IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 18 November 2024, accepted 9 January 2025, date of publication 14 January 2025, date of current version 22 January 2025.

Digital Object Identifier 10.1109/ACCESS.2025.3529667

RESEARCH ARTICLE

Automatic Generation of Simulators for Processors Enhanced for Security in Virtualization

SWAPNEEL C. MHATRE AND PRIYA CHANDRAN, (Senior Member, IEEE)
Department of Computer Science and Engineering, National Institute of Technology Calicut, Kozhikode, Kerala 673601, India

Corresponding author: Swapneel C. Mhatre (swapneelcoep@gmail.com)

This work was supported in part by Anusandhan National Research Foundation (erstwhile Science and Engineering Research Board), New Delhi, in 2021, under Grant CRG/2020/003584.

ABSTRACT All new computer architectures need to be performance evaluated for acceptance and simulation is the most widely used method for evaluation of new processor designs. Sharing resources with virtualization raised many security concerns, leading to the development of processors that are enhanced for security in virtualization. The simulators for processors enhanced for security in virtualization need to perform simulation of hypervisor instructions, simulation of security in virtualization, and simulation of new instructions. However, a simulator with all of these three features is not found in the literature. Hence, this paper proposes an approach for the simulation of processors enhanced for security in virtualization that provides all these three features. For user convenience, the simulators are generated automatically from the target processor specifications using a simulator generator. The paper also proposes an approach for simulating a new pipeline with a designer-specified number of stages with automatic detection of pipeline hazards and automatic stalling or flushing of the pipeline on detection of hazards. To demonstrate the use of the simulator generator and the generated simulator, three case studies are considered - simulation of RISC-V with HyperWall, simulation of RISC-V with bit-serial dot-product unit, and simulation of RISC-V with Galois Field arithmetic extension. The paper concludes that the proposed approaches help in accurately simulating the overhead due to security in virtualization and also in providing flexibility to the designer to simulate the desired processor configurations.

INDEX TERMS Compiler, computer architecture, hypervisor, operating system, security in virtualization, simulation.

I. INTRODUCTION

All new computer architectures need to be performance evaluated for acceptance. Potential evaluation methods to assess whether a particular idea improves a particular target metric are as follows:

1) Theoretical proof
2) Analytical modeling/estimation
3) Simulation (at varying degrees of abstraction and accuracy)
4) Prototyping with a real system (e.g., field programmable gate arrays)
5) Real implementation

Out of these methods, simulation is the most widely used method for the evaluation of new processor designs. But simulation is a challenging task, especially, for new types of architectures such as *HyperWall* [1], [2] and *Architectural Support for Memory Isolation (ASMI)* [3], that provide security in virtualization.

Virtualization facilitates resource sharing in many domains, and cloud computing is the hottest example. In system virtualization, an additional software layer called the hypervisor (HV), runs below the operating system (OS).

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

When a system call (call to the operating system) is made, the control is transferred to the operating system. In a virtualized system, the OS may make a hypercall (call to the hypervisor) or may trap to the hypervisor in which case the control is transferred to the hypervisor. Hence, it becomes necessary to simulate the hypervisor instructions as well for accurate timing simulation of virtualized systems. Applications that require accurate simulation of the overhead due to virtualization include distributed systems, storage, virtual memory, and so on [4]. However, most of the present-day simulators for virtualized systems abstract the hypercalls, or traps to the hypervisor code from the OS code, i.e., they emulate the hypervisor instructions as a single instruction set architecture (ISA) instruction. Hence, they cannot perform accurate timing simulation of virtualized systems.

Furthermore, in many modern processors, there are modifications done at the architectural and microarchitectural levels to provide security in virtualization. Such processors require additional clock cycles while accessing memory to perform an access check. However, most of the present-day simulators do not simulate such security-related enhancements of the processor. Hence, there is a need to develop simulators that simulate such virtualization-enhanced processors, especially, those with security-related enhancements [5].

Moreover, in hardware-assisted virtualization, additional rings are added so that the hypervisor can be at a higher privilege level than the operating system without evicting the OS from the ring where it is intended to be. Since processors supporting hardware-assisted virtualization provide an additional mode of operation, there are additional instructions in the ISA to enter and exit from the additional mode. In virtualization-enabled processor designs, instruction set modifications have been done at different privilege levels and need to be simulated.

Thus, the simulators for processors enhanced for security in virtualization need to perform:

1) Simulation of hypervisor instructions
2) Simulation of security in virtualization
3) Simulation of new instructions

Hence, this paper proposes an approach for the automatic generation of simulators for processors enhanced for security in virtualization that provides all of these features. Automatic generation of simulator provides user convenience by using a simulator generator to automatically generate simulators from the processor specifications written in an *Architecture Description Language (ADL)*, which is a *domain specific language* [6]. A new ADL for processor specifications is developed which can be read by the proposed simulator generator to generate the desired simulator.

To simulate hypervisor instructions, the simulator jumps to the hypervisor code on hypercalls.

To simulate security in virtualization, the ADL provides the facility to specify the protection table that contains protection encodings of the pages as specified by the customer. The customer also specifies the protection encodings for which the hypervisor is allowed to access the memory. If access check is enabled, the additional clock cycles required to access the protection table to perform memory access checks are simulated.

To simulate new instructions, the ADL provides the facility not only to specify the format of the instructions but also to specify what every instruction does in each stage of the pipeline.

In addition to the three features needed in the simulators for virtualized systems, viz., simulation of hypervisor instructions, simulation of security in virtualization, and simulation of new instructions, the generated simulator can also simulate a new pipeline with a designer-specified number of stages. The paper proposes an approach for automatic detection of pipeline hazards and automatic stalling or flushing of pipelines on detection of hazards.

The contributions of the paper are:

1) An approach for the simulation of security in virtualization
2) An approach for the simulation of a pipeline with a designer-specified number of stages with automatic detection of pipeline hazards and automatic stalling or flushing of the pipeline on detection of hazards

Section II gives the background of the automatic generation of architecture simulators and the capabilities of various simulators for virtualized systems. Section III gives the overview of the proposed approach for the automatic generation of simulators for processors enhanced for security in virtualization. The approach includes the approaches for the simulation of hypervisor instructions, simulation of security in virtualization, and simulation of new instructions. The next three sections - section IV, section V, and section VI describe each of these three approaches. Section VII describes the proposed approach for the simulation of a new pipeline. To demonstrate the capability of the simulator generator and the generated simulator, three case studies are taken. Section VIII describes the three case studies of the simulation of different processors. Section IX gives the results obtained by simulating the binaries in each of the three case studies. Section X concludes that the proposed approaches help in accurately simulating the overhead due to security in virtualization and also in providing flexibility to the designer to simulate the desired processor configurations. Section XI gives the future scope.

II. BACKGROUND AND RELATED WORK

A. VIRTUALIZATION

A *virtual machine (VM)* is taken to be an *efficient, isolated duplicate* of the real machine [7]. It is the environment created by the *virtual machine monitor (VMM)* or *hypervisor* [7]. In a system virtual machine, the underlying hardware platform is called the *host*, and the OS and the applications that run above the host are collectively called the *guest*. Thus, the guests (guest OSs and guest applications) share the resources of the host (hardware) among themselves. This resource sharing is managed by the hypervisor.

According to the conditions laid down by Popek and Goldberg [7], for a processor to be virtualizable, every *sensitive instruction* (instruction that changes the privileged state or exposes the privileged state) should be a *privileged instruction* (instruction that executes in system mode but traps to the kernel when executed in user mode). In other words, the ISA should be designed in such a way that if the OS tries to execute any instruction that either changes the privileged state or exposes the privileged state, it should trap to the hypervisor. An architecture that is virtualizable as per the conditions laid down by Popek and Goldberg is called an *efficiently virtualizable architecture* or a *fully virtualizable architecture*. However, many Complex Instruction Set Computing (CISC) architectures like x86 and many Reduced Instruction Set Computing (RISC) architectures like Advanced RISC Machines (ARM) are not fully virtualizable. On the other hand, the RISC architecture RISC-V [8], [9], [10], [11], [12] is fully virtualizable.

In virtualization, any operating system can be run inside a virtual machine. However, running a complete operating system including the kernel requires a large amount of system resources. Hence, a new lightweight technology, called *containerization* [13], [14], [15], has been developed as an alternative to virtualization. In containerization, only the executables and their dependencies run inside a container. Hence, the container uses less system resources than the virtual machine.

Virtual machines are subject to various attacks like covert channel attacks, malware attacks, and attacks in migration [16], [17]. Hence, modern architectures are enhanced to provide security in virtualization.

B. SECURITY IN VIRTUALIZATION

Modern architectures remove the hypervisor from the trusted computing base (TCB) and protect the guest VMs from even the hypervisor itself.

For example, *HyperWall* [1], [2] uses Confidentiality and Integrity Protection (CIP) tables to provide security to VMs. *Extended-HyperWall* [18], [19] integrates the CIP table and Rollback-Sensitive Data Memory (RSDM) table to prevent rollback-based attacks. *ASMI* [3] provides memory isolation and hence security to each virtual machine by making modifications to the central processing unit (CPU), memory management unit (MMU), and ISA in Intel Virtualization Technology. Intel has introduced *Intel Software Guard Extensions (SGX)* [20] and *Intel Trust Domain Extensions (TDX)* [21] for security in virtualization. Similarly, to provide security in virtualization, AMD has also introduced *Secure Encrypted Virtualization (SEV)* [22].

Such architectures need to be simulated to evaluate their performance.

C. SIMULATION

Simulation is imitating the system being modeled to the desired level of detail in the desired time [23]. An *emulator* is a simulator that performs only functional simulation of the ISA without regard to how the instructions are executed. As against this, a *simulator* may perform functional simulation or timing simulation or both and a simulator that performs timing simulation considers how the instructions are executed to determine the timing and timing-dependent behaviour of the architecture.

Simulation of computer architecture may be done using any of the following methods:

1) Constructing a simulator (Using a design-specific simulator)
2) Modifying a simulator (Using a customized simulator)
3) Automatic generation of simulator

Constructing or modifying a simulator is time-consuming and error-prone [24]. Although it is faster and easier to modify an existing simulator than to construct a new simulator, it is still becoming increasingly time-consuming and error-prone due to the increasing complexity of processors. Furthermore, although many simulators are available in literature [25], sometimes, it may not be possible to simulate the proposed architecture by modifying any of the existing simulators. For example, *GPGPU-Sim* was developed because none of the existing simulators for graphics processing unit (GPU) architectures could model the newly designed general-purpose GPU architecture [26]. Hence, often automatic generation of the simulator is used in which a simulator is generated automatically from a machine specification written in a *domain specific language*. A domain-specific language for describing the computer architecture is called an *ADL*. Thus, an ADL provides an abstraction layer for describing the computer architecture and automatically generating corresponding simulators [24].

A simulator generator can use the ADL program to generate a simulator using either *interpretive simulation* or *compiled simulation* [27]. *Interpretive simulation* is a technique in which instruction decoding is done at run time, while *compiled simulation* is a technique in which instruction decoding is done at compile time. The differences between interpretive simulation and compiled simulation are given in Table 1.

Some simulator generators like *Facile compiler* [28] generate simulators from the descriptions of the simulators and not from the descriptions of the target processors.

The system for automatic generation of architecture simulators from target processor description using interpretive simulation is shown in Fig. 1.

The target processor descriptions consist of two parts, viz., *declarative* and *imperative* [6].

The *declarative part* of the processor description consists of the part whose actions are implicit in the declaration. This includes *artifacts*, which are hardware objects with well-established operational semantics [29]. The simulator code for the declarative part is fixed and is hence, hard-coded in the simulator generator. Examples of declarative parts are registers, register files, caches, etc. Since the operations are fixed, the simulator generator only needs the parameters

TABLE 1. Differences between interpretive simulation and compiled simulation.

Sr. No.	Interpretive Simulation	Sr. No.	Compiled Simulation
1.	Instruction decoding is done at run time.	1.	Instruction decoding is done at compile time.
2.	The simulator generated is not fixed for any specific target program executable.	2.	The simulator generated is fixed for the specific target program executable.
3.	The target program executable is not given as input to the automatic simulator generator but is given as input to the host machine when the simulator runs.	3.	The target program executable is given as input to the automatic simulator generator itself and not to the host machine when the simulator runs.

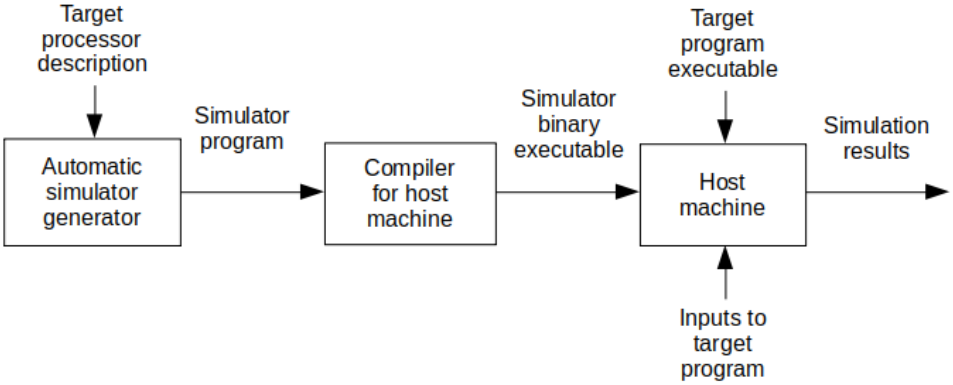

FIGURE 1. System for automatic generation of architecture simulators from target processor description using interpretive simulation.

(e.g., length of the registers, number of registers in the register files, size of the caches, etc.) to generate the corresponding simulator code.

The *imperative part* of the processor description consists of the part whose actions are to be explicitly specified. This includes hardware objects whose operational semantics are not generalizable but need to be specified using a procedural code. The simulator code for the imperative part is generated from the procedural code, written in a *Register Transfer Language (RTL)*, using an *RTL compiler* (consisting of *lexer* and *parser*) in the simulator generator. Examples of imperative parts are the different pipeline stages. Since the operations performed in the different pipeline stages of different processors vary greatly (e.g., the operations of execute, memory access, and write back may be performed in a single stage as in ARM7 or may be performed in three different stages as in MIPS), the simulator generator needs the description of the stages to generate the corresponding simulator code.

Simulator generators may be classified into two categories depending on whether they generate *instruction set simulators* or *microarchitecture simulators*.

Instruction set simulator generators automatically generate *instruction set simulators* or *functional simulators* from ADL programs that consist of the specification of the ISA. *SimGen* [30], *Markus* and *CGEN (CPU tools GENerator)* are examples of instruction set simulator generators. SimGen uses a procedural description language, with user-written C routines using user-defined macros and event handlers, Markus uses an ADL called *nML* [31], while CGEN uses *Scheme*, a dialect of *LISP (list processing)*.

Microarchitecture simulator generators automatically generate *microarchitecture simulators* or *timing simulators* from ADL programs that consist of the specification of the ISA as well as the organization of the components of the microarchitecture. *University of Pittsburgh Flexible Architecture Simulation Tool (UPFAST)* [29], *Automatic Generation of Usable Simulation Tools (August)* [6] and *Computer Architecture Description Language (CADL) compiler* [24] are examples of microarchitecture simulator generators.

The simulators generated by *SimGen* [30], *Markus*, *CGEN*, *UPFAST* [29], *August* [6] and *CADL compiler* [24] neither emulate nor simulate hypervisor instructions and hence, they cannot simulate virtualized systems.

The simulators for virtualized systems may use any of the following approaches:

1) *Multikernel simulation* [32] is an interesting approach to simulate rollback-sensitive memory architecture. It simulates the hardware modifications by modifying the kernels at different privilege levels. It modifies the hypervisor and the guest OS and if required, even the guest application program. However, multikernel simulation performs only functional simulation and not timing simulation.
2) Another interesting approach is to modify the hypervisor to extract the information required for simulation and transfer it to the simulator. For example, *Xen* [33] may be modified to detect and count hypercalls [34] and the structures used by the processor may be simulated in hardware to determine the number of memory accesses [35]. However, this requires modifications to be done to the hypervisor source code.

3) Yet another approach is to store the application program binary code, the OS binary code, and the hypervisor binary code in the code areas of the corresponding portions of the memory and to jump to the OS code on system calls and to the hypervisor code on hypercalls [36].

In order to model the increased overhead due to CPU virtualization and memory virtualization in virtualized systems, the simulators for virtualized systems need to simulate hypervisor instructions. These simulators also need to model the increased overhead due to security in virtualization. Since new instructions may be added for virtualization and for providing security in virtualization, these simulators also need to simulate new instructions.

A comparison of the capabilities of various simulators for virtualized systems is given in Table 2.

The timing simulation of application program instructions and OS instructions is performed by many simulators. However, to simulate virtualized systems, the hypervisor instructions need to be emulated (for functional simulation) or simulated (for timing simulation).

The original version of *gem5* [45] neither emulates nor simulates hypervisor instructions and hence, it does not simulate virtualized systems. *gem5v* [40] emulates hypervisor instructions but does not simulate them, hence, it can simulate the functionality of virtualized systems but cannot model the increased overhead due to the execution of hypervisor instructions. Support for the simulation of single-core RISC-V [46], multi-core RISC-V [47] and RISC-V full system [4] has been added to gem5. The RISC-V full system version of gem5 [4] simulates hypervisor instructions. However, it simulates machine-mode (M-mode) hypervisor *Diosix* and does not simulate the hypervisor extension of RISC-V [12]. The simulator developed in [36] simulates the hypervisor instructions and also simulates the hypervisor extension of RISC-V. However, it is a *functional-first simulator* [23] in which the functional simulator generates the instruction information on the fly and updates the state of the program and the state of the hardware and a separate timing simulator accepts the instruction information generated by the functional simulator and generates the performance statistics. Being a functional-first simulator, it does not support wrong-path execution, i.e., it simulates flushing of the pipeline on branch misprediction, but the instruction flushed is the correct instruction. *VMcSim* [44] simulates only those hypervisor instructions that perform scheduling, hence, it can model the increased overhead in CPU virtualization but cannot model the increased overhead in memory virtualization in virtualized systems.

None of these simulators can model the increased overhead due to security in virtualization. For example, additional clock cycles required to access memory for performing memory access checks cannot be modeled.

Using simulators like *gem5*, the designer can only select between the ISAs supported by the simulator, but cannot simulate newly designed instructions. Also, the simulator developed in [36] supports only RISC-V ISA and cannot simulate new instructions. The simulators generated by *UPFAST*, *August* and *CADL compiler* can simulate new instructions by allowing the designer to specify the operation performed by every instruction using an ADL, but the simulators generated by these simulator generators do not simulate virtualized systems.

Hence, an approach for the automatic generation of simulators for processors enhanced for security in virtualization is proposed.

III. PROPOSED APPROACH FOR AUTOMATIC GENERATION OF SIMULATORS FOR PROCESSORS ENHANCED FOR SECURITY IN VIRTUALIZATION

The proposed approach for the automatic generation of simulators for processors enhanced for security in virtualization uses *integrated simulation* in which functional simulation and timing simulation are unified into a single code [23]. Since integrated simulators perform the functional and timing simulation of the instructions in every stage of the pipeline, they can simulate wrong-path execution, i.e., they can simulate flushing of the incorrect instructions in the pipeline on branch misprediction.

The proposed simulator generator generates simulators from target processor descriptions using *interpretive simulation* as shown in Fig. 1. The target processor descriptions are written in a newly proposed ADL. The proposed ADL is based on XML because XML is both human-readable and machine-readable. The simulator generator is written in Python and the generated simulators are also in Python.

The ADL specifies the declarative part of the processor description using XML elements and attributes. The imperative part of the processor description is represented by elements consisting of segments of an RTL. The segments contain RTL statements along with high-level functions for specifying architectural features and concepts.

The processor specifications in the ADL are divided into *architecture specifications* and *microarchitecture specifications*. The architecture specifications consist of the specifications of the *RISC ISA*, *registers*, and *two-stage address translation*. The microarchitecture specifications consist of the specifications of the *hardware mode*, *resources*, and *scalar sequential pipeline* [48]. All the architecture specifications are declarative and the corresponding simulator code is hard-coded in the simulator generator. In the microarchitecture specifications, the specifications of the mode and resources are declarative, while the specifications of the pipeline are imperative. Thus, the simulator code for the mode and the resources is hard-coded in the simulator generator, while the simulator code for every stage of the pipeline is generated from the procedural code, written in RTL, using an *RTL compiler*.

The RTL compiler converts the RTL code into Python code using *lexer* and *parser* as shown in Fig. 2. The lexer reads the input characters and produces tokens as output.

TABLE 2. Capabilities of various simulators for virtualized systems.

Simulator	Timing Simulation of Application Program Instructions	Timing Simulation of OS Instructions	Timing Simulation of Hypervisor Instructions	Modeling of the Increased Overhead in CPU Virtualization in Virtualized Systems	Modeling of the Increased Overhead in Memory Virtualization in Virtualized Systems	Modeling of the Increased Overhead due to Security in Virtualization
QEMU [37] (Full system emulation)	×	×	×	×	×	×
PVMsim [38]	✓	×	×	×	×	×
Virtual-GEMS [39]	✓	✓	×	×	×	×
gem5v [40]	✓	✓	×	×	×	×
RISC-V full system version of gem5 [4]	✓	✓	✓	✓	✓	×
Mambo [41]	✓	✓	×	×	×	×
PTLsim (Full system version) [42]	✓	✓	×	×	×	×
MARSS [43]	✓	✓	×	×	×	×
VMcSim [44]	✓	×	✓*	✓	×	×
Simulator developed in [36]	✓	✓**	✓***	×	✓	×
Simulator generated by the proposed simulator generator	✓	✓**	✓***	×	✓	✓

* Simulates only those hypervisor instructions that perform scheduling
** Simulates read, write, fstat and brk system calls
*** Simulates console_io and console file status hypercalls
QEMU: Quick EMUlator
MARSS: Micro ARchitectural and System Simulator

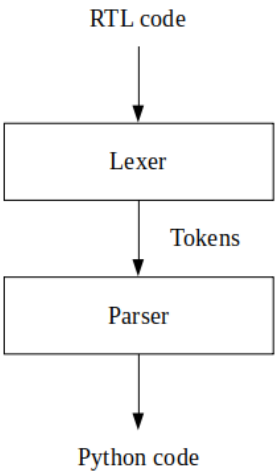

FIGURE 2. RTL compiler.

The parser accepts the tokens produced by the lexer. *Syntax-directed translation* is used to generate the Python code for the simulator. The RTL compiler is a *one-pass compiler* and is written using PLY (Python Lex-Yacc).

The approach for automatic generation of simulators for processors enhanced for security in virtualization includes approaches for the simulation of hypervisor instructions, simulation of security in virtualization, and simulation of new instructions.

IV. PROPOSED APPROACH FOR THE SIMULATION OF HYPERVISOR INSTRUCTIONS

In order to simulate OS and hypervisor instructions, the simulator supports different hardware modes - mode for user, mode for OS, and mode for HV. The application program binary code, the OS binary code, and the hypervisor binary code are stored in the code areas of the corresponding portions of the memory. Using the ADL, the designer can specify the operation to be performed in each stage of the pipeline during the execution of the instruction that invokes the system call and hypercall, depending upon the current hardware mode. For example, saving the mode and appropriate registers, including the program counter (pc), and switching to the appropriate mode may be specified. The simulator supports three types of target addresses - pc-relative target address, register-relative target address, and absolute target address - and the appropriate target address type may be specified for the instruction that invokes the system call and hypercall. Similarly, the operation to be performed during the execution of the trap-return instruction may be specified.

An approach for the simulation of security in virtualization is also proposed.

V. PROPOSED APPROACH FOR THE SIMULATION OF SECURITY IN VIRTUALIZATION

In processors enhanced for security in virtualization, the protection of the pages is dictated by the protection encodings in the protection table (e.g., CIP table in *HyperWall* [1], [2]) stored in hardware-only accessible memory. Corresponding to each page, there is a protection encoding in the protection table as specified by the customer. The customer also specifies the protection encodings for which the hypervisor is not allowed to access memory. Additional clock cycles are required to access the protection table to perform a memory access check. Algorithm 1 gives the proposed approach for the simulation of security in virtualization. Since the memory is being protected from malicious HV, if the HV is not running, memory access is allowed.

An approach for the simulation of new instructions is also proposed.

VI. PROPOSED APPROACH FOR THE SIMULATION OF NEW INSTRUCTIONS

The ADL provides the facility not only to specify the format of the instructions but also to specify what every instruction does in each stage of the pipeline. Hence, any new instruction may be specified. Using the ADL, the designer specifies the instruction type of every instruction, the resource required for every instruction type in each stage of the pipeline, and the latency of each resource. From this, the latency of every instruction in each stage of the pipeline is computed.

The ability to simulate new instructions is useful not only to simulate new instructions for virtualization and for providing security in virtualization but also to simulate any new instruction in general.

Algorithm 1 Algorithm for the Simulation of Security in Virtualization

```
Perform data translation lookaside buffer (D-TLB) lookup.
if D-TLB hits then
    Access memory.
else
    Perform address translation to convert virtual address (VA) of the data to physical address (PA).
    Check if hypervisor is running.
    if hypervisor is not running then
        Memory access is allowed.
    else
        Compute physical page number (PPN) by performing integer division of the physical address of the data and the page
        size in bytes.
        Get protection encoding for this page from the protection table stored in hardware-only accessible memory.
        if protection encoding of this page is in the set of protection encodings for which the hypervisor is not allowed to access
        memory then
            Memory access is not allowed.
        else
            Memory access is allowed.
        end if
    end if
    if memory access is allowed then
        Access memory.
    else
        Do not access memory.
    end if
end if
The corresponding performance statistics are updated at every step.
```

Many architectures have been developed with different standard extensions to the RISC-V ISA [49], [50]. For example, *Arrow* co-processor [49] implements a subset of the RISC-V 'V' vector extension version 0.9 [51] aimed at edge machine learning inference. Although RISC-V ISA is now supported by gem5 and many other simulators, there are many standard extensions to the RISC-V ISA [11], [52] and many of these are not supported by any simulator found in the literature.

Also, many architectures have been developed with customized extensions to the RISC-V ISA [53], [54]. For example, *NeuralScale* neural processor [53] implements customized vector extensions with fixed-width 32-bit instruction format in addition to the RISC-V 'V' vector extension version 0.8 [55].

Many architectures have also been developed with a completely new ISA [56]. For example, *Cambricon* [56] is a novel domain-specific ISA for neural network accelerators, which is a load-store architecture that integrates scalar, vector, matrix, logical, data transfer and control instructions, based on a comprehensive analysis of existing neural network techniques.

The proposed approach to simulate new instructions may be used to simulate such architectures with standard and customized extensions to the RISC-V ISA as well as a completely new ISA.

An approach for the simulation of a new pipeline is also proposed.

VII. PROPOSED APPROACH FOR THE SIMULATION OF A NEW PIPELINE

The proposed approach simulates a scalar sequential pipeline [48] with a designer-specified number of stages.

Every instruction in a pipeline is represented by its static and dynamic information. The static information includes the following:

1) Mnemonic
2) Source register 1 index (rs1)
3) Source register 2 index (rs2)
4) Destination register index (rd)
5) Immediate data
6) A bit 'register_read' which indicates whether the instruction performs register read
7) A bit 'register_write' which indicates whether the instruction performs register write
8) Instruction type which may be integer ALU, float ALU, unconditional jump, conditional branch, load, store, or no-operation
9) Branch condition which specifies the condition when the branch is taken

10) Target address type which may be pc-relative target address, register-relative target address, or absolute address
11) A bit 'environment_call' which indicates whether the instruction is an environment call
12) A bit 'trap_return' which indicates whether the instruction is a trap return

The dynamic information includes the following:

1) Source register 1 data (rs1_data)
2) Source register 2 data (rs2_data)
3) Destination register data (rd_data)
4) Instruction virtual address
5) Data virtual address
6) A bit 'taken_branch_or_unconditional_jump' which indicates whether the instruction is a taken branch or unconditional jump
7) Target address

The pipeline stages are simulated in reverse order so that at time t, stage i will operate on the output produced by stage (i-1) at time (t-1). When an instruction in a pipeline updates the state (software state and/or hardware state), the simulator updates the corresponding simulated state and also updates the corresponding dynamic information of the instruction. When an instruction advances in the pipeline, the corresponding information (static and dynamic information) is moved from the current stage to the next stage of the pipeline. To relieve the designer of the processor from the cumbersome task of explicitly specifying the conditions for pipeline hazards, an approach for automatic detection of pipeline hazards is proposed. Automatic detection and resolution of pipeline hazards is done for every pipeline stage i. This is done by first detecting structural hazards, data hazards, and control hazards and then using this information to find out whether the instruction will advance in the next clock cycle, progress in the same stage in the next clock cycle, or stall in the next clock cycle.

Algorithm 2 gives the proposed approach for the detection of a *structural hazard* to advance stage i in a scalar sequential pipeline.

Algorithm 3 gives the proposed approach for the detection of a *read-after-write (RAW) hazard* to advance stage i in a scalar sequential pipeline. The simulator simulates the detection of all the data hazards in an instruction decode (ID) stage, which is stage 2 of the pipeline. This approach is followed in the RISC-V integer pipeline because it reduces the hardware complexity by never requiring the hardware to suspend an instruction that has updated the state of the processor unless the entire processor is stalled [57]. Hence, if $i \neq 2$, then there is no RAW hazard to advance stage i.

Algorithm 4 gives the proposed approach for the detection of a *control hazard* in a scalar sequential pipeline.

Algorithm 5 gives the proposed approach for automatic detection and resolution of pipeline hazards in a scalar sequential pipeline. Since the simulator models a scalar sequential pipeline, write-after-read (WAR) hazards and write-after-write (WAW) hazards cannot occur. The only data hazards that may occur are RAW hazards. Hence, there is a data hazard to advance stage i if and only if there is a RAW hazard to advance stage i.

To demonstrate the use of the simulator generator and the generated simulator, three case studies are considered.

An ADL program is written so that the simulator generator generates the simulator with the desired configuration.

VIII. CASE STUDIES

Since the main motivation for developing the proposed simulator generator is to generate simulators to simulate processors enhanced for security in virtualization, the simulator generator is demonstrated by simulating an architecture that provides security in virtualization. *HyperWall* [1], [2] is one such architecture in which the protection of the pages is dictated by the protection encodings in the CIP table stored in hardware-only accessible memory.

Over the past decade, there has been a revolution in terms of the applications that can benefit from near-sensor intelligent data analytics which has led to the growing importance of *tinyML (Tiny Machine Learning)* [58], [59], [60], [61] that deals with machine intelligence at the very edge of the cloud using ultra-low-power *machine learning (ML)* technologies. This consequently led to the development of recent architectures that enhance the speed of executing tinyML applications and hence, the simulator generator is also demonstrated by simulating an example of an architecture that accelerates tinyML applications - Bit-Serial Dot-Product Unit (BISDU) for microcontrollers [62].

Also, with the growing importance of security, many architectures include cryptographic modules and provide instructions in the ISA to enhance the speed of performing encryption and decryption, hence, the simulator generator is also demonstrated by simulating an architecture that enhances the speed of security operations. One such example is the acceleration of Galois Field (GF) multiplication for Advanced Encryption Standard (AES) using RISC-V Galois Field arithmetic extension [63].

RISC-V is an open standard ISA. Also, despite being new, the software support for RISC-V has been increasing over the past few years [64]. RV32I (where 'I' stands for 'integer') is one of the base integer ISAs in RISC-V that has a 32-bit address space. 32-bit address spaces lower memory traffic and energy consumption, are sufficient for many embedded and client devices, and are also sufficient for educational purposes [11]. Hence, these architectural and microarchitectural enhancements are added to a processor with RV32I ISA, and the following three case studies are considered:

1) Simulation of RISC-V with HyperWall
2) Simulation of RISC-V with bit-serial dot-product unit

Algorithm 2 Algorithm for the Detection of Structural Hazard in a Scalar Sequential Pipeline

```
if stage i is the last stage of the pipeline then
    There is no structural hazard to advance stage i.
else
    if instruction in stage (i+1) is not a no-operation instruction and instruction in stage (i+1) is not going to advance in the next clock cycle then
        There is a structural hazard to advance stage i.
    else
        There is no structural hazard to advance stage i.
    end if
end if
```

Algorithm 3 Algorithm for the Detection of RAW Hazard in a Scalar Sequential Pipeline

```
if i ≠ 2 then
    There is no RAW hazard to advance stage i.
else
    if instruction in stage i reads a register that is written by an instruction in any stage j such that j > i and j is less than the stage number in which register file is written then
        There is a RAW hazard to advance stage i.
    else
        if instruction in stage i performs a memory read from a location where an instruction in stage j performs a memory write, such that j > i and j is less than the stage number in which memory is written then
            There is a RAW hazard to advance stage i.
        else
            There is no RAW hazard to advance stage i.
        end if
    end if
end if
```

3) Simulation of RISC-V with Galois Field arithmetic extension

A. SIMULATION OF RISC-V WITH HYPERWALL

HyperWall [1], [2] is a hardware enhancement to provide security to virtual machines even in the presence of an untrusted hypervisor. HyperWall has already been implemented in an OpenSPARC T1 simulator, modeled after the UltraSPARC commercial microprocessor, and has already been simulated by modifying the Legion simulator [1], [2]. However, modifying a simulator is time-consuming and error-prone [24]. Furthermore, evaluating the performance of HyperWall added to different processors exacerbates the time and effort. The proposed simulator generator reduces the time and effort required for the performance evaluation by automatically generating the simulators for the different processors from the processor specifications written by the designer using the ADL.

An ADL program is written to simulate a 32-bit RISC-V processor with HyperWall. The processor has a five-stage pipeline consisting of the stages - instruction fetch (IF), instruction decode (ID), execute (EX), memory access (MEM), and write back (WB). Both the simulator generator and the simulator are run on a computer with a Core i5 processor with 16 GiB dynamic random access memory (DRAM) and 256 GiB solid state disk (SSD) installed with Ubuntu 18.04.6 LTS operating system. The configuration of the automatically generated simulator is given in Table 3.

The following three C programs are compiled using RISC-V GNU compiler toolchain to generate the corresponding RV32I program binaries:

1) Searching (Linear search) that outputs the result using printf()
2) Sorting (Bubble sort) that outputs the result using printf()
3) 999.specrand_ir workload from SPEC CPU2017 benchmarks

B. SIMULATION OF RISC-V WITH BIT-SERIAL DOT-PRODUCT UNIT

The bit-serial dot-product unit supports and accelerates the execution of mixed-precision low-bit quantized neural networks (QNNs) on resource-constrained microcontrollers. Mixed-precision QNNs limit the accuracy loss caused by low-bit quantization. BISDU is a multiplier-less dot-product unit that leverages the conventional logical operations of a microcontroller to perform multiplications. BISDU has already been evaluated by integrating it in a baseline 32-bit TinyRocket, a RISC-V (RV32) core available in the open-source Chipyard framework [62]. However, simulating

Algorithm 4 Algorithm for the Detection of Control Hazard in a Scalar Sequential Pipeline

```
if stage i is not the stage in which taken branch or conditional jump is detected or control hazard is already detected and the pipeline is already flushed then
    There is no control hazard due to instruction in stage i.
else
    if instruction in stage i is a taken branch then
        There is a control hazard due to instruction in stage i.
    else
        if instruction in stage i is an unconditional jump then
            There is a control hazard due to instruction in stage i.
        else
            There is no control hazard due to instruction in stage i.
        end if
    end if
end if
```

Algorithm 5 Algorithm for Automatic Detection and Resolution of Pipeline Hazards in a Scalar Sequential Pipeline

```
for every pipeline stage i do
    Find out whether there is a structural hazard to advance stage i.
    Find out whether there is a read-after-write (RAW) hazard to advance stage i.
    if there is a RAW hazard to advance stage i then
        There is a data hazard to advance stage i.
    else
        There is no data hazard to advance stage i.
    end if
    if there is a structural hazard to advance stage i or there is a data hazard to advance stage i then
        There is a hazard to advance stage i.
    else
        There is no hazard to advance stage i.
    end if
    Find out whether there is a control hazard due to instruction in stage i.
    if there is a control hazard due to instruction in stage i then
        Flush the pipeline from the first stage to stage (i-1).
    end if
    if instruction in stage i is a no-operation instruction or the number of clock cycles left to complete instruction in stage i is equal to 1 then
        The instruction in stage i is ready to advance.
    else
        The instruction in stage i is not ready to advance.
    end if
    if instruction in stage i is ready to advance and there is no hazard to advance stage i then
        The instruction in stage i will advance in the next clock cycle.
    else
        if instruction in stage i is not ready to advance then
            The instruction in stage i will progress in the same stage in the next clock cycle.
        else
            if instruction in stage i is ready to advance and there is a hazard to advance stage i then
                The instruction in stage i will stall in the next clock cycle.
            end if
        end if
    end if
end for
```

TABLE 3. Configuration of the automatically generated simulator.

Parameter	Value	Parameter	Value
Pipeline length	5		
Virtual address length	32 bits	Guest physical address length	34 bits
Physical address length	34 bits	Page size	4 KiB
PTE size	4 bytes		
I-TLB:			
No. of PTEs in each block	1	Associativity	1
No. of entries	128	Latency	1 cycle
D-TLB:			
No. of PTEs in each block	1	Associativity	1
No. of entries	128	Latency	1 cycle
Integer ALU latency	1 cycle	Float ALU latency	3 cycles
Integer MUL latency	3 cycles	Float MUL latency	5 cycles
Integer DIV latency	21 cycles	Float DIV latency	24 cycles
L1 I-cache:			
Block size	64 bytes	Associativity	8
Size	32 KiB	Latency	3 cycles
L1 D-cache:			
Block size	64 bytes	Associativity	8
Size	32 KiB	Latency	3 cycles
L2 cache:			
Block size	64 bytes	Associativity	8
Size	256 KiB	Latency	6 cycles
L3 cache:			
Block size	64 bytes	Associativity	8
Size	15 MiB	Latency	29 cycles
Physical memory latency	200 cycles		

ISA: Instruction set architecture, PTE: Page table entry, ALU: Arithmetic logic unit
I-TLB: Instruction translation lookaside buffer, D-TLB: Data translation lookaside buffer
L1: Level 1, L2: Level 2, L3: Level 3, I-cache: Instruction cache, D-cache: Data cache

it before prototyping will reduce the time for design space exploration.

An ADL program is written to simulate a 32-bit RISC-V processor with BISDU. BISDU adds five new instructions to RISC-V ISA - dot.n.u, dot.n.s, dot.s.u, dot.s.s, and pack - and these custom instructions are specified in addition to the RV32I instructions using the ADL. The configuration of the automatically generated simulator is the same as given in Table 3.

A macro BISDU_DOTP is written in C that can generate a custom BISDU instruction - dot.n.u, dot.n.s, dot.s.u or dot.s.s - depending upon the parameter, using the support for custom instructions in RISC-V inline assembly in gcc. Similarly, a macro BISDU_PACK is written in C that can generate the custom BISDU instruction - pack. A 3-bit signed $\times$ 2-bit signed bit-serial dot-product function - dotp_3x2 - is written in C as a static inline function that invokes the macro BISDU_DOTP six times with different parameters to process 32 elements per invocation. A bit-serial matrix multiplication function - matmul_3sx2s - is written in C that calls the function dotp_3x2 once to multiply two matrices of sizes 1×32 and 32×1. A C main() function is written to compute the dot product of two vectors A and B, each of length 32. The main() function invokes the macro BISDU_PACK twice, once to pack the bits of matrix A and next to pack the bits of matrix B. The main() function then calls the function matmul_3sx2s to compute the dot product of the vectors A and B. The program consisting of these macros and functions is compiled using the RISC-V GNU compiler toolchain to generate the corresponding RV32I program binary.

C. SIMULATION OF RISC-V WITH GALOIS FIELD ARITHMETIC EXTENSION

Advanced Encryption Standard uses Galois Field multiplication and hence, Galois Field multiplication has been accelerated using RISC-V Galois Field arithmetic extension [63]. GF multiplication may be performed in two steps - carry-less multiplication followed by polynomial reduction. Galois Field arithmetic extension for RISC-V has already been implemented using SweRV-EL2 1.3 on a Nexys A7 field-programmable gate array (FPGA) [63]. However, simulating it before prototyping will reduce the time for design space exploration.

An ADL program is written to simulate a 32-bit RISC-V processor with GF arithmetic extension. Four new instructions have been added to RISC-V ISA - clmul, clmulh, ffwidth, and ffred - and these custom instructions are specified in addition to the RV32I instructions using the ADL. The configuration of the automatically generated simulator is the same as given in Table 3.

Macros GF_ARITHMETIC_EXTENSION_CLMUL, GF_ARITHMETIC_EXTENSION_CLMULH, GF_ARITHMETIC_EXTENSION_FFWIDTH and GF_ARITHMETIC_EXTENSION_FFRED are written in C that can generate the custom GF arithmetic instructions - clmul, clmulh, ffwidth, and ffred - respectively, using the support for custom instructions in RISC-V inline assembly in gcc.

A C main() function is written to compute the GF multiplication of two polynomials $a(x)$ and $b(x)$, each represented as a number, the result of which is intended to

TABLE 4. Performance statistics.

Performance Statistics	RISC-V with HyperWall			RISC-V with BISDU	RISC-V with GF Arithmetic Extension
	Searching (Linear Search)	Sorting (Bubble Sort)	999.specrand_ir Workload from SPEC CPU2017 Benchmarks	Computing Dot Product	Computing GF Multiplication
I-TLB hit count	11259	16671	76265332	7365	3273
I-TLB miss count	21	21	81	29	21
D-TLB hit count	2904	4448	3406607	2065	858
D-TLB miss count	780	1048	776879	480	242
L1 I-cache hit count	11250	16653	76265231	7361	3267
L1 I-cache miss count	30	39	182	33	27
L1 D-cache hit count	8584	11990	8845317	5510	2623
L1 D-cache miss count	68	56	475	89	55
L2 cache hit count	70	55	451	94	54
L2 cache miss count	28	40	206	28	28
L3 cache hit count	18	34	197	22	24
L3 cache miss count	10	6	9	6	4
Instruction count	9634	14331	63555070	6466	2776
Clock cycle count	504133	694798	674506754	335144	171144
CPI	52.33	48.48	10.61	51.83	61.65
Simulation time	6.063 s	8.141 s	17447.164 s	3.879 s	1.963 s

BISDU: Bit-serial dot-product unit
GF: Galois Field
I-TLB: Instruction translation lookaside buffer, D-TLB: Data translation lookaside buffer
L1: Level 1, L2: Level 2, L3: Level 3, I-cache: Instruction cache, D-cache: Data cache
CPI: Cycles per instruction

be used for AES. The main() function invokes the macro GF_ARITHMETIC_EXTENSION_FFWIDTH to specify the degree m and the irreducible polynomial f. AES uses the irreducible polynomial $f(x) = x^8 + x^4 + x^3 + x + 1$ that has a degree of 8. The main() function then invokes the macro GF_ARITHMETIC_EXTENSION_CLMUL to compute the lower 32 bits of the carry-less product of $a(x)$ and $b(x)$. Since AES uses polynomials that belong to $GF(2^8)$, the higher 32 bits of the carry-less product of $a(x)$ and $b(x)$ are always 0 and hence, the macro GF_ARITHMETIC_EXTENSION_CLMULH is not invoked, but the corresponding value is directly specified as 0. Then, the main() function invokes the macro GF_ARITHMETIC_EXTENSION_FFRED to perform the polynomial reduction. The program consisting of these macros and the main() function is compiled using RISC-V GNU compiler toolchain to generate the corresponding RV32I program binary.

In each of the three case studies, the binaries are simulated using the simulator generated by the simulator generator, and the results are obtained.

IX. RESULTS

The performance statistics obtained from the simulator while running the different programs are tabulated in Table 4. It is seen that as the instruction count (number of dynamic instructions simulated) increases, the cycles per instruction (CPI) reduces. This is because of the nature of pipelining. The simulation time is also shown in Table 4. Thus, the simulators for three different architectures are automatically generated using the simulator generator by simply writing the different processor specifications using the ADL.

X. CONCLUSION

Since modern processors are enhanced for security in virtualization, this research is valuable as it provides an approach to simulate the additional clock cycles required for performing memory access checks to provide security in virtualization. The paper also provides an approach to simulate a new RISC ISA (ISA having fixed-length instructions) and a new pipeline with any number of stages with automatic detection of pipeline hazards and automatic stalling or flushing of pipeline on detection of hazards. This is done by developing a new ADL for processor specifications which can be read by the proposed simulator generator to generate the desired simulator. The paper concludes that the proposed approaches help in accurately simulating the overhead due to security in virtualization and also in providing flexibility to the designer to simulate the desired processor configurations.

XI. FUTURE SCOPE

The simulator generated by the proposed simulator generator supports only one user process and only one OS. It may be modified further to support multiple processes and multiple OSs. The proposed simulator generator may also be extended to simulate superscalar, out-of-order, and multicore processors.

FPGAs may be harnessed for simulation [23] because FPGA-accelerated simulators are several orders of magnitude faster than their software counterparts, especially when simulating multicore systems [65].

Whenever a new ISA is designed, a new compiler is also needed to generate binaries for the new ISA. Hence, another interesting research direction would be to generate a compiler automatically along with the simulator from the ISA specifications.

REFERENCES

[1] J. Szefer and R. B. Lee, "Architectural support for hypervisor-secure virtualization," *ACM SIGPLAN Notices*, vol. 47, no. 4, pp. 437–450, Mar. 2012.

[2] J. M. Szefer, "Architectures for secure cloud computing servers," Ph.D. thesis, Dept. of Electrical Engineering, Princeton Univ., Princeton, NJ, USA, 2013.

[3] R. Jithin and P. Chandran, "Dynamic partitioning of physical memory among virtual machines: ASMI: Architectural support for memory isolation," in *Proc. 31st Annu. ACM Symp. Appl. Comput.*, New York, NY, USA, Apr. 2016, pp. 474–476.

[4] P. Y. H. Hin, X. Liao, J. Cui, A. Mondelli, T. M. Somu, and N. Zhang, "Supporting risc-v full system simulation in gem5," in *Proc. Comput. Archit. Res. RISC-V (CARRV)*, New York, NY, USA, 2021, pp. 1–6.

[5] S. C. Mhatre, P. Chandran, and R. Jithin, "On the simulation of processors enhanced for security in virtualization," in *Proc. Companion ACM/SPEC Int. Conf. Perform. Eng.*, New York, NY, USA, Apr. 2018, pp. 51–52.
[6] P. Chandran, "Automatic generation of compiled cycle level microarchitecture simulators for superspeculative processors," Ph.D. thesis, Supercomput. Educ. Res. Centre, Indian Inst. Sci. Bengaluru, Bengaluru, Karnataka, Jun. 2004.
[7] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Commun. ACM*, vol. 17, pp. 412–421, Jul. 1974.
[8] K. Asanovic and D. A. Patterson, "Instruction sets should be free: The case for risc-v," Dept. Electrical Engineering and Computer Sciences, Univ. California at Berkeley, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2014-146, Aug. 2014.
[9] A. Waterman, "Design of the RISC-V instruction set architecture," Dept. Electrical Engineering and Computer Sciences, Univ. California at Berkeley, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2016-1, Jan. 2016.
[10] T. Chen and D. A. Patterson, "RISC-V geneology," Dept. Electrical Engineering and Computer Sciences, Univ. California at Berkeley, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2016-6, Jan. 2016.
[11] *The RISC-V Instruction Set Manual Volume I, Unprivileged Architecture, Version 20240411*, RISC-V Found., Berkeley, CA, USA, May 2024.
[12] *The RISC-V Instruction Set Manual: Volume II, Privileged Architecture, Version 20240411*, RISC-V Found., Berkeley, CA, USA, May 2024.
[13] Z. Benomar, F. Longo, G. Merlino, and A. Puliafito, "Cloud-based enabling mechanisms for container deployment and migration at the network edge," *ACM Trans. Internet Technol.*, vol. 20, no. 3, pp. 1–28, Jun. 2020.
[14] Q. Zhang, L. Liu, C. Pu, Q. Dou, L. Wu, and W. Zhou, "A comparative study of containers and virtual machines in big data environment," in *Proc. IEEE 11th Int. Conf. Cloud Comput. (CLOUD)*, Jul. 2018, pp. 178–185.
[15] G. Radchenko, A. Alaasam, and A. Tchernykh, "Comparative analysis of virtualization methods in big data processing," *Supercomput. Frontiers Innov.*, vol. 6, no. 1, pp. 48–79, 2019.
[16] R. Jithin and P. Chandran, "Virtual machine isolation—A survey on the security of virtual machines," in *Proc. The 2nd Int. Conf. Secur. Comput. Netw. Distrib. Syst. (SNDS)*, 2014, pp. 91–102.
[17] M. Pearce, S. Zeadally, and R. Hunt, "Virtualization: Issues, security threats, and solutions," *ACM Comput. Surv.*, vol. 45, pp. 17:1–17:39, Mar. 2013.
[18] S. Shoundic, P. Chandran, P. Krishna, V. Reddy, B. Jayachandra, and L. Pande, "Extended-HyperWall: Hardware support for rollback secure virtualization," in *Proc. Int. Conf. Adv. Comput., Commun. Informat. (ICACCI)*, Sep. 2016, pp. 1674–1681.
[19] S. Shoundic, "Extended hyperwall: Hardware support for rollback secure virtualization," M. Tech dissertation, Dept. of Computer Science and Engineering, Nat. Inst. Technol. Calicut, Calicut, Kerala, May 2015.
[20] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata, "Innovative technology for cpu based attestation and sealing, white paper," in *Proc. 2nd Int. Workshop Hardw. Architectural Support Secur. Privacy*, Aug. 2013, pp. 1–7.
[21] "Intel trust domain extensions," Intel Corp., Santa Clara, CA, USA, White Paper, Feb. 2023.
[22] D. Kaplan, J. Powell, and T. Woller, "AMD memory encryption," Adv. Micro Devices, Inc., Santa Clara, CA, USA, White Paper, 2021.
[23] H. Angepat, D. Chiou, E. S. Chung, and J. C. Hoe, *FPGA-Accelerated Simulation of Computer Systems*. San Rafael, CA, USA: Morgan & Claypool, 2014.
[24] C. Barnes, P. Vaidya, and J. John Lee, "An XML-based ADL framework for automatic generation of multithreaded computer architecture simulators," *IEEE Comput. Archit. Lett.*, vol. 8, no. 1, pp. 13–16, Jun. 2009.
[25] A. Akram and L. Sawalha, "A survey of computer architecture simulation techniques and tools," *IEEE Access*, vol. 7, pp. 78120–78145, 2019.
[26] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation and scheduling for efficient GPU control flow," in *Proc. 40th Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, Dec. 2007, pp. 407–420.
[27] V. Zivojnovic and H. Meyr, "Compiled HW/SW co-simulation," in *Proc. 33rd Design Autom. Conf.*, Jun. 1996, pp. 690–695.
[28] E. C. Schnarr, M. D. Hill, and J. R. Larus, "Facile: A language and compiler for high-performance processor simulators," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement.*, New York, NY, USA, May 2001, pp. 321–331.
[29] S. Onder and R. Gupta, "Automatic generation of microarchitecture simulators," in *Proc. Int. Conf. Comput. Lang.*, May 1998, pp. 80–89.
[30] F. Larsson, P. Magnusson, and B. Werner, "Simgen: Development of efficient instruction set simulators," Swedish Inst. Comput. Sci., Kista, Sweden, Tech. Rep. R97, 1997, p. 3.
[31] M. Freericks, "The nML machine description formalism," Dept. Comput. Sci., Technische Universitat Berlin, Berlin, Germany, Tech. Rep., 1991.
[32] P. Chandran, L. Garg, and A. Kumar, "Multikernel simulation: A new approach to study rollback sensitive memory architecture," in *Proc. IEEE 3rd Int. Conf. Collaboration Internet Comput. (CIC)*, Oct. 2017, pp. 437–442.
[33] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 164–177, Oct. 2003.
[34] S. C. Mhatre and P. Chandran, "On making xen detect hypercalls and memory accesses for simulating virtualization-enabled processors," in *Proc. 35th Annu. ACM Symp. Appl. Comput.*, New York, NY, USA, Mar. 2020, pp. 154–161.
[35] S. C. Mhatre and P. Chandran, "On the measurement of performance metrics for virtualization-enhanced architectures," in *Proc. 38th ACM/SIGAPP Symp. Appl. Comput.*, New York, NY, USA, Mar. 2023, pp. 49–56.
[36] S. Mhatre and P. Chandran, "On the simulation of hypervisor instructions for accurate timing simulation of virtualized systems," *Int. J. Inf. Technol.*, vol. 2024, pp. 1–10, May 2024.
[37] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proc. Annu. Conf. USENIX Annu. Tech. Conf.*, Apr. 2005, p. 41.
[38] J. Wu, X. Zhao, X. Sui, and X. Yang, "PVMsim: A parallel simulation platform to evaluate virtual machines," in *Proc. 2nd Int. Conf. Future Comput. Commun.*, vol. 2, May 2010, pp. V2–551–V2–556.
[39] A. Garcia-Guirado, R. Fernandez-Pascual, and J. M. Garcia, "Virtualgems: An infrastructure to simulate virtual machines," in *Proc. Int. Workshop Modeling, Benchmarking Simulation*, 2009, pp. 53–63.
[40] S. H. Nikounia and S. Mohammadi, "Gem5v: A modified gem5 for simulating virtualized systems," *J. Supercomput.*, vol. 71, no. 4, pp. 1484–1504, Apr. 2015.
[41] P. Bohrer, J. Peterson, M. Elnozahy, R. Rajamony, A. Gheith, R. Rockhold, C. Lefurgy, H. Shafi, T. Nakra, R. Simpson, E. Speight, K. Sudeep, E. Van Hensbergen, and L. Zhang, "Mambo: A full system simulator for the PowerPC architecture," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 31, no. 4, pp. 8–12, Mar. 2004.
[42] M. T. Yourst, "PTLsim: A cycle accurate full system x86–64 microarchitectural simulator," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, Apr. 2007, pp. 23–34.
[43] A. Patel, F. Afram, S. Chen, and K. Ghose, "MARSS: A full system simulator for multicore x86 CPUs," in *Proc. 48th ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, Jun. 2011, pp. 1050–1055.
[44] A. Tchana, B. Ekane, B. Teabe, and D. Hagimont, "VMcSim: A detailed manycore simulator for virtualized systems," in *Proc. IEEE 8th Int. Conf. Cloud Comput.*, Jun. 2015, pp. 195–202.
[45] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.
[46] A. Roelke and M. R. Stan, "RISC5: Implementing the RISC-V ISA in gem5," in *Proc. Comput. Archit. Res. RISC-V (CARRV)*, New York, NY, USA, 2017, pp. 1–7.
[47] T. Ta, L. Cheng, and C. Batten, "Simulating multi-core RISC-V systems in gem5," in *Proc. Comput. Archit. Res. RISC-V (CARRV)*, New York, NY, USA, 2018, pp. 1–24.
[48] L. Das, *The X86 Microprocessor*, 2nd ed., London, U.K.: Pearson, 2014.
[49] I. A. Assir, M. E. Iskandarani, H. R. A. Sandid, and M. A. R. Saghir, "Arrow: A RISC-V vector acclerator for machine learning inference," in *Proc. Comput. Archit. Res. RISC-V (CARRV)*, New York, NY, USA, 2021, pp. 1–6.
[50] N. M. Qui, C. H. Lin, and P. Chen, "Design and implementation of a 256-bit RISC-V-Based dynamically scheduled very long instruction word on FPGA," *IEEE Access*, vol. 8, pp. 172996–173007, 2020.
[51] *Risc-v 'V' Vector Extension, Version 0.9*, RISC-V Found., Berkeley, CA, USA, May 2020.
[52] E. Cui, T. Li, and Q. Wei, "RISC-V instruction set architecture extensions: A survey," *IEEE Access*, vol. 11, pp. 24696–24711, 2023.

[53] R. Zhan and X. Fan, "Neuralscale: A RISC-V based neural processor boosting ai inference in clouds," in *Proc. Comput. Archit. Res. RISC-V (CARRV)*, New York, NY, USA, 2021, pp. 1–7.
[54] K. K. Balasubramanian, M. Di Salvo, W. Rocchia, S. Decherchi, and M. Crepaldi, "Designing RISC-V instruction set extensions for artificial neural networks: An LLVM compiler-driven perspective," *IEEE Access*, vol. 12, pp. 55925–55944, 2024.
[55] *Risc-v 'V' Vector Extension, Version 0.8*, RISC-V Found., Berkeley, CA, USA, Dec. 2019.
[56] Y. Chen, H. Lan, Z. Du, S. Liu, J. Tao, D. S. Han, T. Luo, Q. Guo, L. Li, Y. Xie, and T. Chen, "An instruction set architecture for machine learning," *ACM Trans. Comput. Syst.*, vol. 36, no. 3, pp. 1–35, Aug. 2019.
[57] J. L. Hennessy and D. A. Patterson, *Computer Architecture, A Quantitative Approach*, 6th ed., San Mateo, CA, USA: Morgan Kaufmann, 2019.
[58] T. Theocharides, C. Frenkel, and L. Cavigelli, "Introduction to the special issue on TinyML," *ACM Trans. Embedded Comput. Syst.*, vol. 23, no. 3, pp. 1–5, May 2024.
[59] V. Tsoukas, A. Gkogkidis, E. Boumpa, and A. Kakarountas, "A review on the emerging technology of TinyML," *ACM Comput. Surv.*, vol. 56, no. 10, pp. 1–37, Jun. 2024.
[60] P. P. Ray, "A review on TinyML: State-of-the-art and prospects," *J. King Saud Univ.-Comput. Inf. Sci.*, vol. 34, no. 4, pp. 1595–1623, Apr. 2022.
[61] M. Shafique, T. Theocharides, V. J. Reddy, and B. Murmann, "TinyML: Current progress, research challenges, and future roadmap," in *Proc. 58th ACM/IEEE Design Autom. Conf. (DAC)*, Dec. 2021, pp. 1303–1306.
[62] D. Metz, V. Kumar, and M. Själander, "BISDU: A bit-serial dot-product unit for microcontrollers," *ACM Trans. Embedded Comput. Syst.*, vol. 22, no. 5, pp. 1–22, Sep. 2023.
[63] Y.-M. Kuo, F. Garcia-Herrero, and J. A. Maestro, "Versatile RISC-V isa Galois field arithmetic extension for cryptography and error-correction codes," in *Proc. Comput. Archit. Res. RISC-V (CARRV)*, New York, NY, USA, 2021, pp. 1–6.
[64] B. W. Mezger, D. A. Santos, L. Dilillo, C. A. Zeferino, and D. R. Melo, "A survey of the RISC-V architecture software support," *IEEE Access*, vol. 10, pp. 51394–51411, 2022.
[65] W. Simoneau and R. Sendag, "An FPGA-based multi-core platform for testing and analysis of architectural techniques," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, Apr. 2012, pp. 68–77.

SWAPNEEL C. MHATRE received the B.E. degree in computer engineering from the Bharati Vidyapeeth College of Engineering (affiliated to the University of Mumbai), Maharashtra, India, in June 2000, and the M.E. degree in computer engineering from the Government College of Engineering, Pune (COEP) (affiliated to the University of Pune), Maharashtra, India, in December 2001. Currently, he is pursuing the Ph.D. degree in computer science and engineering with the National Institute of Technology (NIT) Calicut, Kerala, India, under the guidance of Dr. Priya Chandran.

He was a Faculty Member with the Engineering Colleges for 15 years and five months. He has filed a patent, has publications in conferences and journals and has written two books on *Microprocessor* (Jaico Publishing House). He has received the Mission 10X certificate in teaching and learning. His area of research interest includes computer architecture.

PRIYA CHANDRAN (Senior Member, IEEE) received the B.Tech. degree in electronics and communications from the Maulana Azad National Institute of Technology, Bhopal, Madhya Pradesh, India, the M.Tech. degree in computer science and technology from Indian Institute of Technology Roorkee, Uttar Pradesh, India, and the Ph.D. degree from Indian Institute of Science, Bangalore, Karnataka, India.

Currently, she is a Professor with the Department of Computer Science and Engineering and Dean (Planning and Development), National Institute of Technology (NIT) Calicut, Kerala, India. Her areas of research interests include computer architecture, design, and analysis of algorithms, theoretical computer science, high performance computing, and formal methods for information security. She has numerous publications in conferences and journals.