Xu YN, Yu ZH, Wang KF *et al.* Functional verification for agile processor development: A case for workflow integration. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 38(4): 737-753 July 2023. DOI: 10.1007/s11390-023-3285-8

Functional Verification for Agile Processor Development: A Case for Workflow Integration

Yi-Nan Xu[1, 2] (徐易难), *Student Member, CCF, Graduate Student Member, IEEE*, Zi-Hao Yu[1] (余子濠)
Kai-Fan Wang[1, 2] (王凯帆), Hua-Qiang Wang[1, 2] (王华强), Jia-Wei Lin[1, 2] (蔺嘉炜), Yue Jin[1, 2] (金　越)
Lin-Juan Zhang[1, 2] (张林隽), Zi-Fei Zhang[1, 2] (张紫飞), Dan Tang[1, 3] (唐　丹), Sa Wang[1] (王　卅)
Kan Shi[1] (石　侃), Ning-Hui Sun[1, 2] (孙凝晖), *Fellow, CCF, Member, ACM, IEEE*, and
Yun-Gang Bao[1, 2, *] (包云岗), *Senior Member, CCF, Member, ACM, IEEE*

[1] *State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China*

[2] *University of Chinese Academy of Sciences, Beijing 100049, China*

[3] *Beijing Institute of Open Source Chip, Beijing 100080, China*

E-mail: xuyinan@ict.ac.cn; yuzihao@ict.ac.cn; wangkaifan@ict.ac.cn; wanghuaqiang20s@ict.ac.cn; linjiawei20s@ict.ac.cn
jinyue20s@ict.ac.cn; zhanglinjuan20s@ict.ac.cn; zhangzifei@ict.ac.cn; tangdan@ict.ac.cn; wangsa@ict.ac.cn
shikan@ict.ac.cn; snh@ict.ac.cn; baoyg@ict.ac.cn

Received April 7, 2023; accepted July 5, 2023.

Abstract Agile hardware development methodology has been widely adopted over the past decade. Despite the research progress, the industry still doubts its applicability, especially for the functional verification of complicated processor chips. Functional verification commonly employs a simulation-based method of co-simulating the design under test with a reference model and checking the consistency of their outcomes given the same input stimuli. We observe limited collaboration and information exchange through the design and verification processes, dramatically leading to inefficiencies when applying the conventional functional verification workflow to agile development. In this paper, we propose workflow integration with collaborative task delegation and dynamic information exchange as the design principles to effectively address the challenges on functional verification under the agile development model. Based on workflow integration, we enhance the functional verification workflows with a series of novel methodologies and toolchains. The diff-rule based agile verification methodology (DRAV) reduces the overhead of building reference models with runtime execution information from designs under test. We present the RISC-V implementation for DRAV, DiffTest, which adopts information probes to extract internal design behaviors for co-simulation and debugging. It further integrates two plugins, namely XFUZZ for effective test generation guided by design coverage metrics and LightSSS for efficient fault analysis triggered by co-simulation mismatches. We present the integrated workflows for agile hardware development and demonstrate their effectiveness in designing and verifying RISC-V processors with 33 functional bugs found in NutShell. We also illustrate the efficiency of the proposed toolchains with a case study on a functional bug in the L2 cache of XiangShan.

Keywords functional verification, agile development, open-source hardware, workflow integration

1 Introduction

Since its inception in the 20th century, the semiconductor industry has developed mature workflows for designing, verifying, and fabricating processor chips. As illustrated in Fig.1(a), the conventional waterfall model follows a sequential development workflow, starting from design specifications for desired

Regular Paper

This work was supported in part by the Strategic Priority Research Program of Chinese Academy of Sciences (CAS) under Grant No. XDC05030200, the National Key Research and Development Program of China under Grant No. 2022YFB4500403, the National Natural Science Foundation of China under Grant Nos. 62090022 and 62172388, the Youth Innovation Promotion Association of the Chinese Academy of Sciences under Grant No. 2020105, and the Innovation Grant No. E261100 by Institute of Computing Technology, Chinese Academy of Sciences.

*Corresponding Author

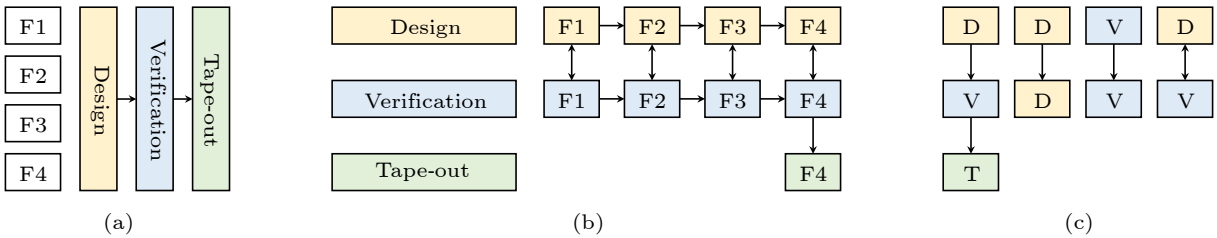


Fig.1. Waterfall and agile models for developing processors with a breakdown of the development workflows. (a) The waterfall model adopts a sequential workflow for predetermined features (F1–F4). (b) The agile model employs iterative processes with incremental design changes for the features. (c) While the waterfall requires only the design (D), verification (V), and tape–out (T) workflow, the agile model has further demands for the design–design (D–D), verification–verification (V–V), and design–verification (D–V) workflows.

features (F1–F4) and ending with the delivery of fully functional chips. Despite its clear and structured approach with well-defined stages and deliverables, the waterfall model can also be inflexible, as changes made at later stages can be difficult and expensive to implement.

Over the past decade, with the growing demand from emerging fields, domain-specific architectures have presented significant challenges in terms of efficiency and flexibility in hardware development[1]. To tackle the challenges, agile hardware development methodologies have attracted great attention from academia and industry[2]. As Fig.1(b) demonstrates, the agile paradigm emphasizes the quick response to requirements and adopts an iterative development process with improved tools and workflows[3].

Despite the research progress, according to recent studies[4, 5], the applicability and scalability of agile hardware development, especially regarding the functional verification efficiency for complicated processor designs, still need to be improved.

This paper attempts to enhance the conventional dynamic functional verification methodology and apply it to agile hardware development. This verification approach usually employs a simulation-based workflow to verify the design functionalities against those of specific golden reference models given the same stimuli[6]. The workflow involves four stages, including the development of the design under test (DUT) and the reference model (REF), co-simulation, test generation, and fault analysis.

Although the industry has established standardized dynamic verification workflows[7], agile development methodologies have changed how the verification stage interacts with the design stage and raised new challenges. As shown in Fig.1(c), the waterfall model mainly adopts the sequential design (D), verification (V), and tape–out (T) workflow with usually separated working teams and toolchains for each stage. However, the agile model further requires more workflows for design–design (D–D), verification–verification (V–V), and design–verification (D–V) iterations to develop various features. In this development model, the verification stage stays in lock-step with the design stage and cannot progress independently. Therefore, the waterfall and agile models put different demands on the hardware development infrastructures. Applying conventional verification workflows and toolchains in agile development remains challenging, as their feasibility and suitability are uncertain.

A significant transition of the agile development model is the adoption of high-level hardware construction languages (HCLs)[8]. While improving design efficiency, they also bring about transformational changes in DUTs (D–D) and disrupt conventional D–V–T workflows that are mainly developed for Verilog designs. For example, in the traditional waterfall model, design parameters and specifications are mostly predetermined, with which designers and verifiers independently construct DUT and REF models using Verilog and SystemVerilog. The verification process spends most of its time comparing the functionalities of DUT and REF without worrying about the development of REF (V–V). However, in the agile model, designs may rapidly change in response to feature requests (D–D) and require changes in V–V as well. If we still use Verilog and SystemVerilog for verification, it will take a long time to develop the REFs due to their limited programming efficiency compared with HCLs.

We observe limited collaboration and information exchange in current functional verification workflows and toolchains with the transition from waterfall development to the agile model. For example, while the conventional waterfall model is suitable for relatively stable processor designs and separated working teams, the agile model with a common adoption of HCLs leads to more rapid design changes. However, the

changing design information cannot be effectively shared across current workflows, resulting in information gaps and development inefficiencies.

To address the challenges, we propose workflow integration with collaborative task delegation and dynamic information exchange. These principles provide a general paradigm for identifying opportunities and optimizing the development toolchains towards the integrated agile design and verification workflows.

In particular, we investigate the obstacles of dynamic functional verification for agile development and enhance the design methodologies of the REFs, co-simulation frameworks, test generation approaches, and fault analysis techniques. We present the diff-rule based agile verification (DRAV) methodology to simplify the REFs. Based on the DRAV methodology, we propose DiffTest, a dynamic verification framework for RISC-V processors, and provide classifications of the diff-rules. We also introduce a generic test generation tool called XFUZZ and propose an efficient snapshot technique Lightweight Simulation Snapshot (LightSSS) to accelerate the debugging process. We further present a comprehensive overview of the integrated design and functional verification workflows with discussions on how useful information is effectively identified, transferred, and utilized through them. We evaluate the proposed workflows in verifying two RISC-V processors and demonstrate the effectiveness of the presented tools by finding 33 functional bugs in NutShell. We also offer a case study on XiangShan that illustrates the efficiency of the integrated workflows and toolchains.

Building upon the results presented in [4, 5], this paper further makes the following contributions.

- We investigate major obstacles to functional verification in the era of agile development and observe limited collaboration and information exchange among various development stages.
- We propose workflow integration with collaborative task delegation and dynamic information exchange as the principles for optimizing workflows and toolchains for agile hardware development.
- We enhance the functional verification workflow with a classification of diff-rules and a coverage-guided test generation technique, and present the integrated workflows for RISC-V processors.
- We evaluate the usage of DiffTest, XFUZZ, and LightSSS and demonstrate their efficiency and effectiveness in verifying two RISC-V processors.

The rest of this paper is organized as follows. We begin by introducing the background of functional verification and agile development in Section 2. We then present our key insights into the emerging challenges and opportunities that the agile paradigm poses on functional verification in Section 3. Next, Section 4 describes the design methodologies of the proposed tools and workflows, which are further evaluated in Section 5. Finally, we discuss the future and related work in Section 6, and conclude this paper in Section 7.

2 Background

2.1 Agile and Open-Source Hardware

Agile chip development and open-source hardware have gained ever-growing attention over the past years. In contrast to the conventional waterfall model, the agile model appeals to an iterative, responsive, and flexible development methodology[3]. The agile approach aims to reduce significant engineering costs and long design cycles for chip development.

The XiangShan team suggests three levels of open-source hardware[5], including 1) L1: instruction sets are open and free, such as the RISC-V, 2) L2: the design and the implementation are open and free, such as the open-source RISC-V processors, and 3) L3: the development infrastructures are open and free. To facilitate a more practical chip development workflow, researchers have proposed a variety of tools for different working stages, such as simulation[9], formal verification[10], and prototyping[11], which collectively contribute to the L3 open-source hardware ecosystem.

However, despite its popularity in recent years, the agile and open methodology is still unrecognized for high-performance and complicated designs. Although a number of chips have been built using agile approaches, most of them are research prototypes and are relatively small or less complicated designs. It is still being determined if similar approaches can be applied to large-scale designs such as modern processors.

The most recent practice of using agile methodologies for developing high-performance RISC-V processors is carried out by the XiangShan team[4, 5]. They present the Minjie platform with novel tools and use the presented toolchains to develop two generations of XiangShan processors with industry-competitive performance. In this paper, we will provide a systematic perspective into the functional verification workflows of Minjie and propose novel design methodologies of the toolchains for agile hardware development.

2.2 Functional Verification

Functional correctness is an essential and fundamental requirement for processors to perform their intended operations correctly. As chips cannot be changed once they are fabricated, it is critical to ensure that all possible scenarios and use cases are covered in the verification stage. The common verification approaches for processors can be classified into two categories, namely static and dynamic methods.

The static verification methodologies use formal and mathematical techniques to fully examine the design space. Nonetheless, its practicality is inevitably curtailed by the significant domain expertise and the state explosion issues[12].

Due to the ease of implementation, dynamic verification approaches have gained greater popularity. They usually employ simulation-based workflows to verify the design functionalities against those of specific golden reference models given the same stimuli[6].

However, as modern processors continue to evolve and incorporate advanced features, it becomes increasingly difficult to identify all potential issues during the functional verification stage. This creates a major bottleneck in the chip development process.

The most challenging aspect of functional verification is to ensure that the design is thoroughly explored and that functional correctness is verified for all possible use cases, given the limited time and resources available before the chips are shipped for manufacturing.

This challenge can be generally tackled by improving functional verification workflows' effectiveness or efficiency. Formal methods and test generation techniques usually address the effectiveness, such as simplifying the problem or using fewer resources to cover a larger state space. On the other hand, the brute-force way proposes faster and reusable tools to reduce and amortize verification costs for every step, such as simulation acceleration tools. Both approaches are believed necessary for the functional verification of processors and complementary to each other.

2.3 Simulation-Based Dynamic Verification

Over the past few decades, due to the limitations of formal methodologies for large-scale designs, simulation-based dynamic verification has remained practically popular in academia and industry.

As illustrated in Fig.2, the common and standardized practice[6, 7] is to build a co-simulation framework between the design under test (DUT) and a reference model (REF) and compare their outcomes with the same test stimuli. Accordingly, to enhance the verification effectiveness and efficiency, researchers have proposed techniques for ① accurately describing the DUTs and REFs, ② improving the co-simulation speed, ③ generating higher-quality test cases, and ④ flexibly analyzing the simulation results.

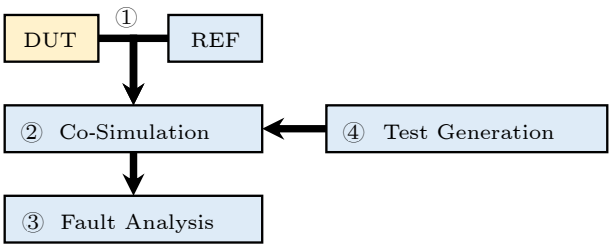


Fig.2. Simulation-based dynamic verification workflows.

Due to the popularity of using Verilog for hardware description, conventional verification toolchains and workflows are developed mainly for Verilog. For example, Verilog supports non-synthesizable expressions used widely by verification engineers. Together with synthesizable grammars, they provide a universal programming environment for hardware developers with end-to-end support of commercial toolchains. However, with the rise of agile hardware development, traditional verification workflows face severe technical challenges, which will be detailed later in Section 3.

3 Challenges

As the agile development methodology is increasingly adopted, it poses new challenges for the functional verification of processor designs, particularly for the dynamic co-simulation workflow. This section explores and discusses the challenges that arise in functional verification due to adopting agile design methodologies.

As introduced in Fig.1 and Subsection 2.1, the waterfall hardware development model is characterized by well-defined stages carried out sequentially (D–V–T), transferring specific deliverables from one stage to the next. In contrast, the agile model is an iterative and flexible approach to hardware development. Design and verification stages are interleaved with frequent interactions, such as D–D, D–V, and V–V.

To facilitate the hardware design efficiency, novel toolchains have been proposed and accepted in prac-

tice. For example, high-level hardware construction languages (HCLs)[8] enable object-oriented and functional programming paradigms for the hardware. They are used to build hardware generators[13] to support rapid design changes in the agile development model.

However, emerging agile design methodologies offer limited benefits to the hardware verification stage and can disrupt the conventional functional verification workflows to some extent. As discussed in Subsections 2.2 and 2.3, simulation-based functional verification typically consists of four stages, namely development of DUT and REF, co-simulation of DUT and REF, test generation, and fault analysis. Since agile design methodologies change how DUTs are developed, other stages in functional verification are significantly affected.

Reference Model (*REF*). Under the traditional development model, given the same hardware design specifications, RTL design engineers implement the processor in hardware description languages, while verification engineers develop corresponding reference models for various parts and levels of the processor. However, the HCL-based hardware generators, with diverse implementation details and thus various behaviors, may require multiple traditional REFs to be maintained for different DUTs and hinder the reusability of REFs across V–V iterations. This issue will be discussed with more examples in Subsections 4.2 and 4.3.

Co-Simulation Framework. High-level HCL designs are often simulated via the emitted Verilog code. However, processor design generators in HCLs can result in frequent changes to their generated code, which may involve a large number of internal signal definitions for co-simulation. For example, Dromajo[6] requires the program counter and instruction for co-simulation, whose implementations will likely differ on different designs. Traditionally, verification engineers must adapt the co-simulation framework manually for different DUTs. This results in repetitive and tedious porting efforts in D–V cycles, which will be addressed later in Subsection 4.4.

Test Generation Techniques. Functional verification adopts test generation approaches to create stimuli for DUTs. Their effectiveness for achieving higher coverage metrics usually relies on design-specific optimizations. However, existing handcrafted proposals may not be suitable for the agile design paradigm with frequent design changes and limited D–V cycles. We attempt to address this challenge in Subsection 4.5.

Fault Analyzers. Software-based RTL simulators are widely used for simulating circuits and offer complete visibility into the simulated designs. Nevertheless, these simulators may only run at a frequency of KHz for large designs, and enabling debugging features, such as dumping waveform, can further slow them down. Despite recent studies trying to tackle this issue, their brute-force solutions still cause either waste of resources or significant performance overhead. We will tackle this issue in Subsection 4.6.

As discussed above, adopting agile development methodologies introduces new challenges to functional verification workflows. By reviewing these emerging issues, we observe limited collaboration and information exchange among various stages.

For example, with the adoption of HCLs, it becomes popular to maintain DUTs as hardware generators with frequent changes in the generated Verilog code. Though co-simulation relies on internal design details, the typical deliverables from the design stage to the verification stage contain solely the design specifications and generated Verilog code, which result in the broken co-simulation framework.

However, hardware designers have a comprehensive understanding of the revisions in HCLs and thus have the capabilities of maintaining verification interfaces for co-simulation. As such, there exist opportunities that this issue could be addressed by standardized co-simulation interfaces for collaboration and information exchange between the design and verification stages.

4 Design

In Section 3, we outline the challenges that agile development presents for functional verification. By understanding the complexities, this section further proposes workflow integration as a viable solution, fostering collaborative task delegation and dynamic information exchange. Based on these design principles for agile development toolchains, we propose novel methodologies and techniques to enhance the conventional functional verification workflows.

4.1 Workflow Integration

To effectively address the challenges on functional verification under the agile development model, we

propose workflow integration. It refers to the process of combining and coordinating different development stages and workflows to enhance the overall efficiency of agile hardware development.

Fig.3 illustrates the proposed functional verification workflows with the adoption of workflow integration. Unlike the separated stages in Fig.2, the enhanced workflows are closely integrated with collaborative task delegation and dynamic information exchange.

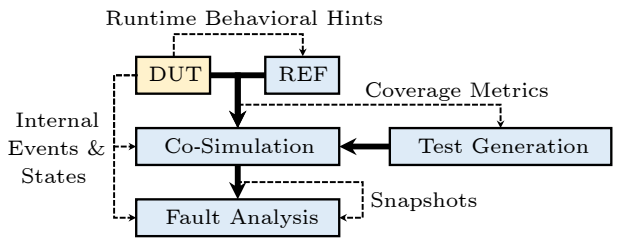


Fig.3. Integrated agile design and verification workflows.

Collaborative task delegation emphasizes the collaborative relationship between development stages. It involves delegating tasks from one stage to another, recognizing that stages can work on behalf of each other to accomplish certain tasks. For example, besides the source code deliverable, the DUTs are also required to annotate and expose their internal events and states utilized by the following stages such as co-simulation. By leveraging collaborative task delegation, the overall development productivity is improved.

Dynamic information exchange focuses on the real-time exchange of information across development stages. It ensures that information is shared not only before or after each stage but also during the stages themselves. For example, the REF leverages the DUT's runtime behaviors as its execution hints to simplify its design. This dynamic exchange of information enables timely decision-making and facilitates agility in responding to changing requirements.

These two principles play a vital role in supporting and facilitating workflow integration within the context of agile development methodologies, as they provide a framework for identifying opportunities and optimizing the agile development workflows.

For example, with emerging toolchains like HCLs, agile development enables iterative and frequent design changes in the design–design (D–D) workflow. While increasing the design efficiency, they also provide more design information that the verification stage could utilize to foster agility in the design–verification (D–V) and verification–verification (V–V) workflows. Following the two design principles of workflow integration, the proposed workflows and toolchains can identify, transfer, and utilize useful information throughout the entire hardware development process.

In this paper, we further provide practical examples for workflow integration by presenting novel methodologies and toolchains for various stages in the functional verification of RISC-V processors. Fig.4 shows an overview of the proposed techniques and workflows.

As Fig.3 previously illustrates, the proposed techniques showcase the prevalent design paradigm for agile development tools of workflow integration with collaborative task delegation and dynamic information exchange. While the proposed tools are only demonstrated with RISC-V processors, they also hold promise for application to general hardware designs.

They optimize various stages in functional verification, including REFs in Subsections 4.2 and 4.3, co-simulation in Subsection 4.4, test generation in Subsection 4.5, and fault analysis in Subsection 4.6.

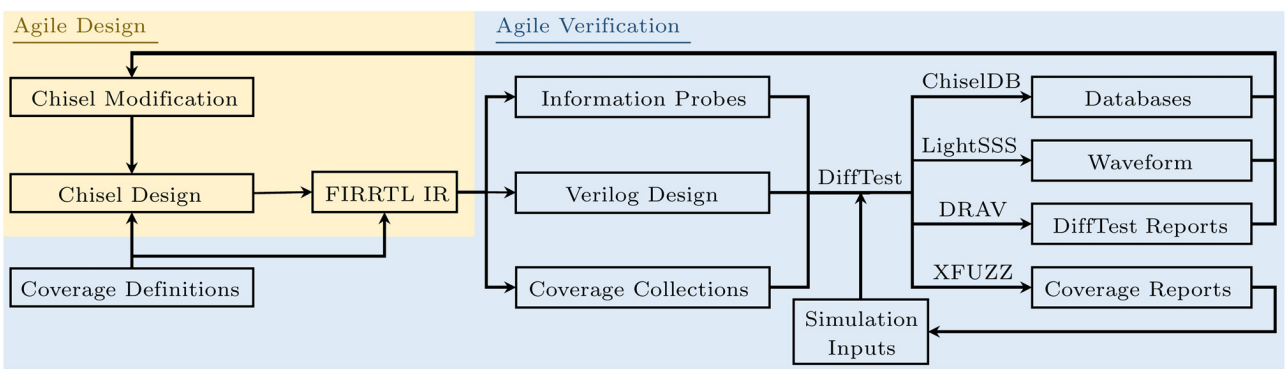


Fig.4. Overview of the proposed agile hardware design and functional verification workflows.

4.2 Diff-Rule Based Agile Verification (DRAV)

The instruction set architecture (ISA) defines the standard software-hardware interface, and thus every RISC-V processor must comply with the RISC-V ISA manuals. To represent the ISA compliance in a machine-readable way, people have built RISC-V golden models either formally or in the form of an instruction set simulator (ISS). Practically, these golden ISA models are selected as REFs for functional verification of RISC-V processors.

When co-simulating RISC-V processors under test with the REFs, architectural registers are usually selected as indicators for the program behavior in co-simulation, such as the general-purpose registers and the program counter. When the DUT outputs a different register value from that of the REF, the co-simulation framework will report a potential bug.

However, in reality, the actual ISS used as the REF implements only one or several simplified variants of the ISA model. There will be scenarios where designs are legally allowed by the ISA to have diverse architectural behaviors, such as the timing of handling asynchronous external interrupts. In an in-order five-stage processor, the instruction at the decode stage typically services the interrupt. By contrast, XiangShan always raises interrupts along with the oldest interrupt-safe instruction. However, an ISS without detailed pipelines responds to an interrupt immediately on the next unfinished instruction. If we co-simulate the ISS with either a five-staged pipelined processor or XiangShan, a mismatch will be reported after the interrupt is serviced. This is a false positive result because the asynchronous behaviors of all three implementations are ISA-compatible.

As shown in the example, though compatible with RISC-V, different designs still appear to behave nondeterministically even with the same stimuli. The diverse behaviors are caused mainly by microarchitectural implementation details and probably result in false positives during co-simulation.

Therefore, considering the diversity an ISA would allow, any co-simulation mechanism for processor functional verification must address the issue of behavioral nondeterminism.

As discussed in Section 3, processors are conventionally designed and verified for a long time. The problem can be tackled by setting up detailed REFs specifically targeted at certain design parts to thoroughly eliminate any nondeterministic possibilities. This fine-grained and tedious verification strategy ensures a 1-to-1 correspondence between the DUT and the REF.

However, despite its usefulness for designs with a long production life-cycle, maintaining the 1-to-1 correspondence between the DUT and the REF can be difficult with a short development cycle and rapid feature changes, commonly seen in the agile development model.

Rethinking the principle of verification, a given design specification, such as the RISC-V manuals, can lead to diverse implementations. Once the behavior of the DUT satisfies the definition of specifications, the implementation details are allowed to be diverse, and the checking between the DUT and the REF in co-simulation could be conditionally relaxed.

Following this principle, it is feasible that DUTs with different implementation details can be verified with the same REF if their behaviors conform to the same specification. This forms an N-to-1 correspondence between DUTs and the REF. Since only one simpler REF is maintained, the functional verification overhead for multiple processors is reduced.

Based on the observations, we propose a novel diff-rule based agile verification mechanism DRAV. The diff-rules abstract legal behaviors defined in specifications with the consideration of nondeterminism in functional verification. DRAV identifies the sources and indicators of nondeterminism (diff-rules), transfers the behavioral hints from the DUT to the REF, and refines the REF on-the-fly to align with the DUT. By utilizing runtime execution information from the designs, REFs no longer require fine-grained implementation details and could be simplified. We enhance the conventional verification by providing both DRAV methodology and an implementation for RISC-V processors.

4.3 DRAV for RISC-V Processors

DiffTest is a co-simulation based verification framework for RISC-V processors that accelerates functional verification by adopting the DRAV methodology. It provides the flexibility to add diff-rules and adaptively reconfigure the reference model, thus being scalable to support multiple designs dynamically.

The key challenge of devising diff-rules is identifying sources of nondeterministic behaviors in the RISC-V architecture. In this subsection, we introduce

in detail several representative sources of nondeterminism in multicore high-performance RISC-V processors.

4.3.1 *Speculative Virtual Address Translations*

In the RISC-V Linux kernel, the operating system chooses not to execute a memory-barrier instruction after allocating a new physical page to avoid flushing instructions until a page fault exception. In most cases, the in-memory page table entries (PTEs) are updated quickly after the retirement of the store instructions. However, the store operation that updates a PTE may not take effect when the TLB accesses the memory, possibly due to the existence of a store queue or a store buffer. In that case, a memory instruction accessing the page will trigger a page fault exception.

Two diff-rules are involved in addressing this issue: 1) the DUT may trigger a page fault exception even if the REF does not trigger; 2) the DUT and the REF should have the same architectural states after executing the same instruction. If the DUT reports a page fault exception, this event serves as a hint to the REF and notifies the REF to conditionally report a page fault exception as well, even if the REF succeeds in the page table walker with a valid page table entry.

4.3.2 *Cache Hierarchy and Multicore Scenarios*

Under the RISC-V weak memory order (RVWMO) model, load and store instructions may have an exponential interleaving space of concurrent memory accesses. We leverage diff-rules to prune the astronomically large interleaving space and co-simulate multicore processors against simple single-core REFs.

For example, load instructions can bypass the value from the private store buffer first and access the global memory if the bypass fails. A naive multicore co-simulation requires maintaining a correct store buffer in the REF unless disagreements between the DUT and the REF will frequently abort the co-simulation.

In DiffTest, we devise a diff-rule from the RVWMO specification that allows DUTs to maintain the global memory and updates the REF memory when they disagree. It introduces the Global Memory (GM) that records the store requests that enter the cache hierarchy in the DUT. Data correctness is checked by both GM and the local memory of single-core REFs. When the single-core REF executes a load with a different value from the DUT, DiffTest accesses the same load address in GM to check whether other hardware threads possibly write this load value. If so, the value will be updated to both the local memory and the destination register of the load instruction in the single-core REF.

4.3.3 *More Sources of Nondeterminism*

We provide additional examples and categorize the diff-rules for RISC-V processors based on the sources and effects of nondeterminism, as listed in Table 1.

Table 1. Classifications of Diff-Rules for RISC-V Processors

Category	Source of Nondeterminism
Static	Implementation-dependent registers
Dynamic	Asynchronous events, speculative execution, weak memory models, hardware timing

On the one hand, a large portion of nondeterministic behaviors can be handled with static implementation-defined configurations of the REFs.

Taking the control and status registers (CSRs) in RISC-V as an example, the supervisor address translation and protection (`satp`) register has a field called `PPN` to represent the physical page number of the root page table. `satp.PPN` can have platform-dependent constraints on its values and thus may hold only valid physical page numbers. Depending on the physical address width, `satp` would have different write masks in various implementations. However, this implementation-dependent behavior does not rely on runtime DUT information and can be statically configured in the REF.

Similarly, we investigate the RISC-V privilege specification and identify at least 120 diff-rules for machine-mode CSRs. However, since these diff-rules do not rely on any runtime information, they are the most straightforward cases and are classified as static diff-rules.

On the other hand, we propose the dynamic diff-rules and further distinguish primary runtime sources of nondeterminism with the root causes.

We classify them into at least four categories. First, asynchronous events cannot be predicted by the REFs, such as the service of external and timer interrupts mentioned in Subsection 4.2. Second, speculative execution and pervasive buffering in high-performance processor designs lead to different architec-

tural behaviors from in-order designs, such as the page fault example discussed in Subsection 4.3.1. Third, as shown in Subsection 4.3.2, weak memory models usually allow a huge nondeterministic program interleaving space, resulting in difficulties in accurately modeling the caches. Moreover, hardware timing and microarchitecture optimizations may also bring nondeterminism. For example, RISC-V allows the load-reserved instruction to adopt an implementation-defined timeout before the store-conditional instruction executes, which can never be mimicked by behavioral ISA models.

It is worth noting that the attempt to cover more diff-rules of processors is never enough, since the increase of rule-checking coverage only increases the confidence of correctness but never ensures the correctness. Instead, agility addresses the verification issue by increasing the workflow efficiency and thus allows more verification trials within a fixed time interval.

Overall, DiffTest leverages the DRAV and diff-rules to verify the RISC-V processors. It offers systematical support for handling co-simulation nondeterminism and significantly reduces the efforts of building correct REFs for complex processors.

4.4 Information Probes

Towards the second challenge of co-simulation discussed in Section 3, we further decompose DiffTest into diff-rule checkers and information probes. The information probes are embedded into processor designs and automatically carry the required information for verification. In this subsection, we discuss the design methodologies of probes and how they bridge the co-simulation workflow with HCL designs.

We observe that functional verification requires only some design information with stable structures across designs. Specifically, we identify two major categories, including architectural events like instruction commits and architectural states like register values. The probes describe them using pre-defined bundles in the Chisel HCL. The bundles are then instantiated in processor designs and highlight the necessary information for other workflows like functional verification.

In the Chisel elaboration stage, the compiler will generate both the corresponding Verilog and structured C++ descriptions of the information. They form the uniform interfaces between the DUT and the REF and are further utilized by the co-simulation framework.

During RTL-simulation in the functional verification process, the extracted information can be used for online rule checking of co-simulation and collected for further debugging at post-simulation stages. For example, by creating SQL tables from the generated C++ classes, ChiselDB is able to automatically record the semantic information and provide elegant SQL interfaces for Chisel designers to visualize the transactions for debugging and analyzing.

While diff-rules bridge the DUTs and REFs with runtime behaviors and states, information probes effectively carry the internal design information to verification toolchains with standardized interfaces. DiffTest with diff-rules and information probes enables universal functional verification workflows with high agility for diverse RISC-V processor designs.

4.5 Test Generation

As mentioned in Section 3, many test generation techniques rely on handcrafted, design-specific knowledge and may need to be better suited for agile development. In this subsection, we propose and introduce an efficient yet generic test generation approach.

The proposed tool, XFUZZ, is a coverage-guided mutational fuzzing technique. Fuzzing aims at finding design bugs by injecting arbitrary but fast-running inputs to the DUT, and recent fuzzing studies have been using the coverage feedback to guide the generation of higher-quality inputs[14-16]. While software fuzzers usually use system crashes as the target, hardware designs generally do not crash. Therefore, XFUZZ integrates the software fuzzing library LibAFL[17] into DiffTest and monitors the hardware assertions and co-simulation mismatches for potential bugs. By observing that RISC-V processors follow the standard stored-program paradigm, XFUZZ mutates and generates the initial memory contents as the uniform fuzzing input for all processor designs.

However, despite being widely adopted, Chisel still lacks native coverage support and relies on open-source and commercial RTL simulators to collect coverage metrics. We harness the benefits of high-level HCLs and develop user-friendly XFUZZ workflows for coverage definition and collection.

First, we design a circuit annotation on the FIRRTL intermediate representation[18] with a transform to create corresponding Verilog and C++ interfaces.

This coverage annotation enables developers to design customized functional and code coverage metrics. Second, with the annotation, we implement new instruction and instruction-immediate coverage metrics with the decoders of RISC-V processors and also port the existing MUX[14] and control register[15] coverage. Third, as one of the circuit optimization passes in the Chisel compiler, we propose the coverage FIRRTL transform that collects and converts the annotations to DPI-C calls in Verilog and C++ classes with statistical information. This generated information is used for coverage feedback of the fuzzing process and quantitative reports of the functional verification process.

By integrating XFUZZ into DiffTest, we have developed a fully automated functional verification pipeline for RISC-V processors, shown in the bottom half of Fig.4. Starting with a seed collection of test cases and coverage points generated by the DUT, XFUZZ continuously mutates inputs with coverage feedback, feeds inputs into the DUT, and monitors the simulation results. As far as DiffTest identifies no faults, the fuzzing iterations continue, and the coverage metrics are expected to increase. This workflow provides a generic and efficient method for verifying RISC-V processors with minimal human effort. We will demonstrate its effectiveness in Subsection 5.2.

4.6 Fault Analysis

Analyzing and debugging the reported faults require information such as the waveform that significantly slows down the simulation speed. For example, the simulation speed for XIANGSHAN when waveform is enabled drops to about 8.5% of the normal speed.

To enhance the efficiency of debugging processes, it is common to focus on the region of interest (ROI), the last seconds of RTL-simulation[19]. This is the time period in which design bugs are more likely to be triggered, and their effects are propagated to the visible output signals. By enabling waveform only during the ROI, this approach prevents spending excessive time and resources on analyzing unnecessary information and allows focusing on resolving the root cause of the problem, significantly improving the overall simulation throughput. Nevertheless, accurately predicting the termination of the simulation due to reported functional bugs remains a significant challenge.

To address this problem, instead of forward prediction, a recent study[20] proposes an alternative approach for taking snapshots of the simulation environment and restoring recent snapshots after simulation abortion for reproducing the ROI. However, the presented simulation snapshot technique LiveSim incurs a considerable overhead for snapshotting, ranging from 10% to 20% of the overall simulation performance.

In this paper, we tackle the performance issues by simplifying the contents of snapshots and propose the Lightweight Simulation Snapshot (LightSSS).

LightSSS considerably reduces the overhead of snapshots while still allowing debugging the ROI with far less resource waste. First, it preserves only the two most recent snapshots in memory and drops any outdated, thus useless ones. Second, instead of creating complete snapshots, it records in memory only the different contents between the last snapshot and the current status. Furthermore, it generates process-level snapshots with operating systems independent of the simulated circuit structure.

LightSSS is integrated into the simulation-based functional verification workflow and implemented using the highly efficient `fork`() system call of the Linux kernel. During RTL-simulation, LightSSS periodically calls `fork`() and treats the forked process as a snapshot. The forked process waits for the parent process to exit abnormally, possibly because DiffTest detects a potential functional bug. Meanwhile, as the parent process continues and updates the memory, the copy-on-write (COW) forking strategy inherently creates incremental snapshots: the operating system allocates physical pages for only the modified pages, leaving unmodified pages shared between processes. LightSSS allows the existence of at most two snapshots and kills older snapshots to avoid consuming any more resources.

With the COW mechanism, performance overhead by LightSSS comes from two sources: the `fork`() system call and copying of the modified pages between the parent and child processes. On the one hand, as the hardware design size increases, the `fork`() overhead is not affected, but the number of modified pages increases. However, the increased snapshot overhead may be negligible, given that larger designs generally simulate much slower than smaller designs. On the other hand, the overall performance overhead is also impacted by the frequency of taking snapshots. Compared with smaller snapshot intervals, a larger snapshot interval reduces the number of `fork`() system calls. Besides, since the COW

strategy copies pages on demand only when a page is modified by the parent or child process for the first time, longer snapshot intervals reduce the overhead by copying modified pages only once during a longer period. Therefore, as the snapshot interval increases, the overall overhead of copying modified pages decreases. We will quantitatively evaluate the performance overhead of LightSSS on different designs with various snapshot intervals later in Subsection 5.3.

As a plugin for the DiffTest framework, LightSSS provides a lightweight approach to verifying the functionalities and analyzing the reported faults. The co-simulation mechanism naturally serves as the indicator for restoring snapshots and enables a continuous workflow without human intervention.

5 Evaluation

To showcase how the proposed tools facilitate more agile workflows, we will employ them to develop and verify two representative RISC-V processor designs: the in-order processor NutShell[21] and the high-performance out-of-order processor XiangShan[22].

5.1 DiffTest on RISC-V Processors

DiffTest is a co-simulation framework supporting multiple RISC-V processors by composing diff-rules and information probes. As discussed in Subsection 4.3, the diff-rules are consistent and thus reused across different RISC-V designs. In this subsection, we further demonstrate the applicability of information probes on NutShell and XiangShan.

As shown in Table 2, DiffTest currently implements 15 information probes, and 10 of them are optional. These probes are mainly classified as events and states, providing standardized interfaces for verifying the RISC-V architectural functionalities.

Since NutShell only supports the single-core configuration, it does not instantiate probes used for only multicore co-simulation, such as the `RefillEvent` for synchronizing the global memory. These unused probes are detected during the Chisel elaboration process and result in undefined macros, which would comment out the corresponding diff-rule checkers in compile time.

In contrast, XiangShan implements a superscalar out-of-order microarchitecture and instantiates more probes multiple times. The number of instantiations will also be collected during the elaboration and passed to the rule checkers as macros to guide the iterations of checking.

To summarize, by using the probes as basic building blocks for design information sharing, DiffTest provides a uniform interface for co-simulating RISC-V processors against the reference models. Next, we evaluate the test generation and fault analysis plugins for DiffTest, namely XFUZZ and LightSSS respectively.

5.2 Fuzzing Effectiveness

To demonstrate the effectiveness of the XFUZZ fuzzing framework introduced in Subsection 4.5, we use it to generate test cases for NutShell with both functional and code coverage metrics as feedback. As a five-stage pipelined RISC-V processor, NutShell is capable of running modern operating systems such as

Table 2. Information Probes for DiffTest and Their Number of Instantiations in Single-Core NutShell and Dual-Core XiangShan

Probe Name	Description	Mandatory	NutShell	XiangShan
`ArchEvent`	Exceptions and interrupts	Yes	1	2
`InstrCommitEvent`	Executed instructions	Yes	1	12
`TrapEvent`	Simulation environment call	Yes	1	2
`CSRState`	Control and status registers	Yes	1	2
`DebugModeState`	Debug mode registers	No	0	2
`ArchIntRegState`	General-purpose registers	Yes	1	2
`ArchFpRegState`	Floating-point registers	No	0	2
`IntWritebackEvent`	General-purpose writeback operations	No	1	16
`FpWritebackEvent`	Floating-point writeback operations	No	0	16
`StoreEvent`	Store operations	No	1	4
`SbufferEvent`	Store buffer operations	No	0	4
`LoadEvent`	Load operations	No	0	12
`AtomicEvent`	Atomic operations	No	0	2
`RefillEvent`	Cache refill operations	No	0	4
`LrScEvent`	Executed LR/SC instructions	No	0	2

Debian and complicated CPU benchmarks like SPEC CPU2006. Despite its design simplicity and trusted functionalities, we aim at detecting more escaped functional bugs of NutShell using XFUZZ.

XFUZZ is compared with an existing hardware fuzzing approach called HWFP proposed by Google[16] that uses only software fuzzers with C++ branch coverage feedback from the generated simulation binary. In contrast, XFUZZ adopts and provides support for both functional coverage metrics, including the instruction and instruction-immediate coverage, and code coverage metrics, including the MUX[14] and control register[15] coverage.

In total, XFUZZ finds 33 and 18 exclusive functional bugs of NutShell and the reference model respectively. These exclusive bugs escape HWFP with only software coverage feedback but are detected by XFUZZ with hardware coverage feedback. Therefore, they showcase the necessity and superiority of using hardware coverage metrics for fuzzing over only software coverage of the simulation binary.

Table 3 lists 13 representative bugs for arithmetic operations, CSR operations, and access control. While the arithmetic bugs may result in malformed functionalities of well-behaved programs, bugs in CSR operations and access control are seldom touched and thus hardly detected by real-world workloads. As the bugs have not been previously discovered, the test cases generated by XFUZZ are demonstrated to be complementary to the real-world workloads originally used by NutShell developers.

To conclude, XFUZZ provides built-in support for customized hardware coverage metrics and finds more bugs that escape HWFP. By integrating fuzzing with co-simulation in DiffTest, XFUZZ is effective and valuable for functional verification of RISC-V processors.

5.3 Snapshot Efficiency

In this subsection, we evaluate the performance overhead of LightSSS and demonstrate its efficiency in creating snapshots for RTL-simulation using two designs of varying sizes as the benchmarks, including single-core NutShell and XiangShan.

Both designs are simulated with two workloads: running the CoreMark benchmark and booting a Linux kernel to user mode. We use Verilator to simulate the RTL designs, with simulation speeds of 843.7 KHz for NutShell (single thread, 58.8k lines of C++ generated by Verilator) and 3.3 KHz for XiangShan (8 threads, 15.2M lines of C++). Though it takes only 10 seconds and five minutes for NutShell to run CoreMark and boot Linux, XiangShan is much more complex and reaches higher performance, thus finishing the workloads in eight and 18 minutes, respectively.

As discussed in Subsection 4.6, the performance overhead of LightSSS, including the `fork()` system calls and on-demand copying of modified pages, is affected by both design sizes and snapshot intervals.

Fig.5 demonstrates the overall results on NutShell and XiangShan with various snapshot intervals. While we increase the snapshot interval from one second to 60 seconds, minor performance deviation is observed on both designs. The maximum performance overhead is caused by a snapshot interval of one second for NutShell, still less than 1% of the normal RTL-simulation speed. Overall, this result demonstrates that LightSSS introduces an order of magnitude lower overhead than the state-of-the-art

Table 3. Representative Design Bugs of NutShell That Escape the AFL++ Fuzzing But Are Detected by XFUZZ

Category	Bug ID	Description
Arithmetic	`769669f`	32-bit AMO instructions do not sign-extend the 32-bit operands
CSR operations	`42c8460`	Reserved and non-writable fields in `mstatus` may be written by CSR instructions
	`ef78025`	The virtual address is not zero-extended to 64-bit if virtual memory is disabled
	`5b60e9c`	`mtval`/`stval` is incorrectly updated without considering the exception delegation
	`b86c319`	`mstatus.mprv` is not cleared when `MRET`/`SRET` to a mode less privileged than `M`
	`6f4cd05`	`mstatus.mpp` is updated and read with an illegal value (`ModeH`)
Access control	`54367ce`	An illegal jump target causes mistakenly executed load/store operations
	`f23acbf`	Misaligned `LR`/`SC` operations are not detected as address-misaligned exceptions
	`5dd6a74`	Non-existent CSRs such as `pmpcfg1` and `pmpcfg3` are enabled in RV64
	`ccd9c7f`	`CSRRC`/`CSRRCI` causes write side effects when `rs1` $=$ `x0` or `uimm`$[4:0] = 0$
	`7f928a3`	`SC` incorrectly updates the reservation sets that should be set by `LR` only
	`f8acb2a`	`mstatus.TVM` does not intercept supervisor virtual-memory management operations
	`c508b32`	Large pages with misaligned PPNs are not detected as page-fault exceptions

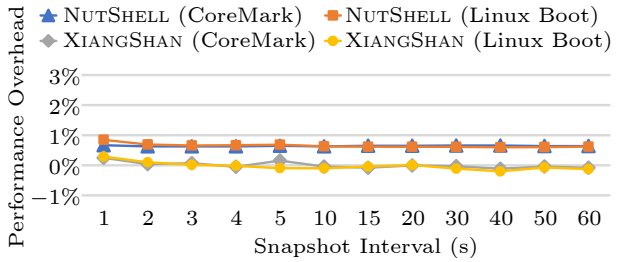


Fig.5. Performance overhead of LightSSS.

LiveSim[20], which reports a 10%–20% performance overhead.

Therefore, by integrating with DiffTest and creating snapshots for the RTL-simulation process with minimum overhead, LightSSS enables quick access to the region of interest and an agile debugging workflow.

5.4 Case Study

In this subsection, we further demonstrate the effectiveness and efficiency of proposed workflows with a functional bug found in the L2 cache of XiangShan.

This complicated bug is exposed when running the Redis benchmark on dual-core XiangShan for over 168 hours after three billion simulated cycles. DiffTest reports a data mismatch between DUT and the Global Memory using the diff-rule described in Subsection 4.3.2. After the co-simulation detects the functional bug, we obtain two types of debugging information for fault analysis.

We start with the behavioral database created by ChiselDB. As introduced in Subsection 4.4, it enables recording the transaction information across the multi-level caches. Reviewing the high-level behaviors of L2 and L3 caches, we find that a TileLink Acquire request from the L2 cache to L3 overlaps with a Probe transaction from L3 to L2 in the same cache block. Though L2 acquires the correct data from L3, later, it grants the wrong data upward to L1, indicating a potential bug in the arbitration logic of the L2 cache.

To further investigate the internal design of the L2 cache, we turn to the waveform generated by LightSSS. As presented in Subsection 4.6, after DiffTest reports the mismatch and while we are reviewing the database, LightSSS activates the second to the last simulation snapshot to re-run the RTL-simulation for the last seconds. It takes only three minutes to simulate the last 30.8k cycles with waveform enabled. Further investigation confirms that L2 MSHR does not handle the overlapping correctly when Probe and GrantData from L3 cache arrive at a specific time interval.

By integrating the proposed tools, we provide an end-to-end workflow for processor developers to design and verify their processor designs. As shown in Fig.3, DRAV and DiffTest share the runtime information of DUTs with REFs and refine the REFs on-the-fly, significantly simplifying the design of REFs. Information probes are initially implanted into DUTs and effectively highlight valuable information for later development stages, e.g., co-simulation, debugging, and performance analysis. By retrieving and utilizing coverage metrics, XFUZZ provides an unceasing workflow to explore the input space deeply and verify the design functionalities. LightSSS monitors the co-simulation status, periodically creates snapshots, and quickly restores the region of interest for debugging. These tools enable chip beginners and non-experts to automate their work and improve development efficiency.

For example, in this case study, without the integration of LightSSS, it may take extra 16 hours to create and restore the simulation snapshots with LiveSim, or even 168 hours to re-run the simulation without any snapshot techniques. Similarly, with information probes and ChiselDB, we can analyze the detected mismatch with high-level behaviors and better debugging efficiency. Overall, this case study showcases the superior efficiency and effectiveness of the proposed techniques and tools.

6 Discussions and Related Work

In recent years, agile chip development methods have garnered attention from both academia and industry. To meet the massive demand for emerging applications and domain-specific architectures, the agile hardware development model pursues fast response and continuous delivery capabilities through optimizations of tools and workflows[2, 3, 13].

As a notable transition from the conventional development model, the adoption of high-level hardware construction languages (HCLs) accelerates the design stage for agile development by enabling hardware reusability, parametrization, and abstraction. In contrast to the traditional hardware description languages (HDLs) with only basic primitives such as `parameter` and `struct`, modern HCLs usually support built-in automation for advanced design paradigms via host languages and compilers[8, 23].

Specifically, Chisel adopts the FIRRTL[18] intermediate representation in compilers and supports customized transformations of the circuits based on FIRRTL. While we present our workflows on Chisel and FIRRTL only, the design methodologies of the proposed tools also apply to other HCLs, and techniques like LightSSS are theoretically independent of the underlying languages and simulators.

In addition to the progress for hardware construction, agile chip development also requires the improvement of verification tools. Software-based simulation techniques have been enhanced for higher throughput for both conventional HDLs[24] and emerging HCLs[25]. Due to the significantly faster clock rate and reconfiguration flexibility, FPGA-accelerated simulation has become popular in recent years, and researchers have presented many tools to address the issues of debuggability[19] and scalability[11]. One of our future studies is to adopt FPGAs to further accelerate the proposed functional verification workflows, such as FPGA-accelerated co-simulation[6, 26].

Despite significant advancements in simulation and debugging acceleration, the most challenging aspect of hardware verification lies in the comprehensive exploration of the design space and identifying potential bugs.

Formal and static approaches are promising towards this objective, thus being actively studied to address the scalability issues and continually proposed for hardware designs[27]. Dynamic simulation-based workflows are still popular in practice. Various testing strategies have been put forward to improve the verification effectiveness, such as property-based testing[28] and coverage-directed test generation[29]. Besides, with the increasing recognition of agile and open-source hardware, there has been a promising proliferation of domain-specific functional verification tools for RISC-V processors[15] and accelerators[30]. With partial reliance on domain-specific knowledge, they present superior tradeoffs between general applicability and effectiveness, which is also important future work for us.

To enable agile development, which emphasizes development efficiency, and functional verification, which demands high levels of effectiveness and quality, both aspects must be carefully considered. Looking toward the future, this paper suggests improving the efficiency and effectiveness of functional verification by further incorporating agile development paradigms and tools. There have been some studies adopting similar strategies. For example, since the hardware design is iteratively refined under the agile model, functional verification can focus on the modified code instead of the whole design[31]. High-level HCLs can also be utilized for formal verification[10], parameterized verification[32], and instrumentation of coverage metrics[33].

7 Conclusions

In this paper, we systematically investigated the challenges that the agile development methodology poses on the conventional functional verification workflows. We proposed workflow integration with collaborative task delegation and dynamic information exchange as the fundamental design principles of agile development toolchains. We enhanced the functional verification toolchains for RISC-V processors and presented the integrated workflows for agile hardware development. The proposed toolchains are quantitatively evaluated by designing and verifying two RISC-V processors. We found 33 functional design bugs and demonstrated the effectiveness of the presented workflows and toolchains.

Acknowledgements The authors would like to thank Zhi-Wei Xu for his valuable insights into the idea. The proposed techniques in this paper have been extensively used by the XiangShan team. We give special thanks to our team members for their feedback and help on the work.

Conflict of Interest The authors declare that they have no conflict of interest.

References

[1] Hennessy J L, Patterson D A. A new golden age for computer architecture. *Communications of the ACM*, 2019, 62(2): 48–60. DOI: 10.1145/3282307.

[2] Bao Y G, Carlson T E. Agile and open-source hardware. *IEEE Micro*, 2020, 40(4): 6–9. DOI: 10.1109/MM.2020.3002606.

[3] Lee Y, Waterman A, Cook H, Zimmer B, Keller B, Puggelli A, Kwak J, Jevtic R, Bailey S, Blagojevic M, Chiu P F, Avizienis R, Richards B, Bachrach J, Patterson D, Alon E, Nikolic B, Asanović K. An agile approach to building RISC-V microprocessors. *IEEE Micro*, 2016, 36(2): 8–20. DOI: 10.1109/MM.2016.11.

[4] Xu Y N, Yu Z H, Tang D, Chen G K, Chen L, Gou L R, Jin Y, Li Q R, Li X, Li Z J, Lin J W, Liu T, Liu Z G, Tan J Z, Wang H Q, Wang H Z, Wang K F, Zhang C Q, Zhang F W, Zhang L J, Zhang Z F, Zhao Y Y, Zhou Y Y, Zhou Y K, Zou J R, Cai Y, Huan D D, Li Z S, Zhao J Y, Chen Z H, He W, Quan Q Y, Liu X, Wang S, Shi K,

Sun N H, Bao Y G. Towards developing high performance RISC-V processors using agile methodology. In *Proc. the 55th IEEE/ACM International Symposium on Microarchitecture*, Oct. 2022, pp.1178–1199. DOI: 10.1109/MICRO56248.2022.00080.

[5] Xu Y N, Yu Z H, Tang D, Cai Y, Huan D D, He W, Sun N H, Bao Y G. Toward developing high-performance RISC-V processors using agile methodology. *IEEE Micro*, 2023, 43(4): 98–106. DOI: 10.1109/MM.2023.3273562.

[6] Kabylkas N, Thorn T, Srinath S, Xekalakis P, Renau J. Effective processor verification with logic fuzzer enhanced co-simulation. In *Proc. the 54th Annual IEEE/ACM International Symposium on Microarchitecture*, Oct. 2021, pp.667–678. DOI: 10.1145/3466752.3480092.

[7] IEEE. IEEE standard for universal verification methodology language reference manual. *IEEE Std 1800.2-2020*, 2020, pp.1–458. DOI: 10.1109/IEEESTD.2020.9195920.

[8] Bachrach J, Vo H, Richards B, Lee Y, Waterman A, Avižienis R, Wawrzynek J, Asanović K. Chisel: Constructing hardware in a Scala embedded language. In *Proc. the 49th Annual Design Automation Conference*, Jun. 2012, pp.1216–1225. DOI: 10.1145/2228360.2228584.

[9] Beamer S, Donofrio D. Efficiently exploiting low activity factors to accelerate RTL simulation. In *Proc. the 57th ACM/IEEE Design Automation Conference*, July. 2020. DOI: 10.1109/DAC18072.2020.9218632.

[10] Yu S Z, Dong Y F, Liu J Y, Li Y, Wu Z L, Jansen D N, Zhang L J. CHA: Supporting SVA-like assertions in formal verification of chisel programs (tool paper). In *Proc. the 20th International Conference on Software Engineering and Formal Methods*, Sept. 2022, pp.324–331. DOI: 10.1007/978-3-031-17108-6_20.

[11] Karandikar S, Mao H, Kim D, Biancolin D, Amid A, Lee D, Pemberton N, Amaro E, Schmidt C, Chopra A, Huang Q J, Kovacs K, Nikolic B, Katz R, Bachrach J, Asanović K. FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud. In *Proc. the 45th ACM/IEEE Annual International Symposium on Computer Architecture*, July. 2018, pp.29–42. DOI: 10.1109/ISCA.2018.00014.

[12] Kern C, Greenstreet M R. Formal verification in hardware design: A survey. *ACM Trans. Design Automation of Electronic Systems*, 1999, 4(2): 123–193. DOI: 10.1145/307988.307989.

[13] Asanović K, Avizienis R, Bachrach J, Beamer S, Biancolin D, Celio C, Cook H, Dabbelt D, Hauser J, Izraelevitz A, Karandikar S, Keller B, Kim D, Koenig J, Lee Y, Love E, Maas M, Magyar A, Mao H, Moreto M, Ou A, Patterson D A, Richards B, Schmidt C, Twigg S, Vo H, Waterman A. The rocket chip generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, 2016. https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html, July. 2023.

[14] Laeufer K, Koenig J, Kim D, Bachrach J, Sen K. RFUZZ: Coverage-directed fuzz testing of RTL on FPGAs. In *Proc. the 2018 IEEE/ACM International Conference on Computer-Aided Design*, Nov. 2018. DOI: 10.1145/3240765.3240842.

[15] Hur J, Song S, Kwon D, Baek E, Kim J, Lee B. DifuzzRTL: Differential fuzz testing to find CPU bugs. In *Proc. the 42nd IEEE Symposium on Security and Privacy*, May 2021, pp.1286–1303. DOI: 10.1109/SP40001.2021.00103.

[16] Trippel T, Shin K G, Chernyakhovsky A, Kelly G, Rizzo D, Hicks M. Fuzzing hardware like software. In *Proc. the 31st USENIX Security Symposium*, Aug. 2022, pp.3237–3254.

[17] Fioraldi A, Maier D C, Zhang D J, Balzarotti D. LibAFL: A framework to build modular and reusable fuzzers. In *Proc. the 2022 ACM SIGSAC Conference on Computer and Communications Security*, Nov. 2022, pp.1051–1065. DOI: 10.1145/3548606.3560602.

[18] Izraelevitz A, Koenig J, Li P, Lin R, Wang A, Magyar A, Kim D, Schmidt C, Markley C, Lawson J, Bachrach J. Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations. In *Proc. the 2017 IEEE/ACM International Conference on Computer-Aided Design*, Nov. 2017, pp.209–216. DOI: 10.1109/ICCAD.2017.8203780.

[19] Kim D, Celio C, Karandikar S, Biancolin D, Bachrach J, Asanović K. DESSERT: Debugging RTL effectively with state snapshotting for error replays across trillions of cycles. In *Proc. the 28th International Conference on Field Programmable Logic and Applications*, Aug. 2018, pp.76–764. DOI: 10.1109/FPL.2018.00021.

[20] Skinner H, Trapani Possignolo R, Wang S H, Renau J. LiveSim: A fast hot reload simulator for HDLs. In *Proc. the 2020 IEEE International Symposium on Performance Analysis of Systems and Software*, Aug. 2020, pp.126–135. DOI: 10.1109/ISPASS48437.2020.00028.

[21] Wang H, Zhang Z, Jin Y, Zhang L, Wang K. Nutshell: A Linux-compatible RISC-V processor designed by undergraduates. https://riscv.org/proceedings/2020/09/risc-v-global-forum-proceedings/, July 2023.

[22] Wang K F, Xu Y N, Yu Z H, Tang D, Chen G K, Chen X, Gou L R, Hu X, Jin Y, Li Q R, Li X, Lin J W, Liu T, Liu Z G, Wang H Q, Wang H Z, Zhang C Q, Zhang F W, Zhang L J, Zhang Z F, Zhang Z Y, Zhao Y Y, Zhou Y Y, Zou J R, Cai Y, Huan D D, Li Z S, Zhao J Y, He W, Sun N H, Bao Y G. XiangShan open-source high performance RISC-V processor design and implementation. *Journal of Computer Research and Development*, 2023, 60(3): 476–493. DOI: 10.7544/issn1000-1239.202221036. (in Chinese)

[23] Lockhart D, Zibrat G, Batten C. PyMTL: A unified framework for vertically integrated computer architecture research. In *Proc. the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2014, pp.280–292. DOI: 10.1109/MICRO.2014.50.

[24] Wang H Y, Beamer S. RepCut: Superlinear parallel RTL simulation with replication-aided partitioning. In *Proc. the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2023, pp.572–585. DOI: 10.1145/3582016.3582034.

[25] Jiang S N, Ilbeyi B, Batten C. Mamba: Closing the per-

formance gap in productive hardware development frameworks. In *Proc. the 55th ACM/ESDA/IEEE Design Automation Conference*, Jun. 2018. DOI: 10.1109/DAC.2018.8465576.

[26] Shi K, Xu S X, Diao Y H, Boland D, Bao Y G. ENCORE: Efficient architecture verification framework with FPGA acceleration. In *Proc. the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Feb. 2023, pp.209–219. DOI: 10.1145/3543622.3573187.

[27] Xing Y, Lu H X, Gupta A, Malik S. Leveraging processor modeling and verification for general hardware modules. In *Proc. the 2021 Design, Automation & Test in Europe Conference & Exhibition*, Feb. 2021, pp.1130–1135. DOI: 10.23919/DATE51398.2021.9474194.

[28] Naylor M, Moore S. A generic synthesisable test bench. In *Proc. the 2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign*, Sept. 2015, pp.128–137. DOI: 10.1109/MEMCOD.2015.7340479.

[29] Shen H H, Wei W L, Chen Y J, Chen B W, Guo Q. Coverage directed test generation: Godson experience. In *Proc. the 17th Asian Test Symposium*, Nov. 2008, pp.321–326. DOI: 10.1109/ATS.2008.42.

[30] Huang B Y, Zhang H C, Subramanyan P, Vizel Y, Gupta A, Malik S. Instruction-level abstraction (ILA): A uniform specification for system-on-chip (SoC) verification. ACM Trans. *Design Automation of Electronic Systems*, 2019, 24(1): Article No. 10. DOI: 10.1145/3282444.

[31] Canakci S, Delshadtehrani L, Eris F, Taylor M B, Egele M, Joshi A. DirectFuzz: Automated test generation for RTL designs using directed graybox fuzzing. In *Proc. the 58th ACM/IEEE Design Automation Conference*, Dec. 2021, pp.529–534. DOI: 10.1109/DAC18074.2021.9586289.

[32] Jiang S N, Ou Y H, Pan P T, Cheng K S, Zhang Y X, Batten C. PyH2: Using PyMTL3 to create productive and open-source hardware testing methodologies. *IEEE Design & Test*, 2021, 38(2): 53–61. DOI: 10.1109/MDAT.2020.3024144.

[33] Laeufer K, Iyer V, Biancolin D, Bachrach J, Nikolić B, Sen K. Simulator independent coverage for RTL hardware languages. In *Proc. the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2023, pp.606–615. DOI: 10.1145/3582016.3582019.

Yi-Nan Xu received his B.Eng. degree in computer science and technology from the University of Chinese Academy of Sciences, Beijing, in 2019. He is currently a Ph.D. candidate at the State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences, Beijing. His research interests include agile development of processors and processor microarchitecture.

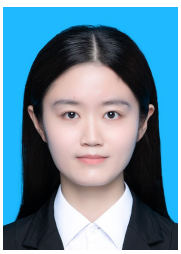
Zi-Hao Yu received his Ph.D. degree in computer systems organization from the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, in 2022. He is currently an engineer at ICT, CAS. His research interests include computer architecture and operating systems.

Kai-Fan Wang received his B.Eng. degree in computer science and technology from the University of Chinese Academy of Sciences, Beijing, in 2020. He is currently a Ph.D. candidate at the State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences, Beijing. His research interests include agile development of processors and processor microarchitecture.

Hua-Qiang Wang received his B.Eng. degree in computer science and technology from the University of Chinese Academy of Sciences, Beijing, in 2020. He is currently a Master student at the State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences, Beijing. His research interests include processor microarchitecture and open-source hardware design.

Jia-Wei Lin received his B.Eng. degree in computer science and technology from the University of Chinese Academy of Sciences, Beijing, in 2020. He is currently a Master student at the State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences, Beijing. His research interests include high-performance computer architecture.

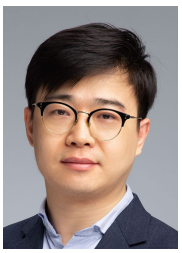
Yue Jin received his B.Eng. degree in computer science and technology from the University of Chinese Academy of Sciences, Beijing, in 2020. He is currently a Ph.D. candidate at the State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences, Beijing. His research interests include high performance processor design, computer architecture security, and processor verification.



Lin-Juan Zhang received her B.Eng. degree in computer science and technology from the University of Chinese Academy of Sciences, Beijing, in 2020. She is currently a Master student at the State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences, Beijing. Her research interests include high-performance computer architecture.

Zi-Fei Zhang received his B.Eng. degree in computer science and technology from the University of Chinese Academy of Sciences, Beijing, in 2020. He is currently a Ph.D. candidate at the State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences, Beijing. His research interests include computer architecture and agile development.

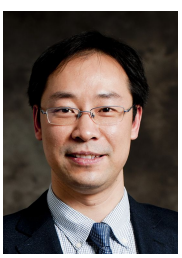
Dan Tang received his Ph.D. degree in computer systems organization from the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, in 2010. He is currently a senior engineer at the State Key Lab of Processors, ICT, CAS, and the assistant director at the Beijing Institute of Open Source Chip, Beijing. His research interests include computer architecture and low power SoC design.



Sa Wang received his Ph.D. degree in computer science from the Institute of Software, Chinese Academy of Sciences, Beijing, in 2016. He is currently an associate professor at the State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences, Beijing. His current research interests include cloud computing and operating systems.

Kan Shi received his Ph.D. degree in digital computing from Imperial College London, in 2015. He is currently an associate professor at the State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences, Beijing. His research interests include agile chip design and verification, and custom computing using FPGAs.

Ning-Hui Sun received his Ph.D. degree in computer systems organization from the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, in 1999. He is currently a professor at the State Key Lab of Processors, ICT, CAS, Beijing, and an academician of the Chinese Academy of Engineering. His research interests include computer architecture and high-performance computing.



Yun-Gang Bao received his Ph.D. degree in computer systems organization from the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, in 2008. He is currently a professor at the State Key Lab of Processors, ICT, CAS, and the deputy director of the ICT, Beijing. His research interests include open source hardware and agile chip design, data center architecture, and memory systems.