



# Design and development of a Hybrid Out-of-Order RISC-V Processor Model

Rajesh Jain, NXP, Noida, India ([rajesh.jain@nxp.com](mailto:rajesh.jain@nxp.com))

Mayuri Gadewar, NXP, Noida, India ([mayuri.gadewar@nxp.com](mailto:mayuri.gadewar@nxp.com))

Ashish Mathur, NXP, Noida, India ([ashish.mathur@nxp.com](mailto:ashish.mathur@nxp.com))

Sourav Roy, NXP, Noida, India ([sourav.roy@nxp.com](mailto:sourav.roy@nxp.com))

**Abstract-** RISC-V has garnered significant attention across both industry and academia, driving extensive research aimed at advancing the architecture and its surrounding ecosystem. In this paper, we present a performance modeling framework designed to facilitate microarchitectural exploration of RISC-V processors. Our solution integrates a functionally accurate, high-speed RISC-V instruction set simulator (ISS) with a highly parameterized performance model of an out of order execution CPU. A key innovation of the framework is its ability to support dynamic switching between fast functional and detailed performance modes at runtime, enabling comprehensive execution-mode performance analysis of complex benchmarks running atop an operating system. The modeling framework features an Instruction Set Architecture (ISA) agnostic interface between the functional and performance models and a simulation kernel that supports co-simulation of the fast functional ISS with a lightweight, parameterized, timing-only CPU performance model, thereby providing an extensible interface that can support custom RISC-V extensions.

**Keywords—**RISC-V, Hybrid Model, Performance Model, RISC-V Linux Benchmarks;

## I. INTRODUCTION AND RELATED WORK

### A. Introduction

The RISC-V instruction set architecture (ISA) has rapidly gained traction as a flexible, open-source alternative to proprietary ISAs, fostering innovation in both academic research and industrial development. Its modular design, extensibility, and open governance model have made it a compelling choice for a wide range of computing domains—from embedded systems to high-performance computing. As the RISC-V ecosystem matures, there is a growing need for robust tools that support architectural exploration, performance evaluation, and design-space analysis.

Accurate performance modeling plays a critical role in the design and optimization of modern processors. However, existing simulation frameworks often present trade-offs between simulation speed, accuracy, and flexibility. Cycle-accurate simulators, while precise, require high effort to develop and are typically slow and cumbersome for large-scale workloads. Conversely, fast functional simulators lack the timing fidelity required for detailed performance studies. Bridging this gap requires a hybrid approach that combines the strengths of both paradigms.

In this work, we present a novel performance modeling framework tailored for RISC-V microarchitectural exploration. Our framework couples a functionally accurate, high-speed RISC-V instruction set simulator (ISS) with a lightweight, highly parameterized CPU performance model. This co-simulation environment enables dynamic switching between fast functional and detailed performance mode at runtime, allowing researchers to conduct execution-mode performance analysis of complex benchmarks, including those running on top of operating systems.

A key feature of this framework is its ISA-agnostic interface and modular simulation kernel, which facilitate extensibility and reuse across different architectural configurations. The timing model is implemented as a queue-



based abstraction, enabling rapid prototyping of various microarchitectural features such as pipeline depth, cache latencies, and instruction issue policies. This design empowers architects to explore a wide range of design choices with minimal overhead.

### B. Related Work

A substantial body of research has been dedicated to the performance evaluation of RISC-V processors using virtual prototyping and simulation frameworks. Among the most widely adopted tools is gem5, an open-source architectural simulator that has been extended to support RISC-V ISA for both syscall emulation and full-system simulation. Recent efforts have focused on enabling full-system simulation capabilities in gem5 for RISC-V, allowing researchers to accurately analyze the system using actual system software stacks [1] [2].

In parallel, lightweight performance modeling approaches have been proposed to couple simple timing models of an in-order execution CPU with functional ISSs. For instance, one study describes the integration of a time-annotated analytical model with a RISC-V ISS to simulate in-order CPU behavior [3]. While effective for early-stage design exploration, such models typically lack the complexity required to represent out-of-order execution or superscalar architectures.

Our work advances the state of the art by enabling co-simulation between a fast, instruction-accurate RISC-V ISS supporting RV64IMAFD, SV39 MMU, and PMP and a detailed, cycle-accurate microarchitectural performance model capable of modeling superscalar, out-of-order CPUs. Unlike prior approaches, our simulation framework features a generic and extensible interface that synchronizes the execution of the functional and performance models in a lock-step fashion. This modular design facilitates easy integration of new RISC-V extensions and supports portability to other ISAs.

Furthermore, our platform introduces a gear-shift mechanism that allows seamless switching between functional and performance simulation modes. It also supports timing back-annotation, enabling the performance model to inform the functional ISS of timing effects, thereby enhancing simulation accuracy and enabling more realistic benchmarking and architectural exploration.

## II. FRAMEWORK DESIGN

### A. Functional ISS Model

In this work, a high-speed functional Instruction Set Simulator (ISS) model for the RISC-V architecture is used. It is rigorously validated against the corresponding RTL (Register Transfer Level) implementation. The model was architected based on the official RISC-V specifications and subsequently verified through extensive comparison with RTL simulations to ensure functional accuracy. The ISS supports both 32-bit (RV32) and 64-bit (RV64) base integer instruction set architectures and is extensible to accommodate a wide range of custom instruction set extensions. This flexibility enables the model to serve as a robust platform for early software and tools development, architectural exploration, and as a solid building block for system-level simulation.

Designed with modularity and integration in mind, the ISS features Transaction-Level Modeling (TLM) interfaces for both instruction and data paths. It also includes multiple interrupt interfaces and standard peripheral connections, making it suitable for integration into larger virtual platforms or co-simulation environments.

The functional model provides a rich set of runtime control APIs, including *run()* and *stepn()*, which allow fine-grained control over simulation execution. The *run()* API executes the model continuously until a predefined condition is met, such as reaching the end of a simulation quantum, encountering a Wait-For-Interrupt (WFI) instruction, or hitting a breakpoint or watchpoint. In addition to execution control, the ISS incorporates industry-standard debug features. These include support for breakpoints, watchpoints, execution tracing, logging, and integration with external debuggers. Such capabilities make the model highly suitable for software bring-up, firmware validation, and debugging tasks in pre-silicon environments.

### B. Queue Timed Model

The CPU performance model is a timing-only micro-architectural queue model representing a high-level, cycle-accurate timing abstraction designed for performance analysis of generic out-of-order microprocessors, hereafter referred to as the queue-timed model. Unlike traditional simulators, it does not execute instructions directly. Instead, it operates on instruction traces or instruction metadata generated by a functional Instruction Set Simulator (ISS). This decoupling from functional execution allows for rapid evaluation of architectural design choices without the overhead of full instruction execution. Rather than explicitly modeling individual pipeline stages, the queue model conceptualizes the processor pipeline as a series of abstract queues. These queues simulate the flow of instructions through the processor, where each queue corresponds to a logical stage or resource. On each clock cycle, instructions are drained from upstream queues and inserted into downstream queues, mimicking the dynamic behavior of a real pipeline. This abstraction enables the model to capture key timing effects while remaining lightweight and highly configurable.

The model accounts for performance-degrading events such as cache misses or branch mispredictions by introducing configurable timing penalties. For example, a last-level cache miss that accesses DDR memory can be modeled with a fixed latency penalty (e.g., 100 cycles). These penalties cause queues to fill up, simulating pipeline stalls and backpressure effects. One of the key strengths of the queue model is its configurability. Users can specify queue depths, processing rates, and behavioral characteristics to reflect different microarchitectural designs. Additionally, parameters such as cache miss rates, cache miss penalties, and branch misprediction penalties can be tuned to explore a wide range of design scenarios. The queue model architecture provides interfaces to allow integration with detailed microarchitecture performance model of the memory subsystem, for wider system level performance exploration.

Once configured, the queue model processes instruction metadata to estimate performance metrics such as Instructions Per Cycle (IPC), queue utilizations, and stall statistics. This makes it a powerful tool for early-stage microarchitectural exploration, enabling rapid design-space evaluation with minimal simulation overhead.

### C. Lock-Step Coupling of Functional ISS and Queue Timed Model

The simulation framework integrates a functional Instruction Set Simulator (ISS) model and a queue-timed microarchitectural model in a tightly coupled, lock-step fashion through a set of custom-designed APIs. At the onset of simulation, the queue-timed model initializes its internal structures, including instruction queues and pipeline stages, based on detailed microarchitectural parameters such as issue widths, execution unit counts, cache latencies, branch penalties, and other parameters (see Figure 1).

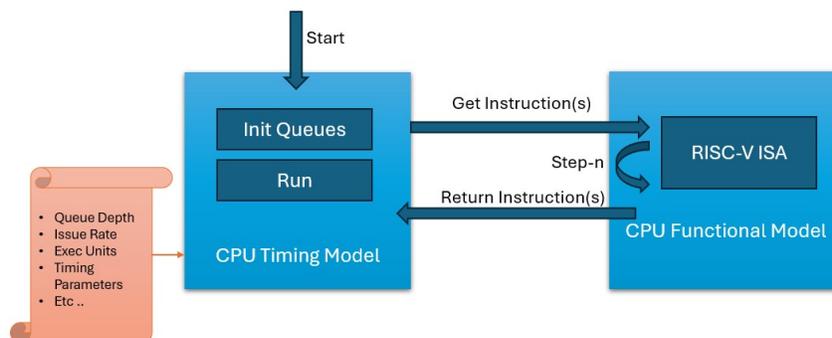


Figure 1 – RISC-V performance mode enabled from start of simulation



From the beginning of the simulation, both models operate in a synchronized manner. The queue-timed model drives the simulation by requesting a configurable number of instructions from the functional ISS model, hereafter referred to as the instruction block. This request is serviced through the *get\_instructions()* API, which internally invokes the *stepn()* function to execute the specified number of instructions. During this execution, the functional ISS populates an ISA-agnostic interface structure with metadata for each instruction. This metadata includes attributes such as program counter (PC) value, source and destination registers, memory addresses (for load/store operations). The hybrid model wrapper layer further adds instruction-specific timing characteristics like execution latency and repeat rate, as published in the CPU specifications. ISA instructions are categorized into distinct functional groups to facilitate accurate modeling of pipeline behavior. These groups include:

- Branch Group
- Jump PC Group
- Multiply Group
- Divide Group
- Load/Store Group
- CSR Modify Group
- Floating Point Group
- Activation Group
- Exceptions Group
- Atomic Memory Operations (AMO) Group
- Other Group

Each group is associated with a set of architectural attributes that influence how the instruction is processed within the queue-timed model. Once the metadata structure for the instruction block is populated, the queue-timed model consumes the instructions cycle-by-cycle, simulating their progression through the pipeline stages according to the architectural constraints. Upon completion of the current instruction block, control is handed back to the functional ISS model to fetch and execute the next set of instructions. This lock-step execution model ensures tight synchronization between functional correctness and performance modeling, enabling accurate simulation of instruction behavior under realistic microarchitectural conditions.

### III. LINUX BOOT ON FUNCTIONAL SIMULATOR

Booting Linux on a functional simulation model of a RISC-V processor requires proper initialization of several critical system components. These components mimic real hardware and software behaviors and provide the necessary support for successful kernel bring-up in a simulated environment.

#### A. Boot ROM Code initialization

The Boot ROM code is the first software executed after reset. In simulation, it acts as minimal firmware responsible for initializing the system and transferring control to the Linux kernel. It also passes essential platform configuration details to the kernel, such as memory size, clock frequency, address of the interrupt controller, peripheral mappings (e.g., interrupt controller). In our setup, the Berkeley Boot Loader (BBL)[4] is used as the Boot ROM. BBL is a lightweight bootloader developed as part of the Berkeley RISC-V software stack.

#### B. Semihosting

Semihosting enables a simulated RISC-V system to perform I/O operations (like console output) via the host machine, which is essential in ISS simulators lacking real peripherals. This is achieved through the Host-Target Interface (HTIF) protocol[4] and the RISCV-FESVR library[4], which together emulate system calls for proxy kernel[4] applications. HTIF transactions are 64-bit values encoding a device ID (e.g., 1 for console), command ID (e.g., 0 for getchar, 1 for putchar), and a 48-bit payload. Communication occurs via memory-mapped registers—*tohost* (target to host) and *fromhost* (host to target), with the simulator polling for transactions at regular intervals. Following diagram shows how RISCV-FESVR library has been integrated into functional ISS model.



### C. Interrupt controller

Core local interrupt controller (CLINT) is a memory mapped module responsible for handling timer and software interrupts. Timer interrupts are triggered when `mtime[8]` exceeds `mtimecmp[8]` setting the `MTIP[8]` bit in the `mip` CSR. Software interrupts are triggered via writes to the `msip[8]` register. These interrupts are handled by the Linux kernel's trap handler, which uses the `mcause[8]` register to identify the interrupt source. Software interrupts are commonly used for context switching and deferred processing in uniprocessor systems.

### D. RISC-V Linux and benchmarks

A single monolithic application binary image was constructed, combining the Berkeley Boot Loader (BBL), the Linux kernel, benchmark binaries, and a launcher script. This unified image ensures a consistent and automated Linux boot environment tailored for benchmarking. Benchmark applications were integrated into the `/usr` directory within the root filesystem along with launcher script. After Linux boots, the launcher script automatically runs benchmark based on a provided benchmark identifier. This identifier is interpreted at runtime, and the corresponding benchmark is executed without requiring manual intervention.

To streamline the boot process and support large-scale, automated benchmarking, Initialization of the `eth0` network interface was intentionally disabled to reduce boot time, and password prompts and login requirements were disabled to enable fully automated runs immediately after boot.

These design choices enabled:

- Uniformity: All benchmarks shared the same kernel, bootloader, and filesystem.
- Automation: No manual file transfers or reconfiguration between benchmark runs.
- Reproducibility: A consistent and isolated execution environment for accurate performance comparisons.

### E. Launcher application for benchmark selection

During Linux boot, a dedicated launcher application is automatically invoked using the `/etc/inittab` configuration, which triggers a shell script. This launcher is responsible for initiating the correct benchmark workload from a precompiled set of applications included in the root filesystem (e.g., `/usr/bin/`). However, a challenge arises in passing the benchmark ID from the simulator to the application. Since the simulator lacks knowledge of applications virtual address mappings, it cannot directly pass arguments to the user-space application. To resolve this, the simulator writes the benchmark ID to a known physical memory location, specifically the last address in the BootROM region, which is reserved for this purpose. At runtime, the launcher application uses the `mmap()` system call to map this physical address into its virtual address space, allowing it to access the benchmark ID like a regular memory variable. This ID is then used to determine and invoke the correct benchmark function.

### F. Benchmark Execution and Validation:

This integrated image and automation framework enable each benchmark to be executed in an easily reproducible environment. Post-boot, the launcher application invokes the appropriate benchmark as determined by the provided ID. The benchmark then runs within the simulated Linux environment, producing console outputs and, in some cases, writing results to files in the root filesystem. To validate functional correctness, output files from benchmarks—such as those produced by SPECint GCC—are compared against reference outputs generated by running the same benchmarks on an x86 host system. Identical outputs confirm that the benchmarks executed correctly and that the functional model behaves consistently across platforms.

A key challenge in running memory-intensive benchmarks is the gap between simulator-configured memory and user-space availability. Though BBL and the simulator may define 2048 MB, the OS often exposes only ~1300 MB due to kernel reservations. Workloads like `429.mcf` can exceed this, leading to segmentation faults. Since the OS relies on the device tree, increasing declared memory is essential for successful execution.

#### IV. SWITCHING FROM FUNCTIONAL MODE TO PERFORMANCE MODE

Accurate performance analysis demands precise control over simulation mode transitions. To ensure that only the relevant execution window is analyzed, a dedicated mechanism is implemented to activate the performance model exactly at the desired point. This prevents interference from unrelated code, such as Linux kernel initialization. The performance model is triggered only after the functional model completes booting into Linux and the target benchmark application begins execution within the simulation environment.

Simulation mode switching can be controlled using three key indicators:

- **Start PC** – For full benchmark execution (without SimPoint[7] slicing), the performance model is triggered at the application’s start PC and remains active until the application terminates as shown in Figure-2. This ensures comprehensive performance profiling of the entire workload.
- **Instruction count** - For SimPoint based simulation, both the instruction count and the application’s start PC are used to control when performance modeling begins. Instruction counting starts only after the program counter matches the specified start PC, ensuring that only user level instructions are considered. When the instruction count reaches the SimPoint start, the simulator transitions to performance mode for the configured interval.
- **System call** - Although a system call could be used to signal the start of the benchmark for performance modeling, this approach was avoided due to the requirement of modifying the application source code and recompiling it with an additional system call. To maintain benchmark integrity and portability, a non-intrusive method of using start PC and instruction count was preferred.

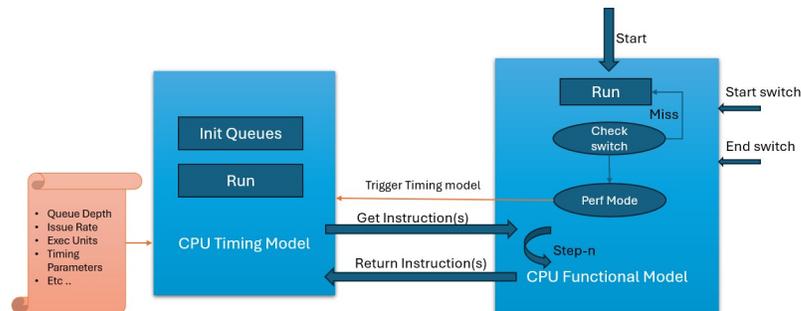


Figure 2 – Switching RISC-V b/w performance mode and functional mode

##### A. Simpoint aware runtime switching framework

Running full benchmark workloads on the performance model is computationally expensive and often impractical. To address this, SimPoints[7] are used to identify and simulate only the most representative portions of the program’s execution. Each SimPoint entry consists of:

Starting point – the dynamic instruction count at which the representative interval begins

Interval length – fixed across all SimPoints

Weight – a scalar value indicating how representative the interval is of the full program’s behavior

These parameters along with application start\_pc are passed to the simulator to enable simpoint execution. Instruction counting begins only after the program counter reaches the specified application start\_pc, which corresponds to the address of the function inside the launcher application that invokes the benchmark. Once instruction counting is active, the simulator tracks dynamic instruction execution. When the count matches a SimPoint’s defined starting point, the simulator transitions from functional to performance simulation. This

continues for the duration of the SimPoint interval, after which it returns to functional mode. This selective simulation ensures that only the most impactful segments are analyzed in detail.

For a given SimPoint with index  $i$ , the modeled instruction window is defined as:

$$\begin{aligned} \text{start} &= i * \text{interval\_length} \\ \text{end} &= (i + 1) * \text{interval\_length} \end{aligned}$$

This defines the exact instruction range that will be simulated using the performance model for each SimPoint, enabling targeted and efficient performance analysis.

### B. Timekeeping and mtime Register Emulation in Simulation

The OS reads `mtime` periodically, using a known tick rate derived from the core clock and timebase frequencies to maintain system time for tasks like scheduling, timers and timekeeping. In functional only simulations, time is emulated via an instruction driven model assuming an IPC of 1. Thus, one instruction equates to one simulated cycle. For a 1 GHz core and a 10 MHz timebase, `mtime` increments every 100 instructions. This ensures monotonic time for OS services like `gettimeofday()`. Upon transitioning to performance simulation, cycle accurate timing is used. The current `mtime` is saved as the base value. After simulation, the additional cycles from performance model are scaled (e.g. divide by 100) and added to the base to update `mtime`, maintaining continuity.

Using CoreMark and `gettimeofday()`, we validated timekeeping accuracy. For 148 million cycles at 1 GHz with a 10 MHz timebase, the expected `mtime` increment is 1.48 million ticks, corresponding to 0.148 seconds. This confirms accurate time emulation across functional and performance phases.

## VI. RESULTS

The novel solution proposed in this paper has been used to develop a hybrid RISC-V performance model. The framework has been successfully tested for functional and performance accuracy using bare metal benchmarks like Coremark and Linux based benchmarks like SPEC integer suite. To run these benchmarks, run-time switching between functional model and performance mode has been extensively used. Figure 3 illustrates the simulation workflow of various 403.gcc benchmark variants from the SPECint2006 suite. During the initial phase of the Linux system bring-up, the simulation operates in functional-only mode, focusing on verifying system correctness and enabling basic software initialization. Once the system reaches a stable operational state, the simulation transitions into performance mode, allowing detailed execution of the benchmark to collect performance metrics and analyze architectural behavior. The Figure captures the instruction counts between functional model and performance mode.

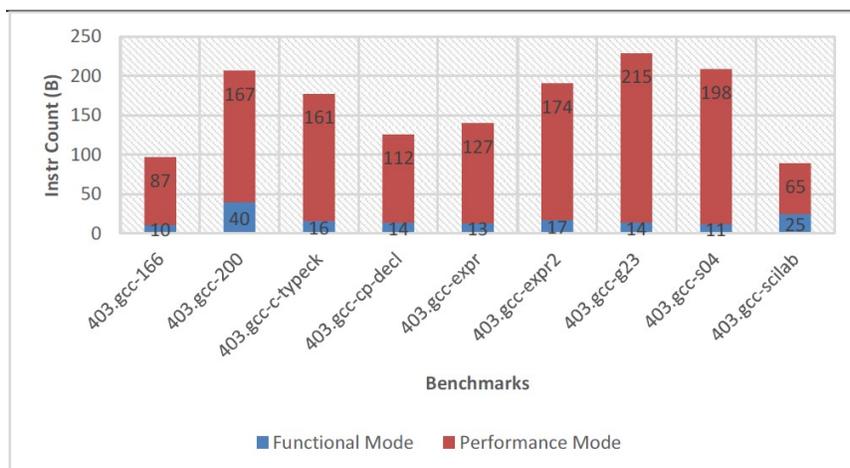


Figure 3 SPECint2006 403.gcc instruction count(B) comparison between performance mode and functional mode



## VII. CONCLUSION AND NEXT STEPS

This paper presents a methodology for implementing hybrid simulation on a RISC-V platform, enabling accurate transition from functional simulation to detailed performance modeling at the desired execution point. A predefined function address, known as the `start_pc`, is used to trigger the switch to the performance model, ensuring simulation focuses exclusively on the target application workload, avoiding boot-time kernel activity.

The supporting infrastructure includes:

- A functional simulator with SV39 MMU support and full Linux bring-up capability.
- An efficient run-control mechanism that manages runtime parameters such as `start_pc`, benchmark ID, and SimPoint metadata (start instruction, interval, weight) to enable fine-grained performance sampling.
- A fully automated Linux environment ensuring consistent and reproducible benchmark execution.
- Functional correctness validation through comparison of benchmark outputs with known-good results.

A key strength of this setup is its integration with a generic, highly parameterized microarchitectural performance model, which allows detailed exploration of various design configurations (e.g., pipeline depth, cache hierarchy, memory latency). This makes the framework adaptable to a wide range of architectural studies. Together, these components provide a powerful platform for efficient and accurate performance analysis. By combining realistic software execution with flexible hardware modeling and precise simulation control, this hybrid simulation framework significantly improves the scalability and reliability of architectural evaluation.

As next steps, we plan to look into designing a SystemC wrapper for the RISC-V hybrid model, enabling plug-n-play into larger SystemC based system level performance models. Another area is to investigate techniques for speeding the simulation time without sacrificing on the cycle accuracy.

## REFERENCES

- [1] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, *et al.*, “The gem5 simulator,” *ACM SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011.
- [2] 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. Workshop on Computer Architecture Research with RISC-V (CARRV)*, 2021.
- [3] V. Herdt, D. Große, and R. Drechsler, “Fast and Accurate Performance Evaluation for RISC-V using Virtual Prototypes,” in *Enhanced Virtual Prototyping: Featuring RISC-V Case Studies*, Springer, 2020.
- [4] RISC-V ISA Simulator (Spike) GitHub Repository: <https://github.com/riscv-software-src/riscv-isa-sim>
- [5] <https://techroose.com/tech/linuxOnSpike.html>
- [6] <https://spec.org/>
- [7] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically characterizing large scale program behavior. In *Proceedings of the ASPLOS X*
- [8] The RISC-V instruction set manual, volume II: Privilege architecture