituent IPs. The IPs are instantiated based on the parameters and interfaced with the relevant fabric to form the top-level RTL implementation of the SoC.

Note that SoCCom does *not* account for any heterogeneity in abstraction, *e.g.*, all IPs to be integrated must be available as RTL designs. Furthermore, SoCCom does not comprehend HW-SW interaction. Our work in this paper extends the idea of automated SoC creation to account for heterogeneity in abstraction and correspondingly enables HW-SW co-validation.

E. Related Work

There has been significant recent work to enable smooth development of Hybrid Virtual Platforms to enable early design and validation of software functionality. Table I provides a quick overview of the existing tools and infrastructures. Synopsys Virtualizer™ [2] includes a *Virtualizer Development Kit* to enable assembly of different blocks into virtual prototypes. VDK can link VP models (developed with Virtualizer) with FPGA-based systems using transactors. Wind River Simics [6] provides interfaces to integrate transaction-level models (TLM) for hardware IPs written in SystemC with the VP infrastructure. QEMU-based virtual platforms [27], [37], [38], [39], [40] have been extensively used to target exploration and validation of binary code for a variety of underlying processor architectures, including ARM, OPENRISC, RISCV, x86, MIPS, and SPARC. Likewise, OVP [31] offers a library of open-source cores and others collaterals needed for VP formation. Simics Vista [4] enables creation, debugging, and analysis of TLM models. Huang *et al.* [28] present an industrial case study of SoC HW/SW verification flow exploiting platforms at different abstractions such as RTL-FPGA co-simulation and SystemC-FPGA co-emulation. Kim *et al.* [9] implement a hybrid virtual platform that enables software development and verification of (Samsung) mobile Application Processor designs at the initial stage of RTL design. The platform integrates VP and HW emulators using transactors. CPU, memories, and other peripherals frequently used for Android platform boot-up reside on the virtual platform, while other (RTL) IPs are implemented on an emulator, maximizing the runtime performance of SW leveraging VP simulation speed. Kang *et al.* [29] demonstrate an industrial case study of SoC development and verification methodology by adopting hybrid VP technology combining high-level abstraction (C++) firmware running on SystemC models, and RTL designs, which helped detect many bugs at an earlier stage. Wicaksana *et al.* [32] present a VP and FPGA-based hybrid prototyping platform that provides a flexible system-validation solution for HW/SW co-design at different stages of development based on the availability of TLM and RTL implementations. Additionally, this methodology enables runtime debugging and offline analysis for early HW/SW architecture exploration. Similarly, Masing *et al.* [33] propose a NoC-based hybrid prototyping solution interfacing a VP and an FPGA using NoC links that provides an optimized interface with low latency. Choi *et al.* [30] introduce a VP solution incorporating two heterogeneous prototypes, i.e., SystemC and RTL, connected through IPC (Inter Process Communication). In addition, the proposed approach shows how it helps reduce setup time for other platforms, such as combining VP with HW emulator. Kabir *et al.* [34] demonstrate an exploration platform called ViVE that provides a configurable hybrid prototyping platform enabling users to explore and exercise system-level use cases targeting automotive security applications. VPSIM [35] integrates the QEMU emulator into SystemC, leveraging QEMU's high-performance emulation technologies like Dynamic Binary Translation and Paravirtualization. Cucchetto *et al.* [36] demonstrate the integration of SystemC with both QEMU and OVP, including support for cycle-accurate simulation of RTL models.

Despite the diversity of these platforms, the scope of each infrastructure is limited. Table I provides a detailed summary of the features and limitations of each available infrastructure. Some common limitations include automation support for creating the platform, as well as resource and communication overhead. In contrast, VirSoC streamlines the hybrid platform generation process by enabling automatic generation and exercise of such platforms, facilitating early validation. Additionally, VirSoC-generated platforms are configurable: the user can form a prototype irrespective of the heterogeneity of the IPs and realize the necessary platforms. The only constraint imposed by VirSoC is the requirement that IPs come with a standardized testbench, which is typically available for most commercial IPs, including both soft and hard IPs (see Section III).

III. Hybrid Virtual Prototyping Challenges and VirSoC Solution

A fundamental challenge in the development of HVPs is the need to address heterogeneity: it must be possible for IPs developed at different levels of abstraction to coordinate and communicate to realize the overall system functionality. Simply connecting the IPs through a standard port abstraction is insufficient. For instance, a microcontroller IP developed for VP is simply modeled to execute a sequence of instructions; low-level features of the hardware implementation of the microprocessor (*e.g.*, clocks) are abstracted in the model. On the other hand, an RTL IP for a bus controller can provide a cycle-accurate model of the controller functionality. Connecting

Table I
Features and Limitations of Related Work

Domain	Tool/Infrastructure	Pros	Cons	References
Industry	Synopsys Virtualizer Development Kit	Enables assembly of different blocks into virtual prototypes.	Not fully automated in terms of development and testing	[2]
	Wind River Simics	Full system simulation with early exploration, reproducibility and debugging features.	Resource intensive and associated with modeling overhead.	[6]
	QEMU	Open source emulator compatible with diverse ISAs to be used in a hybrid virtual platforms.	Needs additional support from tools for deployment in system-level applications.	[27]
	InPA Co-emulator	Supports multi-mode operations: RTL-FPGA co-simulation, SystemC-FPGA co-emulation, providing flexibility for different stage of SoC design flow.	Substantial efforts required for setting up the validation environment	[28]
	Hybrid Emulation Platform by Samsung	Facilitates early software development and demonstrated on android platform with significant speed-up.	While it helps in reducing manual design efforts, there is still some automation overhead and communication overhead between virtual and emulation platforms.	[9]
	An Industrial case study of SoC Verification using VPs	Provides a seamless verification methodology leveraging VPs demonstrating fault detection in early stages.	No support for automatic creation of platforms	[29]
	IPC(Inter Process Communication)-based Hybrid VP solution	Use of IPC between heterogeneous simulators facilitates communication and cooperation, potentially improving the efficiency of the co-verification process.	Requires significant building efforts and lacks automation support.	[30]
	Open Virtual Platforms (OVP) by Imperas Ltd.	Provides fast simulation, library of free open source processor and peripheral models, enabling developers to access a wide range of components for their simulations.	Associated with integration complexity, may require a learning curve and face limitations in model availability.	[31]
	Siemens Vista	Facilitates ESL design by enabling quick TLM model creation, debugging, and analysis, allowing for first-pass success in prototyping complex systems.	Requires a learning curve to adopt and limited model availability.	[4]
Academia	A VP and FPGA-based Hybrid prototyping Methodology	Offers flexibility and efficiency in validating hardware and software designs at various stages of development and includes features for runtime performance analysis and debugging.	Associated with complexity in synchronization of different abstractions, overhead from remote communication and limited comparison with existing work provided	[32]
	A Network-on-Chip (NoC)-based hybrid prototyping solution	Use of NoC links facilitates transparent operation, homogeneity, and suitability for repacketizing data; provides an optimized physical interface between the VP and FPGA with low latency.	Incorporates synchronization challenges due to no support for automation.	[33]
	ViVE	Enables early exploration and validation of system-level functionalities comprehending their interactions.	Limited to modeling of automotive systems and applications.	[34]
	VPSIM	Integrates QEMU-based emulation with dynamic accuracy control, and utilizes a Python front-end for expressive and efficient virtual prototyping.	Possible complexity in dynamic platform composition, reliance on XML for description, and careful tuning required for optimal performance.	[35]
	Co-simulation of SystemC models in QEMU and OVP virtual platforms	Allows for rapid prototyping and simulation of full HW/SW systems, offering benefits such as efficient management of design complexity and decoupling of software development from hardware availability,	Performance may be impacted compared to native QEMU-based approaches, and cycle-accurate simulation of QEMU/OVP and SystemC is not fully supported.	[36]

these two IPs to create a meaningful system-level functionality entails design of transactors to translate data into respective abstractions. In practice, careful design of interfaces enabling smooth functioning of a heterogeneous system designed by the composition of IPs at different levels of abstraction is a complex manual process involving weeks or even months [8]. Consequently, HVP designs have to be carefully planned to be in synchrony with RTL drops at different levels of maturity, and the system-level properties to be explored to ensure appropriate transactors and interfaces are in place. The situation is exacerbated by speed mismatch between different IPs, *e.g.*, exercising an event in RTL simulation takes several orders of magnitude more time than VP exploration of an event of comparable complexity.

VirSoC addresses the heterogeneity problem by the observation that appropriate collateral necessary for designing transactors is already available at the time of HVP integration, albeit not part of the integration flow. In particular, testbenches are developed for (RTL) IPs and available at the time of delivery. The goal of a testbench is, apparently, to perform functional validation of the IP. However, in the process, the testbench performs input translation to match the underlying design abstraction. For instance, typical testbenches for RTL or netlist IPs can take inputs in a standard transaction-level model (TLM) abstraction and apply them possibly across multiple clock cycles as required by the underlying IP. VirSoC repurposes this feature, turning the testbench into the intermediary for translating abstract transaction-level communications provided by the VP to (possibly multi-cycle) communication inputs as necessary for an RTL, netlist, or silicon IP. We show how to make this idea work in practice to enable automated creation of hybrid VP environments. Furthermore, this approach shields the VP models from the idiosyncrasies and optimizations involved in the communication interface of individual IPs and obviates the need for standardizing the access sequence for each IP: the existence of a testbench that can translate transaction-level communication payloads to inputs for the IP is sufficient. Obviously, this comes with the requirement for standardizing testbench structures so that the VP can seamlessly communicate with testbenches for different IPs. However, in our experience, this standardization does not induce additional cost, which is already occurring in current industrial practice with almost universal adoption of UVM [41].

IV. VirSoC Architecture and Tool flow

We start with a motivating example to explain the key ingredients of VirSoC. Suppose the envisioned SoC architecture includes a CPU running an X86 instruction set, as well as an AES encryption module, connected to a Wishbone bus (together with other IPs), as shown in Fig. 2. Consider a software binary $\mathcal{B}$ slated to execute on the CPU on top of an operating system, and suppose $\mathcal{B}$ includes invocation of an encryption function which is to be implemented in the integrated SoC by invoking the AES module. In a silicon implementation, $\mathcal{B}$ (and the OS) would execute on top of the silicon implementation of the X86 CPU, and the CPU, together with the AES IP and other hardware IPs would be integrated possibly with a system bus and communicating with each other through bus transactions. However, our goal is to exercise and validate the interaction involved *before* an SoC has been fabricated in silicon. Suppose we have a VP model of the X86 microcontroller, together with RTL IPs such as Wishbone bus and AES (among others). To implement this

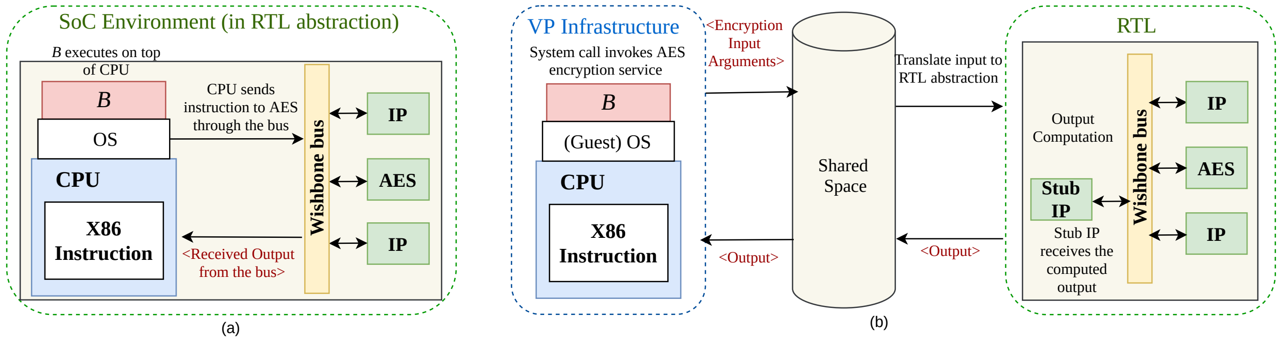

SoC Environment (in RTL abstraction)
B executes on top of CPU
B
OS
CPU
X86 Instruction
CPU sends instruction to AES through the bus
<Received Output from the bus>
Wishbone bus
IP
AES
IP
(a)
VP Infrastructure
System call invokes AES encryption service
B
(Guest) OS
CPU
X86 Instruction
<Encryption Input Arguments>
<Output>
Shared Space
Translate input to RTL abstraction
<Output>
RTL
Output Computation
Stub IP
Stub IP receives the computed output
Wishbone bus
IP
AES
IP
(b)

Figure 2. A sample program P Execution in an SoC Environment. (a) Flow of operations in a real SoC. (b) Flow of operations in the Hybrid VP system.

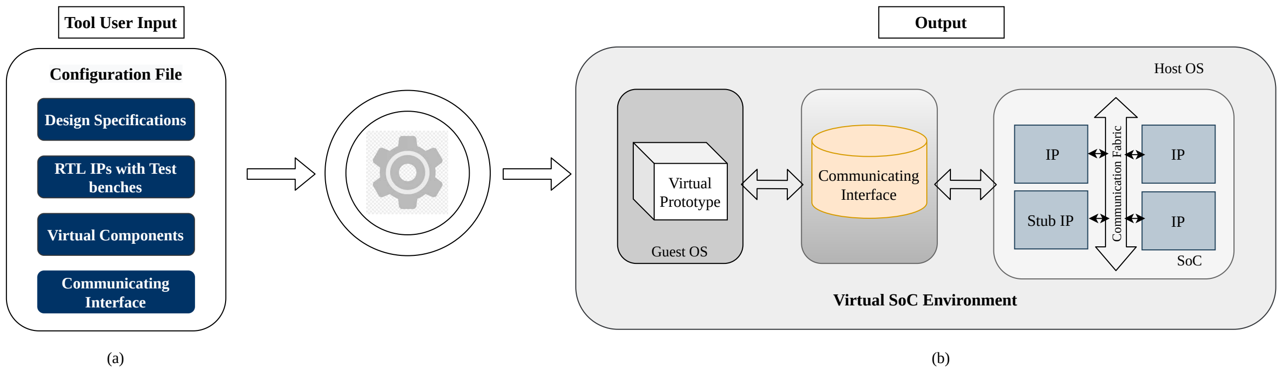


Figure 3. An Example Virtual SoC formation through VirSoC. (a) User Input. (b) Generated Virtual SoC Environment.

interaction, the following would be a typical flow of operations in most hybrid VP systems.

1) The program $\mathcal{B}$ includes a system call through a special command when it needs to invoke the encryption function from a different RTL IP. The system call is trapped by the guest OS as a special command and passed along via the VP infrastructure through the host OS.
2) The encryption command and its arguments are passed into a special space in the filesystem under the Host OS, from where it is accessible to the RTL simulator running on the host system.
3) The argument is translated (either manually or through some tool) to inputs in the RTL abstraction. When the model involves a bus like Wishbone, this entails sending the command as a bus transaction directed to the AES IP.
4) THe AES IP (in RTL) receives the encryption directive (as a bus transaction), performs the encryption functionality, and returns the result as a bus transaction directed to the CPU.
5) Since the CPU is in VP, the bus output is captured and transmitted to the VP model, which is passed on by the guest OS as the result of the computation to program $\mathcal{B}$.

VirSoC creates a CAD infrastructure to automate the interactions and data flow between VP and RTL (resp., silicon, or FPGA) models so that the heterogeneity of abstractions is hidden from the user. Instead, a homogeneous view of an SoC environment is offered where the user only needs to provide the binary $\mathcal{B}$ as they would have when interacting with the SoC in silicon. The result is a powerful validation platform that removes the complexities of manually developing VP with various hardware models to support different use cases while enabling early exploration and validation of hardware-software use cases.

Figure 3 shows the overall flow of VirSoC and the environment created. The tool "glues together" the different SoC components at various levels of abstraction: (1) creating an SoC-level testbench from individual IP-level testbenches; (2) integrating different virtual abstractions to create a unified SoC environment. It uses a (user-specified) shared address space[1] to enable smooth coordination between components modeled at different levels of abstraction; this address space is used as a shared read-write memory to enable communication of computation commands or results (See Section V). Platform generation of the VirSoC makes use of the following two ingredients.

Integrated Testbench: The purpose of the testbench is to mediate between the RTL environment and the IPs in virtual abstractions and translate the incoming inputs to the format the RTL subsystem receives. Testbenches are conventionally included with the designs to enable validation. A key functionality of a testbench is to translate incoming (transaction-level) inputs to match the underlying abstraction of the implementation. We use this indirect functionality of the testbench to enable smooth communication of data from VP models to low-level abstractions. In our motivating example, the encryption command created by the VP module is a transaction-level software command that is passed on to the testbench, which

[1]In typical VirSoC implementations on a server with an operating system, the shared address space can be implemented as a folder or directory managed by the OS. The VirSoC framework supports such implementation but is not tied to the specifics of the implementation. In our implementation, the user is permitted to designate a shared address space as a configuration parameter.

translates it to appropriate inputs for an RTL module for the target SoC (*e.g.*, appropriate values for activating the bus signals, sending the command through the bus to the crypto IP). Note that this functionality is integrated into the testbench for the crypto IP already since it is required for functional verification of the IP core. By integrating the IP testbenches, VirSoC avoids the need for developing a complicated (and customized) translation mechanism for the VP to communicate with each IP.

Stub Modules: Stub Modules are dummy IPs implemented in RTL for each VP component in the environment. The purpose of the stub IP is to receive the computed results from the communication fabric of the SoC and deliver them to the external environment. To understand its role, note that in our motivating example, the AES IP, after completing its computation, must send the result to the CPU (step 4). However, the CPU itself is in VP, and therefore, the result must be captured and translated back to the VP environment (step 5). In VirSoC, the stub IP facilitates this as follows. There is a stub IP for the CPU in RTL that is connected to the bus using the same interface that the CPU would have been connected to had it been in RTL. The computed result from the AES block is consequently communicated to this IP. The functionality of a stub IP is simply to output anything it receives to the external environment in the shared space, where it can then be "picked up" to transfer to the VP.

Algorithm 2 shows the high-level VirSoC platform generation algorithm. The algorithm uses several configuration parameters, including information about various architecture and IP collateral involved, *e.g.*, the location of the different IPs at different abstractions (RTL, VP, FPGA), list of the IP-level testbenches and the shared address space used for communication between different abstraction levels. VirSoC platform generation then operates in two steps: (1) Creating a structurally integrated block of IP components in the same level of abstraction, *e.g.*, in this step, all RTL IPs are integrated and coordinated to form a cohesive RTL subsystem; and (2) assembling and gluing together design blocks at different levels of abstraction, *e.g.*, in this step, interfaces are generated to enable VP modules to interact with RTL, FPGA, and silicon IPs. Note that the first step essentially integrates RTL IPs and can consequently be realized through repurposing the SoCCom framework (Algorithm 1). However, unlike SoCCom, it uses stub modules in place of the IPs that are not available in RTL,

Table II
Glossary of Notations

Term	Definition
C_{FG}	Configuration file
R	RTL IPs
I_{TB}	IP-level Test benches
S_T	Stub IP
$\mathcal{VP}$	Virtual Platform
C	Communication channel between VP and RTL abstraction
S	Generated SoC with RTL IPs
S_{TB}	SoC-level Test bench
S_C	Collection of scripts gluing SoC with virtual abstractions to enable exercise use cases
E_{IO}	Extracted I/O from C_{FG} and R
E_P	Extracted parameters from C_{FG} and R
E_H	Extracted headers from C_{FG} and R
E_F	Extracted function names from I_{TB}
E_M	Extracted parameters from I_{TB}
M_{TB}	Modified IP-level Test bench with added suffix

Algorithm 2: Virtual SoC Environment Generation

```
Input:
  Configuration file ($C_{FG}$) containing
  (a) Connectivity specification of RTL IPs
  (b) Pointer to RTL IPs ($R$) and Stub IPs ($S_T$)
  (c) Pointer to IP-level testbenches ($I_{TB}$)
  (d) Placeholder variables receiving stimuli in IP
      testbenches ($E_I$)
  (e) List of Virtual Platforms ($\mathcal{VP}$)
  (f) Pointer to shared address space between each
      virtual component and RTL components ($C$)
Output: Virtual SoC environment
1 $S \leftarrow$ RTL-Subsystem-Gen($C_{FG}, R, S_T$)
2 $S_{TB} \leftarrow$ SoC-Testbench-Gen($C_{FG}, I_{TB}, E_I$)
3 $S_C \leftarrow$ Connecting-Framework($C_{FG}, C, S_{TB}, E_I$)
4 return $\langle S, \mathcal{VP}, S_C \rangle$
```

Algorithm 3: RTL Subsystem Generation

```
1  Procedure RTL-Subsystem-Gen ($C_{FG}, R, S_T$)
2      foreach $\mathcal{VP} \in C_{FG}$ do
3          $C'_{FG} \leftarrow$ Include-$S_T$-Spec ($C_{FG}$)
4      end
5      $S' \leftarrow$ SoCCOM($C'_{FG}, R$)
6      $F_{IO} \leftarrow$ Trace-fabric-ports ($E_{IO}$)
7      $S' \leftarrow$ Connect-to-top ($F_{IO}, S'$)
8      foreach $S_T \in C'_{FG}$ do
9          $S_{TO} \leftarrow$ Trace-$S_T$-output-ports ($E_{IO}$)
10         $S \leftarrow$ Connect-to-top ($S_{TO}, S'$)
11     end
12     return $S$
```

Algorithm 4: SoC Testbench Generation

```
1  Procedure SoC-Testbench-Gen ($C_{FG}, I_{TB}, E_I$)
2      foreach $I_{TB} \in C_{FG}$ do
3          $E_F \leftarrow$ Extract-func-name($I_{TB}$)
4          $E_M \leftarrow$ Extract-param($I_{TB}$)
5          $\langle E'_F, E'_M \rangle \leftarrow$ Add-suffix($\langle E_F, E_M \rangle$)
6          $M_{TB} \leftarrow$ Modify-Tb ($I_{TB}, \langle E_F, E_M \rangle, \langle E'_F, E'_M \rangle$)
7      end
8      $S'_{TB} = \sum_{i=1}^{n} M_{TBi}$
9      $F \leftarrow$ Trace-connected-fabric-ports($S, E_I$)
10     $S_{TB} \leftarrow$ Modify-Tb ($S'_{TB}, E_I, F$)
11     return $S_{TB}$
```

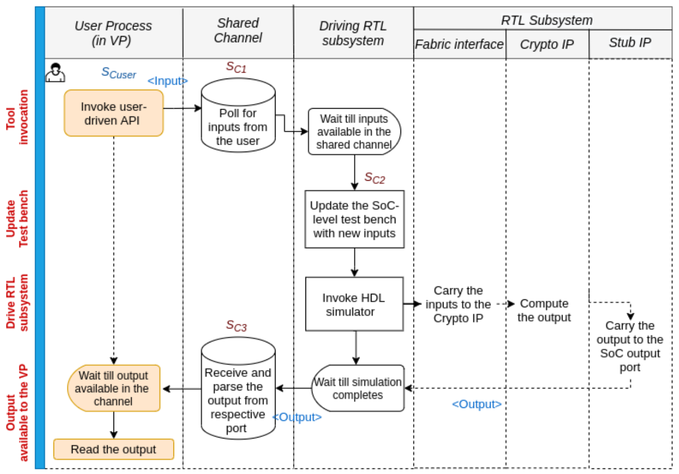

User Process (in VP)
Shared Channel
Driving RTL subsystem
RTL Subsystem
Fabric interface
Crypto IP
Stub IP
Tool invocation
Update Test bench
Drive RTL subsystem
Output available to the VP
S_{Cuser}
<Input>
Invoke user-driven API
S_{C1}
Poll for inputs from the user
Wait till inputs available in the shared channel
S_{C2}
Update the SoC-level test bench with new inputs
Invoke HDL simulator
Carry the inputs to the Crypto IP
Compute the output
Carry the output to the SoC output port
S_{C3}
Receive and parse the output from respective port
Wait till simulation completes
<Output>
<Output>
Wait till output available in the channel
Read the output

Figure 4. A sample use case demonstration showing a series of S_C execution initiated by user-API invocation.

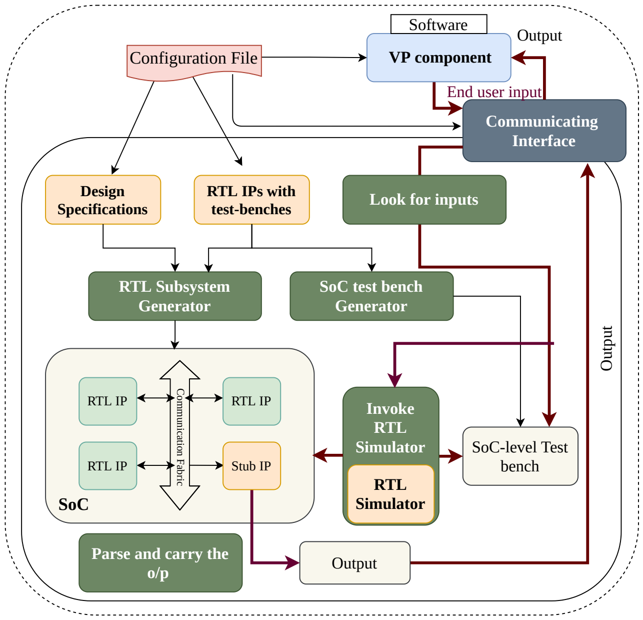


Figure 5. VIRSOCArchitecture: Design Components & User interactions with the platform. *The bold arrow shows the automation flow to carry out the user instructions.*

as described below (see below). The second step creates the integrated testbench for the coordination of IPs at different levels of abstraction (see Algorithm 4). Furthermore, APIs (S_C) are generated to enable communication of different components through testbenches by using a connecting framework (see the last step of Algorithm 2).

Generating RTL Subsystem: Algorithm 3 shows the RTL subsystem generation. The key idea is to configure and include the stub modules and then use SoCCOM to synthesize the RTL subsystem.[2] For this purpose, INCLUDE-S_T-SPEC generates a SoCCOM-compatible intermediary configuration file C'_{FG} that includes the connectivity information of the stub IPs in place of the virtual components. For each $\mathcal{VP}$ present in the C_{FG}, VIRSOC specifies a respective stub module, *i.e.*, it specifies a stub IP in the C'_{FG} for the mentioned example. The connectivity specification of the stub IPs follows a similar structure to the regular RTL IPs (R) specified. SoCCOM then combines the IPs in RTL, including stub IPs, and forms an RTL subsystem with the specified communication fabric (see Section II-D). To realize that, SoCCOM first systematically extracts and aggregates the design metadata, including headers, such as define, import, and include (components) of each IP, network adapter, router, shared-bus crossbar, port definition, port types, and width, and local or global parameters. The extracted metadata is then utilized to create a compatible interface for each IP with the communication fabric (see Algorithm 1). The TRACE-FABRIC-PORTS function parses the fabric ports from the extracted metadata (E_{IO}), and CONNECT-TO-TOP function introduces them to the (SoCCOM generated) top module S' to enable receiving external stimuli provided by a testbench. Correspondingly, for each stub IP specified, the output ports are connected to the top module to deliver the respective output to the external environment and finally realize RTL subsystem S. In our example, this step entails creating the RTL subsystem, including the crypto IP and other RTL blocks with a stub module for the CPU to form a (bus-based) RTL subsystem.

Generating SoC-level Testbench: Algorithm 4 shows the SoC-level testbench generation by assembling the testbenches from individual IPs. The algorithm first checks that all IP-specific testing components are available, compatible, and can be integrated into a single testbench. For this purpose, the functions EXTRACT-FUNC-NAME and EXTRACT-PARAM parse and extract necessary information from each testbench, including the functions' names and parameters, and add IP-specific suffixes to differentiate them (implemented by ADD-SUFFIX).[3] The algorithm modifies each testbench with the new function names and parameters using MODIFY-T_B function. This avoids possible redefinition, conflicts of the same function, or parameter names used by different testbenches. In the example, this step would ensure the appropriate crypto IP is provided the stimuli given the presence of other crypto IPs with similar testing components. Next, the modified IP testbenches are merged to form an intermediary testbench (S'_{TB}). A critical aspect of having an SoC-level testbench is abstracting direct communication with the IPs and enabling communication through the fabric. To realize this, TRACE-CONNECTED-FABRIC-PORTS parses the RTL subsystem (S) to trace the fabric ports the local IP input ports E_I (that receive stimuli) are mapped

[2]For pedagogical reasons, we present the description of the RTL system in this paper as if it is incorporated into a simulation environment. However, the VIRSOC infrastructure itself is independent of whether the RTL modules are exercised through a simulation environment, FPGA, or silicon. The only requirement is that the different hardware IPs are accessible through the testbench. See below.

[3]If two instances of the same IP are used, this step will ensure that the two instances are individually invocable with distinct names.

to and MODIFY-T_B replaces them with associated fabric ports in the integrated testbench. In our example, the tool forms a testbench that integrates the crypto and other IPs' testing components, providing stimuli to the top-level RTL subsystem via the wishbone bus.

Generating a Complete Environment: Finally, to enable the users to work with IPs represented in different formats and ensure smooth interoperability between different levels of abstraction, the last step for VIRSOC (Algorithm 2) is to create a complete composite subsystem by gluing together the different available collateral. The function CONNECTING-FRAMEWORK (Algorithm 2 Step 3) performs this task by generating a variety of scripts (S_C) to integrate IPs represented in different abstractions and ensure smooth interoperability. This function parses the configuration file to generate a variety of interface functions that can be used by the user; it also extracts a variety of logistic and book-keeping information (*e.g.*, the number of bytes to be transmitted from an IP to through the fabric to support a specific computation service). Figure 4 provides a detailed visualization of how these scripts glue different components of VIRSOC and the execution flows and hand-offs necessary to support a use case involving the VP and RTL components. For our crypto example, scripts include interfaces for "consuming" a request coming from the user program in VP and translating the sequence of bytes output through the stub module to transaction-level data transmitted to the VP.

V. CASE STUDIES

We applied VIRSOC on several hardware-software co-validation case studies. Here, we summarize three representative case studies. The case studies discussed here are sanitized and simplified from typical industrial hardware-software validation use cases to illustrate the role of VIRSOC. Table III provides an estimate of the gate count of the target SoC designs involved and respective platform creation time. Note that VIRSOC is independent of the target design size and can effectively support the automatic creation and testing of platforms across a range of design complexities.

Remark 1: Although VIRSOC design itself is independent of the specifics of the VP models, communication fabric, or the implementations and abstractions of the IPs in RTL or FPGA, any case study with the infrastructure requires a specific instantiation. For the case studies discussed here, we use a VP based on QEMU-x86 that incorporates the virtual model of a processor implementing the IA32 Instruction set, running Ubuntu 18.04 as a guest OS. The RTL IPs discussed here are SystemVerilog modules exercised with a SystemVerilog simulator running on a Linux server; the communication fabric is a Wishbone bus.

Remark 2: In the case studies below, note that the initial segment of the flow of operation (first four columns) is exactly the same for all use cases. See Figures 6, 7, and 8. Furthermore, the last two columns are different for each flow only in the payload of the computed result and the IP from which the payload originates. The VIRSOC framework automates all these hand-offs, leaving the user with only the requirement of activating the operation flow through a user-level interface invoked from the software in VP.

Cryptographic Application: The goal of this case study is to explore and validate the interaction of software driver with crypto IP in the SoC. It is a slight elaboration of the motivating example discussed in Section IV. The use case corresponds to software running on the processor invoking the service of the crypto module, the crypto IP performing the computation, and the results transmitted back to the software driver running on the processor. In this hybrid VP exploration, the software runs on the VP platform (as an X86 binary), while the crypto IP is in RTL. Our instantiation includes two different crypto IPs, AES and DES3. Figure 6 shows the corresponding operation flow. Note that for VIRSOC, the only change to be made to explore the different interactions with the crypto IPs is for the user program to switch the encryption command directive; the platform handles the rest of the flow automatically.

Processor-Memory Co-validation: The goal of this use case is to ensure that software running on a processor correctly performs load and store operations to the different memory units. Figure 7 defines the operation flow for this use case. Here, we use the processor IP in VP that interacts with a RAM IP. The use case explores various ways in which the processor performs a variety of load-store operations on the RAM. Other than the update to the driving software, no other human intervention is required to set up the platform to explore the use case.

DSP Application: In this use case, we consider software interacting with an IIR (Infinite Impulse Response) IP. The interaction (Figure 8) is a sanitized instantiation of representative DSP applications. The IIR IP takes a series of input samples from which it calculates the output samples. Note that this entails a highly analog response. However, from VIRSOC standpoint, the platform generated simply needs to incorporate the IIR functionality (and the stub module for the processor in RTL) to interact with the software running on the VP.

VI. DISCUSSION

A. Reduction in Manual Effort

It is instructive to reflect upon the manual effort reduction facilitated by VIRSOC vis-a-vis the current state of the practice discussed in Section II-C. In particular, note that VIRSOC generates the hybrid VP from constituent available collateral *automatically*. Note that for each of the case studies above, the platform generation through VIRSOC takes less than a second. Furthermore, the only user activity entails activation of the operation flow, through the user-level interface invoked from the software VP (as well as the design of the test software for exercising the use case). ***This is in stark contrast to current practice, where the setup of a hybrid VP targeted for a specific class of use cases takes several person months (Figure 1).***

Obviously, as explained above, this significant reduction in manual effort is achieved only via the standardization of testbench structures. Indeed, the entire VIRSOC infrastructure

Table III
Estimated Design size (in NAND2 equivalent) and Hybrid VP Creation Time

Use cases	Estimated gate count of the Core	Gate count of the rest	Estimated gate count of the envisioned SoC	Hybrid VP Formation time (sec)
VP Firmware uses RTL AES IP during secure boot for derivation of crypto keys	200k	125k	325k	0.4
VP Firmware uses RTL DES3 IP during secure boot for derivation of crypto keys	200k	130k	330k	0.47
VP Firmware uses RTL IIR IP to perform certain filtering operation	200k	135k	335k	0.45
VP Firmware uses RTL RAM IP to perform certain memory write and read operation	200k	125k	325k	0.44

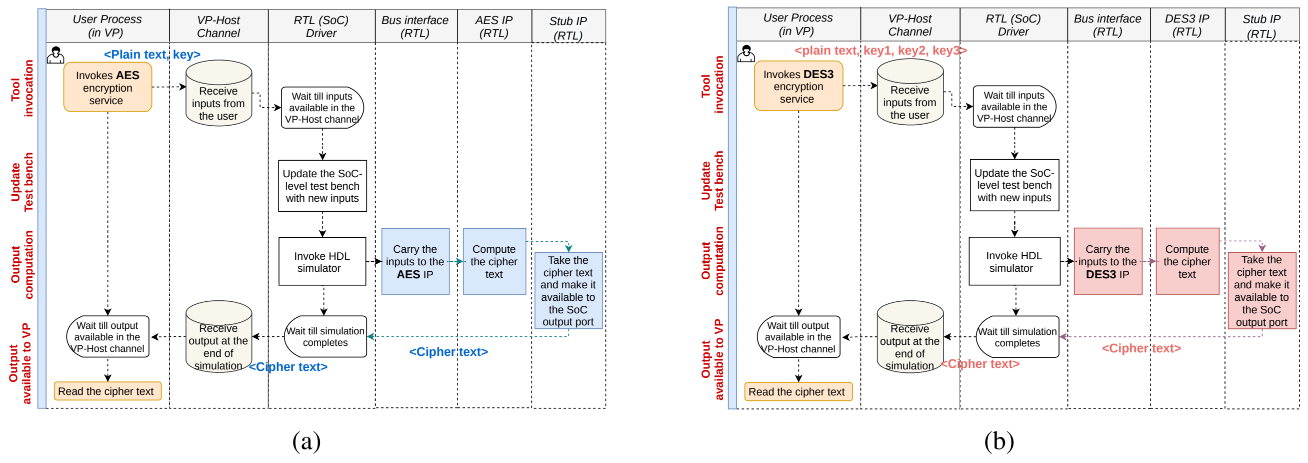

User Process (in VP)
VP-Host Channel
RTL (SoC) Driver
Bus interface (RTL)
AES IP (RTL)
DES3 IP (RTL)
Stub IP (RTL)
Tool invocation
Update Test bench
Output computation
Output available to VP
<Plain text, key>
<plain text, key1, key2, key3>
Invokes AES encryption service
Invokes DES3 encryption service
Receive inputs from the user
Wait till inputs available in the VP-Host channel
Update the SoC-level test bench with new inputs
Invoke HDL simulator
Carry the inputs to the AES IP
Carry the inputs to the DES3 IP
Compute the cipher text
Take the cipher text and make it available to the SoC output port
Wait till simulation completes
Receive output at the end of simulation
Wait till output available in the VP-Host channel
Read the cipher text
<Cipher text>
(a)
(b)

Figure 6. Crypto Case Study with VirSoC. (a) Crypto services provided by AES. (b) Crypto services provided by DES3.

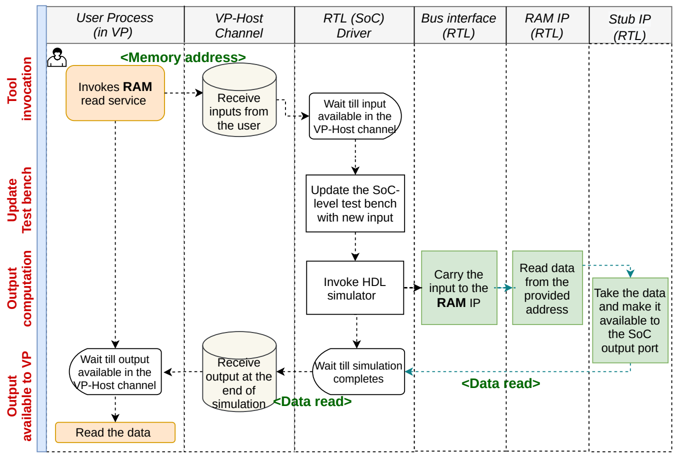

User Process (in VP)
VP-Host Channel
RTL (SoC) Driver
Bus interface (RTL)
RAM IP (RTL)
Stub IP (RTL)
Tool invocation
Update Test bench
Output computation
Output available to VP
<Memory address>
Invokes RAM read service
Receive inputs from the user
Wait till input available in the VP-Host channel
Update the SoC-level test bench with new input
Invoke HDL simulator
Carry the input to the RAM IP
Read data from the provided address
Take the data and make it available to the SoC output port
Wait till simulation completes
Receive output at the end of simulation
Wait till output available in the VP-Host channel
Read the data
<Data read>

Figure 7. Operation Flow for Memory Read in Processor-Memory Co-validation.

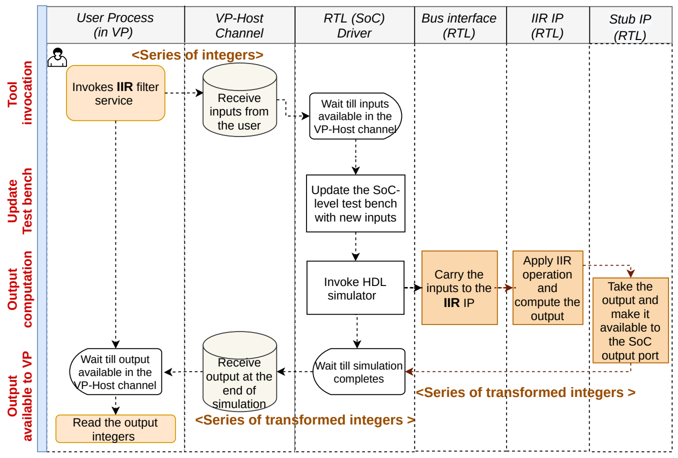
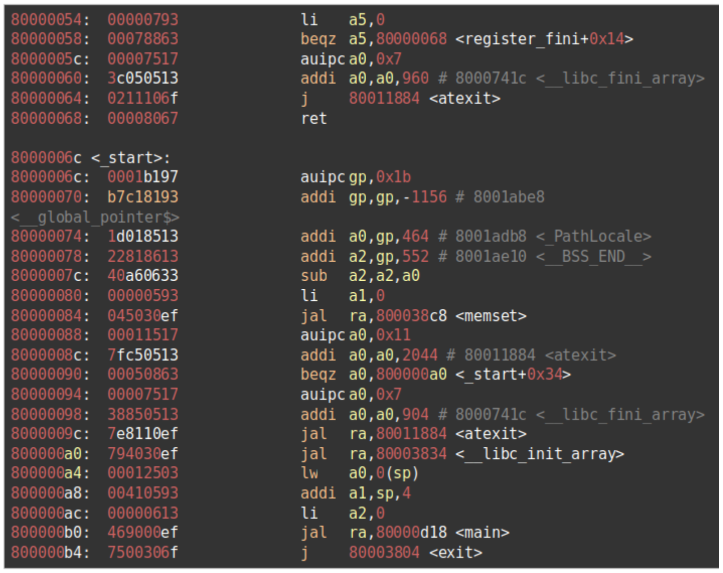

User Process (in VP)
VP-Host Channel
RTL (SoC) Driver
Bus interface (RTL)
IIR IP (RTL)
Stub IP (RTL)
Tool invocation
Update Test bench
Output computation
Output available to VP
<Series of integers>
Invokes IIR filter service
Receive inputs from the user
Wait till inputs available in the VP-Host channel
Update the SoC-level test bench with new inputs
Invoke HDL simulator
Carry the inputs to the IIR IP
Apply IIR operation and compute the output
Take the output and make it available to the SoC output port
Wait till simulation completes
Receive output at the end of simulation
Wait till output available in the VP-Host channel
Read the output integers
<Series of transformed integers >

Figure 8. Use case demonstration exercising IIR IP.

is designed to exploit this standardization for automating the integration of IPs into a single unified environment by hiding the heterogeneity of their underlying implementations and abstractions. For legacy IPs where the testbenches have been designed in an ad hoc manner, it may be a significant effort to retrofit them to work with VirSoC. However, as explained in Section III, this standardization is already happening in current industrial practice. VirSoC exploits this trend by repurposing the already developed standardized testbenches for automatic generation of hybrid VP.

B. *Execution Performance of* VirSoC*-generated Platforms*

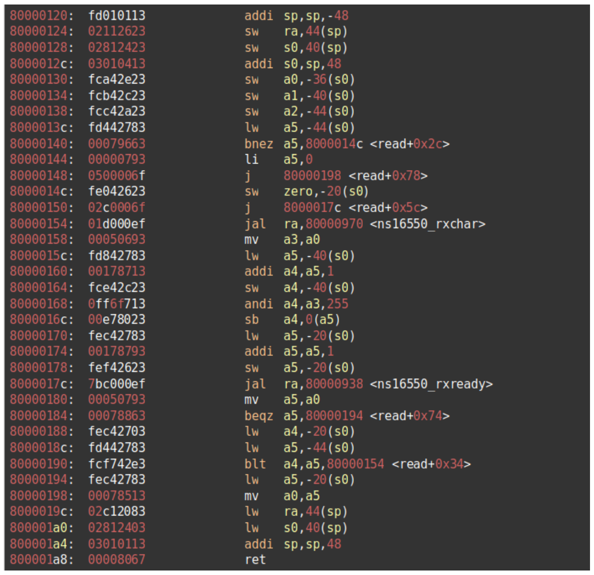
A key motivation for a hybrid virtual platform is to enable fast execution of use cases involving hardware-software interaction early in the development (when the silicon module for various components or even FPGA implementations are unavailable). In particular, typical handcrafted hybrid virtual platforms execute about 10^6 times faster than RTL simulations [11], [8], [12]. Since the VirSoC platform is automatically



```
80000054:  00000793             li    a5,0
80000058:  00078863             beqz  a5,80000068 <register_fini+0x14>
8000005c:  00007517             auipc a0,0x7
80000060:  3c050513             addi  a0,a0,960 # 8000741c <__libc_fini_array>
80000064:  0211106f             j     80011884 <atexit>
80000068:  00008067             ret

8000006c <_start>:
8000006c:  0001b197             auipc gp,0x1b
80000070:  b7c18193             addi  gp,gp,-1156 # 8001abe8
<__global_pointer$>
80000074:  1d018513             addi  a0,gp,464 # 8001adb8 <_PathLocale>
80000078:  22818613             addi  a2,gp,552 # 8001ae10 <__BSS_END__>
8000007c:  40a60633             sub   a2,a2,a0
80000080:  00000593             li    a1,0
80000084:  045030ef             jal   ra,800038c8 <memset>
80000088:  00011517             auipc a0,0x11
8000008c:  7fc50513             addi  a0,a0,2044 # 80011884 <atexit>
80000090:  00050863             beqz  a0,800000a0 <_start+0x34>
80000094:  00007517             auipc a0,0x7
80000098:  38850513             addi  a0,a0,904 # 8000741c <__libc_fini_array>
8000009c:  7e8110ef             jal   ra,80011884 <atexit>
800000a0:  794030ef             jal   ra,80003834 <__libc_init_array>
800000a4:  00012503             lw    a0,0(sp)
800000a8:  00410593             addi  a1,sp,4
800000ac:  00000613             li    a2,0
800000b0:  469000ef             jal   ra,80000d18 <main>
800000b4:  7500306f             j     80003804 <exit>
```

- Total Execution Time: 38 seconds
- Instruction Count: 25
- Average execution time = 1.52 seconds/instruction



```
80000120:  fd010113             addi  sp,sp,-48
80000124:  02112623             sw    ra,44(sp)
80000128:  02812423             sw    s0,40(sp)
8000012c:  03010413             addi  s0,sp,48
80000130:  fca42e23             sw    a0,-36(s0)
80000134:  fcb42c23             sw    a1,-40(s0)
80000138:  fcc42a23             sw    a2,-44(s0)
8000013c:  fd442783             lw    a5,-44(s0)
80000140:  00079663             bnez  a5,8000014c <read+0x2c>
80000144:  00000793             li    a5,0
80000148:  0500006f             j     80000198 <read+0x78>
8000014c:  fe042623             sw    zero,-20(s0)
80000150:  02c0006f             j     8000017c <read+0x5c>
80000154:  01d000ef             jal   ra,80000970 <ns16550_rxchar>
80000158:  00050693             mv    a3,a0
8000015c:  fd842783             lw    a5,-40(s0)
80000160:  00178713             addi  a4,a5,1
80000164:  fce42c23             sw    a4,-40(s0)
80000168:  0ff6f713             andi  a4,a3,255
8000016c:  00e78023             sb    a4,0(a5)
80000170:  fec42783             lw    a5,-20(s0)
80000174:  00178793             addi  a5,a5,1
80000178:  fef42623             sw    a5,-20(s0)
8000017c:  7bc000ef             jal   ra,80000938 <ns16550_rxready>
80000180:  00050793             mv    a5,a0
80000184:  00078863             beqz  a5,80000194 <read+0x74>
80000188:  fec42703             lw    a4,-20(s0)
8000018c:  fd442783             lw    a5,-44(s0)
80000190:  fcf742e3             blt   a4,a5,80000154 <read+0x34>
80000194:  fec42783             lw    a5,-20(s0)
80000198:  00078513             mv    a0,a5
8000019c:  02c12083             lw    ra,44(sp)
800001a0:  02812403             lw    s0,40(sp)
800001a4:  03010113             addi  sp,sp,48
800001a8:  00008067             ret
```

- Total Execution Time: 41 seconds
- Instruction Count: 35
- Average execution time = 1.17 seconds/instruction

Figure 9. Estimation of Average Instruction execution time on RISC-V.

generated, it is worthwhile to determine if it also performs a corresponding speedup. Here, we explain the experiments we did to enable back-of-the-envelope calculations to demonstrate that the execution speed of VirSoC is consistent with expectations and that, indeed, it provides a viable approach for hardware-software co-validation.

To examine this question, we instantiated VirSoC with QEMU-x86 running Ubuntu 16.04 as guest, constituting the VP and SystemVerilog modules for the RTL components (exercised by Verilator simulator) on a Dell server running Ubuntu 18.04 as host. Given this setup, the goal of the performance comparison is to see the speedup over a conventional RTL implementation. Obviously, an RTL implementation is expected to be slower than *any* hybrid VP; however, the VP created through VirSoC involves communication among multiple heterogeneous components, including many in RTL and exercised through simulation, as well as a number of scripts handling various software handoffs. Consequently, it is not *a priori* obvious that the speedup from VirSoC would be substantial.

There are two key practical challenges with performing performance evaluation for a hybrid VP. First, since RTL simulation can take a long time, it is impossible to do an actual time calculation for the entire hardware-software use case; consequently, we resort to an estimate using the calculation shown below. Furthermore, realistic RTL models of X86 processor are not directly available for integration with IPs to form a complete RTL simulation environment. (Indeed, that is in part why a hybrid VP is often used to exercise hardware-software use cases.) To address this deficiency, we use a RISC-V processor as a proxy for the CPU IP in RTL for comparison purposes. Note that RISC-V is a much simpler architecture than X86. Consequently, the projected simulation time reported is lower than what would happen if it were an X86 processor. *However, this implies the relative speed of the VP vis-a-vis RTL would be even higher if compared with an RTL platform with X86-based processors.*

Calculating Projected Simulation Speed for pure RTL model: To estimate the projected simulation speed, we had to estimate the time that would be taken if the VP model (in this case CPU) was instead replaced by RTL. Obviously, such a processor model would need to execute all the instructions that are currently executed in VP. To estimate the number of instructions involved, we used the QEMU TCG plugin [42]. Based on this data, the estimated number of baseline instructions involved is 893365348 *simply to boot the system and initialize the software use case* as necessary for exploring *any* hardware-software interactions. Given that this includes instructions of different types with variations in running time on simulation, we choose different (random) instruction subsequences from this set to estimate the average running time per instruction. Figure 9 shows the result of this experiment. The conservative estimated simulation time per instruction on RTL as a result of this experiment is $\sim$1second per instruction.[4]

Table IV summarizes the results. Note that virtual SoC environments execute within a few seconds, while it would take years to exercise the same use cases in RTL simulation. While the RTL simulation estimate is not surprising, the demonstration does show that hybrid VP is crucial to our ability to perform early Hardware-software co-validation, and

[4]This estimate is consistent with other folkloric estimates on typical simulation times, which estimate simulation time for one instruction to be about a second in RTL for a cycle-based simulator if the targeted silicon speed is 1GHz [11], [8].

Table IV
Comparing Simulation Speed for VirSoC with RTL Platform

Use cases	Simulation Speed in VirSoC generated platform (second)						Estimated RTL Simulation Speed (hour)
	Exp 1	Exp 2	Exp 3	Exp 4	Exp 5	Average execution time	
VP Firmware uses RTL AES IP during secure boot for derivation of crypto keys	7.359	7.229	7.372	7.910	8.113	7.596	28 years (projected)
VP Firmware uses RTL DES3 IP during secure boot for derivation of crypto keys	7.681	7.277	7.240	8.343	8.655	7.839	
VP Firmware uses RTL IIR IP to perform filtering operation	7.1631	7.233	7.453	7.882	8.220	7.590	
VP Firmware uses RTL RAM IP to perform certain memory write and read operation	7.194	8.317	7.203	7.56	8.39	7.732	

automatically generated virtual environments from VirSoC indeed is a viable platform for exploration of such use cases.

Remark 3: One may criticize the above calculations as too simplistic. However, note that the goal of the experiments is not to compare hybrid VP performance with RTL simulation. Indeed, the relative speed of VP over RTL is well-understood and is the primary reason why VP is used for early hardware-software co-validation in practice; establishing this speedup is *not* a contribution of this paper. The only point to establish is whether the automatically generated hybrid VPs from VirSoC perform adequately in practice with execution speedup similar to those of hand-crafted VP over RTL simulation so that they can be treated as viable platforms for hardware-software co-validation. Our back-of-the-envelope calculation establishes this viability. Furthermore, it is also well-known that the speedup would be less if we use IPs in FPGA as a point of comparison instead of RTL simulation (and negligible or even slower if the IP is in silicon). However, FPGA requires mature RTL, and silicon implementations are only available after initial fabrication. Consequently, RTL simulation seems to be a better comparison point to establish the viability of VirSoC for *early* hardware-software co-validation as required for a VP solution.

VII. Conclusion

Early exploration of hardware-software interaction is a crucial requirement for SoC validation. In this paper, we addressed this problem through a new infrastructure, VirSoC, that can automatically generate hybrid virtual platform environments usable for exploring such interactions. This permits the validator to exploit the high simulation speed of VP models while integrating them with the more accurate and refined RTL models to target various use cases. We showed how to use available collateral such as testbenches and stub modules to enable automated integration of components at different levels of abstraction and create a uniform virtual SoC environment. We demonstrated the use of such environments in several validation use cases. We also provided performance comparisons with pure RTL simulation to demonstrate the viability of the generated platforms.

In future work, we will extend the infrastructure to enable more automation in interface generation. We will also explore the use of the platform on other hardware-software use cases.

Acknowledgments: This research has been partially supported by the Semiconductor Research Corporation under Contract No. 19-CADT-2868 and by the National Science Foundation under Grant No. SATC-2223045. We thank Debjit Pal and Kshitij Raj for help with the implementation and evaluation of VirSoC.

References

[1] F. Ghenassia, *Transaction level modeling with SystemC*. Springer, 2005.
[2] "Synopsys Virtualizer," https://www.synopsys.com/verification/virtual-prototyping/virtualizer.html.
[3] "Cadence Helium," https://www.cadence.com/en_US/home/tools/system-design-and-verification/helium-virtual-and-hybrid-studio.html.
[4] "Siemnens Vista," https://eda.sw.siemens.com/en-US/ic/vista-virtual-prototyping/.
[5] "Simics Virtualizer," https://www.windriver.com/products/simics.
[6] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50–58, 2002.
[7] C. woo Lee, "QEMU," 2018, https://www.qemu.org/documentation/.
[8] Y. Abarbanel, E. Singerman, and M. Y. Vardi, "Validation of soc firmware-hardware flows: Challenges and solution directions," in *Proceedings of the 51st Annual Design Automation Conference*, 2014, pp. 1–4.
[9] W. Kim, H. Park, H. Kim, S. B. Choi, and S. Kim, "Early software development and verification methodology using hybrid emulation platform," *language*, vol. 1, p. 2, 2017.
[10] S. Yerramili, "Addressing Post-silicon Validation Challenge: Leverage Validation and Test Synergy," in *International Test Conference (ITC 2006)*, 2006.
[11] P. Mishra, R. Morad, A. Ziv, and S. Ray, "Post-silicon validation in the soc era: A tutorial introduction," *IEEE Design & Test*, vol. 34, no. 3, pp. 68–92, 2017.
[12] A. Sinha and S. Ray, "From Test to Post-silicon Validation: Concepts and Recent Trends," in *International Test Symposium*, 2017.
[13] W. Chen, S. Ray, J. Bhadra, M. Abadir, and L.-C. Wang, "Challenges and trends in modern soc design verification," *IEEE Design Test*, vol. 34, no. 5, pp. 7–22, 2017.
[14] 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 *2013 Formal Methods in Computer-Aided Design*, 2013, pp. 121–128.
[15] K. Cong, F. Xie, and L. Lei, "Automatic concolic test generation with virtual prototypes for post-silicon validation," in *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2013, pp. 303–310.
[16] L. Lei, F. Xie, and K. Cong, "Post-silicon conformance checking with virtual prototypes," in *Proceedings of the 50th Annual Design Automation Conference*, ser. DAC '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2463209.2488770
[17] P. Herber and S. Glesner, "A hw/sw co-verification framework for systemc," *ACM Trans. Embed. Comput. Syst.*, vol. 12, no. 1s, Mar. 2013. [Online]. Available: https://doi.org/10.1145/2435227.2435257

[18] B. Chen, C. Havlicek, Z. Yang, K. Cong, R. Kannavara, and F. Xie, "Crete: A versatile binary-level concolic testing framework," in *Fundamental Approaches to Software Engineering*, A. Russo and A. Schürr, Eds. Cham: Springer International Publishing, 2018, pp. 281–298.

[19] B. Chen, K. Cong, Z. Yang, Q. Wang, J. Wang, L. Lei, and F. Xie, "End-to-end concolic testing for hardware/software co-validation," in *2019 IEEE International Conference on Embedded Software and Systems (ICESS)*. IEEE, 2019, pp. 1–8.

[20] B. Lin, K. Cong, Z. Yang, Z. Liao, T. Zhan, C. Havlicek, and F. Xie, "Concolic testing of systemc designs," in *2018 19th International Symposium on Quality Electronic Design (ISQED)*. IEEE, 2018, pp. 1–7.

[21] T. Alam, Z. Yang, B. Chen, N. Armour, and S. Ray, "Firver: Concolic testing for systematic validation of firmware binaries," in *ASP-DAC*, 2022.

[22] "Confirma," https://www.synopsys.com.

[23] "TAI Logic Module," https://www.s2ceda.com/en/.

[24] "Palladium," https://www.cadence.com/en_US/home/tools/system-design-and-verification/emulation-and-prototyping/palladium.html.

[25] "ZeBu," https://www.synopsys.com/verification/emulation.html.

[26] A. P. D. Nath, K. Raj, S. Bhunia, and S. Ray, "Soccom: Automated synthesis of system-on-chip architectures," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 30, no. 4, pp. 449–462, 2022.

[27] F. Bellard, "Qemu, a fast and portable dynamic translator, in proceedings of the annual conference on usenix annual technical conference," *Anaheim, CA*, pp. 41–41, 2005.

[28] C.-Y. Huang, Y.-F. Yin, C.-J. Hsu, T. B. Huang, and T.-M. Chang, "Soc hw/sw verification and validation," in *16th Asia and South Pacific Design Automation Conference (ASP-DAC 2011)*, 2011, pp. 297–300.

[29] K. Kang, S. Park, B. Bae, J. Choi, S. Lee, B. Lee, and J.-B. Lee, "Seamless soc verification using virtual platforms: An industrial case study," in *2019 Design, Automation & Test in Europe Conference & Exhibition(DATE)*, 2019, pp. 1204–1205.

[30] J. Choi, K. Kang, B. Lee, S. Park, and J. Im, "Early hw/sw co-verification using virtual platforms," in *2021 18th International SoC Design Conference (ISOCC)*, 2021, pp. 1–2.

[31] "Open Virtual Platforms," https://www.ovpworld.org/.

[32] A. Wicaksana, A. Charif, C. Andriamisaina, and N. Ventroux, "Hybrid prototyping methodology for rapid system validation in hw/sw co-design," in *2019 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, 2019, pp. 35–40.

[33] L. Masing, F. Lesniak, and J. Becker, "A hybrid prototyping framework in a virtual platform centered design and verification flow," *IEEE Embedded Systems Letters*, vol. 13, no. 1, pp. 1–4, 2021.

[34] M. R. Kabir, N. Mishra, and S. Ray, "Vive: Virtualization of vehicular electronics for system-level exploration," in *2021 IEEE International Intelligent Transportation Systems Conference (ITSC)*, 2021, pp. 3307–3312.

[35] A. Charif, G. Busnot, R. Mameesh, T. Sassolas, and N. Ventroux, "Fast virtual prototyping for embedded computing systems design and exploration," in *Proceedings of the Rapid Simulation and Performance Evaluation: Methods and Tools*, 2019, pp. 1–8.

[36] F. Cucchetto, A. Lonardi, and G. Pravadelli, "A common architecture for co-simulation of systemc models in qemu and ovp virtual platforms," in *2014 22nd International Conference on Very Large Scale Integration (VLSI-SoC)*, 2014, pp. 1–6.

[37] X. Bian, "Implement a virtual development platform based on qemu," in *2017 International Conference on Green Informatics (ICGI)*. IEEE, 2017, pp. 93–97.

[38] G. Delbergue, M. Burton, F. Konrad, B. Le Gal, and C. Jego, "Qbox: an industrial solution for virtual platform simulation using qemu and systemc tlm-2.0," in *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, 2016.

[39] T.-C. Yeh and M.-C. Chiang, "On the interfacing between qemu and systemc for virtual platform construction: Using dma as a case," *Journal of Systems Architecture*, vol. 58, no. 3-4, pp. 99–111, 2012.

[40] P. Dovgalyuk, N. Fursova, I. Vasiliev, and V. Makarov, "Qemu-based framework for non-intrusive virtual machine instrumentation and introspection," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 944–948.

[41] K. A. Meade and S. Rosenberg, *A practical guide to adopting the universal verification methodology (UVM)*. Cadence Design Systems, 2010.

[42] "QEMU TCG Plugins," https://qemu.readthedocs.io/en/latest/devel/tcg-plugins.html.

Tashfia Alam is a Ph.D. candidate in the department of Electrical and Computer Engineering (ECE) at the University of Florida. She received her Bachelor's degree from Bangladesh University of Engineering and Technology in Electrical and Electronic Engineering. After completing her undergraduate studies, Tashfia gained industry experience at a VLSI startup, focusing on device modeling and analog IP design. She also holds a great interest in digital design, computer architecture and formal validation which eventually became her research interest. Tashfia's current research focuses on developing hardware-software co-validation techniques leveraging formal methods for trustworthy System-on-Chip design.

Indira Bhoomareddy Ramaiah holds a Bachelor's Degree in Electronics and Communication Engineering from Visvesvaraya Technological University in India, followed by a Master's Degree in Computer Engineering from the University of Florida, Gainesville. Indira's research interests range from improving Power, Performance and Area of System on Chips to Hardware-Software Co-Validation. Indira has gained industry experience as a Synthesis and Implementation Engineer at Qualcomm Technologies Inc., San Diego, where she contributed to the development of cutting-edge CPUs. Currently, Indira serves as a Lead Application Engineer for the Synthesis and Implementation team at Cadence Design Systems, San Jose.

Sandip Ray (SM'13) received the Ph.D. degree from the University of Texas at Austin. He is a Professor at the Warren B. Nelms Institute for the Connected World affiliated with the Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL, USA. Prior to that, he worked with NXP Semiconductors and Intel Strategic CAD Laboratories where he led industrial research and R&D projects in pre-silicon and post-silicon validation of security and functional correctness of SoC designs, design-for-security and design-for-debug architectures, and security validation for automotive and the Internet-of-Things applications. His current research targets correct, dependable, secure, and trustworthy computing. He is the Author of three books and over 100 publications in international journals and conferences. He has also served as a TPC Member of over 50 international conferences and as Guest Editor for several journals.