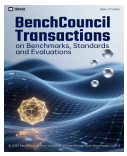


BenchCouncil Transactions on Benchmarks, Standards and Evaluations, 2026

Full Length Articles

FULL LENGTH ARTICLES

TraceRTL: Agile Performance Evaluation for Microarchitecture Exploration

Zifei Zhang[1,2] Yinan Xu[1] Kaichen Gong[4] Sa Wang[1,2] Dan Tang[1,3] and Yungang Bao[1,2,*]

[1]State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences, 100190, Beijing, China, [2]University of Chinese Academy of Sciences, 100190, Beijing, China, [3]Beijing Institute of Open Source Chip, 100080, Beijing, China and [4]School of Information Science and Technology, ShanghaiTech University, 200000, Shanghai, China

[*]Corresponding author. baoyg@ict.ac.cn

Received on 2 February 2026; Accepted on 29 March 2026

Abstract

While agile chip development methodologies have accelerated RTL design and simulation, performance evaluation remains constrained by challenges: (1) limited benchmark availability due to incomplete peripheral/software simulation environments or unavailable source code; (2) inefficient feature prototyping caused by the tight coupling between functional correctness and performance evaluation, particularly for large-scale, error-prone microarchitectures. To address these challenges, we propose TraceRTL, an agile, trace-driven performance evaluation methodology that decouples the functional and performance components of CPU RTL designs. It introduces three contributions to the benchmarking community: (1) a trace-driven exploration framework that bypasses full functional correctness while preserving performance behavior and supports replaying workload traces on RTL designs; (2) a quantitative analysis and mitigation methodology to identify and reduce trace-driven performance discrepancies; (3) a trace transformation technique, TraceBridge, that converts benchmark traces between different formats and instruction sets. Using TraceRTL, we have developed the first trace-driven RTL CPU derived from XiangShan, a high-performance out-of-order RISC-V processor. TraceRTL achieves performance accuracy of 99.87% and 99.86% on SPECint2017 and SPECfp2017, respectively. With TraceBridge, we evaluate x86 Google workload traces on a RISC-V RTL CPU and reveal distinct memory-bound behavior.

Key words: Trace-driven simulation, Performance evaluation, Cross ISA benchmarking

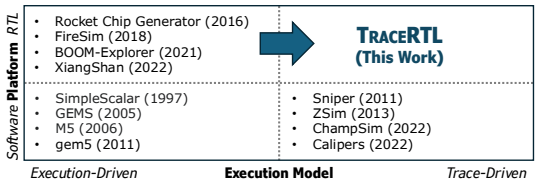
1. Introduction

Performance has always been a central consideration in CPU development. As Moore's Law slows and application demands diversify, achieving further performance improvements has become increasingly challenging. This highlights the importance of microarchitecture exploration methodologies. A key question is: given a baseline CPU design, how can we efficiently quantify the performance impact of a proposed hardware feature using representative benchmarks?

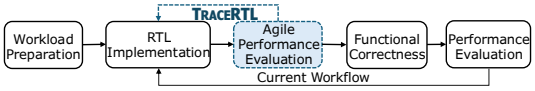
Among available evaluation methods for assessing CPU design changes using diverse benchmarks, the most faithful approach is to use the register-transfer level (RTL) implementation. As the definitive description of the microarchitecture, RTL is the most reliable basis for assessing CPU microarchitecture designs. Ultimately, any proposed feature must be implemented and evaluated in RTL to determine its true performance impact.

However, since the RTL development process is time-consuming, the computer architecture community has adopted more efficient approaches to accelerate early-stage exploration before implementing a proposed feature in RTL. As shown in Fig. 1(a), software-based architectural simulators [1–8] model low-level hardware components using high-level languages and abstractions, enabling fast simulation and rapid design iteration. Despite their high productivity in *early-stage* exploration, the *last mile* remains unavoidable: performance must still be re-evaluated at the RTL level after initial simulator studies, since the additional modeling layer inevitably introduces discrepancies that require substantial engineering efforts and costly calibration with the actual implementation [9].

Another fundamental yet often overlooked challenge is the benchmarking asymmetry across the development workflow. While software simulators [5, 6, 8, 10] widely adopt trace-driven methodologies to execute diverse benchmarks in trace format, RTL models lack the capability to replay traces and faces limited benchmarks due to immature simulation environments. The *benchmarking gap* prevents a consistent and continuous evaluation flow from early-stage modeling to final hardware implementation.



(a) Approaches to microarchitecture exploration.



(b) Current and TraceRTL performance evaluation workflows.

Figure 1. Microarchitecture exploration methods and workflows.

Recent advancements in RTL design and simulation offer strong potential for a seamless, progressive refinement workflow from early-stage exploration to the last mile. On the design side, high-level hardware construction languages [11–13] enable parameterized and reusable components, allowing rapid implementation and iteration of new microarchitectural ideas. On the evaluation side, efficient RTL simulation methods [14–18], especially FPGAs and emulators [19–22], have significantly accelerated large-scale RTL simulation. These capabilities have already been demonstrated in several open-source, industrial-competitive CPUs [23–27], which provide realistic and accessible microarchitecture research platforms [28–32]. For example, it takes less than 200 minutes and approximately 300 lines of modified Chisel code to implement an instruction scheduler policy PUBS [33] on the XiangShan, a high-performance RISC-V CPU achieving >15 SPECint2006/GHz [26, 34].

These trends motivate us to adapt proven exploration techniques from simulators directly to RTL, aiming to *inherit the agile workflow of simulator-based exploration while enabling a seamless integration into RTL for last-mile evaluation.*

To realize this opportunity, we *propose* **TraceRTL**, an RTL-based performance evaluation methodology that derives a trace-driven RTL model from an existing execution-driven RTL implementation. As illustrated in Fig. 1(a), TraceRTL reuses open-source, silicon-validated RTL designs as a solid foundation for faithful microarchitecture exploration. It drives performance-critical modules with pre-generated traces, enabling simulator-like agility for early-stage exploration on RTL. By deriving the trace-driven model directly from RTL, it inherently avoids costly last-mile calibration and preserves performance fidelity for evaluation of the proposed feature.

One key motivation behind TraceRTL is to overcome the execution-driven nature of current RTL designs. As illustrated by the white boxes in Fig. 1(b), the modified CPU must first pass full functional verification *before* any performance evaluation can be conducted. This tight coupling forces every RTL design modification to undergo complete implementation, verification, and lengthy simulation, even when the modification is unrelated to performance. For example, evaluating optimizations for virtualized, two-stage address translation requires implementing complex privileged operations to guarantee correct functionality, whose functional details, however, do not affect performance.

With TraceRTL, this strict dependency between functional correctness and performance evaluation is eliminated. As highlighted in Fig. 1(b), TraceRTL enables ***agile performance evaluation*** without first implementing or verifying unrelated functional details. Additionally, it accepts trace inputs from ***a broader range of real-world applications***, including those with unavailable source code, different ISAs, or peripheral dependencies [10, 35–37], without requiring porting to an RTL simulation. To realize these capabilities, however, we need to address three key challenges.

1) *Feature prototyping: Can we develop a trace-driven RTL CPU with minimal modifications to the existing execution-driven microarchitecture while preserving performance accuracy?* Our key insight is that hardware module interfaces can be categorized into functional interfaces, which determine what each instruction computes and where execution proceeds next, and performance-sensitive interfaces, which determine how efficiently the instruction stream is realized. Based on this distinction, TraceRTL selectively takes over key interfaces to decouple the functional model while preserving performance behaviors using externally supplied traces. This preserves cycle-accurate performance accuracy while eliminating the complexity of managing full functional correctness.

2) *Performance accuracy: Can we mitigate the performance discrepancies introduced by trace-driven simulation?* Conventional trace-driven simulation often suffers from fidelity loss due to the lack of necessary information to replicate execution-driven behaviors. We observe that these information gaps stem from two primary sources: intentional abstraction and dynamic omission. We quantitatively analyze the performance impact of these missing components, revealing their essential role for fidelity. TraceRTL proposes a dynamic information reconstruction mechanism that synthetically reconstructs missing data, achieving high performance accuracy.

3) *Broader workloads: Can we bridge the semantic gap across diverse trace formats and ISAs?* Industrial workloads are valuable for microarchitecture exploration, but the scarcity of RISC-V workloads necessitates cross-ISA transformation to generate benchmark traces. This transformation is performed only once during trace preparation. However, differences in trace formats and ISAs hinder the direct execution of publicly available traces on RTL CPU models. Since trace-driven simulation relaxes the need for full functional correctness, TraceRTL introduces TraceBridge, a trace transformation technique that leverages instruction and register mapping to enable the replay of traces from different formats and ISAs.

To demonstrate the feasibility of TraceRTL, we develop a trace-driven RTL model derived from XiangShan [26, 38]. It achieves performance accuracy of 99.87% and 99.86% on SPECint2017 and SPECfp2017, respectively, reducing performance discrepancies by 10.31× and 29.21× compared to a calibrated XS-gem5 model. By leveraging TraceBridge, we evaluate x86-based Google workload traces [36] on XiangShan, and reveal distinct memory-bound behavior compared to SPECint2017.

TraceRTL expands the possibilities for microarchitecture research by supporting both RTL-based exploration and seamless integration with simulator-based workflows. By preserving a simulator-like, trace-driven environment for workloads and simulation, it effectively bridges early-stage exploration on simulators and last-mile RTL evaluation.

To summarize, this paper makes the following contributions.

- We propose TraceRTL, bringing trace-driven simulation to RTL CPUs for agile microarchitecture exploration.

- We quantify the sources of performance discrepancies and implement dynamic information reconstruction to achieve high performance accuracy.
- We propose TraceBridge, which enhances trace compatibility to expand the sources of benchmark workloads.
- We demonstrate TraceRTL by using x86 workload traces collected from Google warehouse-scale computers for performance evaluation of XiangShan, a RISC-V CPU.

2. Background

2.1. Out-of-Order Microarchitecture

Modern CPUs improve performance primarily by exploiting parallelism and speculation. The front-end speculatively fetches instructions using branch prediction, while the back-end decodes, schedules, and issues them to execution units for computation and to memory subsystem for data access.

The efficiency of this pipeline depends on several critical microarchitectural components. Branch prediction and instruction fetching determine the instruction supply rate. Execution pipelines and scheduling queues affect throughput. The memory hierarchy bridges the large speed gap between CPU and DRAM by caching frequently used data. The memory management unit (MMU) accelerates address translation by caching recently used address mappings near the CPU.

2.2. Exploration on RTL

While RTL models offer higher accuracy for design space exploration, directly evaluating performance on RTL presents several challenges, including the inflexibility of traditional hardware description languages, slow simulation speeds, and the lack of open-source RTL processors. Recent efforts have focused on these issues.

Flexibility. Many emerging high-level hardware description languages [12, 13, 39] offer enhanced expressiveness and parameterization that accelerate the development of microarchitectures . New hardware design methodologies [11] are also proposed to further improve design modularity.

Simulation Speed. Novel RTL simulation techniques have been proposed to accelerate software-based [14–16] or hardware-based [19, 21, 22] simulation of RTL designs.

Additionally, sampling-based methods [40–42] estimate full-program performance by aggregating results from several representative program segments.

Open-Source RTL Processors. With the rapid growth of the RISC-V open-source community, a number of RTL processors have emerged, including in-order designs [43, 44] and out-of-order designs such as BOOM [23–25], XuanTie-910 [27], and XiangShan [26]. These designs provide accessible and realistic platforms for microarchitecture research, enabling agile exploration directly on RTL.

2.3. Simulation Methodologies

In computer architecture research, performance evaluation of novel designs predominantly relies on two core methodologies: execution-driven and trace-driven simulations. These approaches fundamentally differ in how they provide program stimuli to performance models, leading to distinct trade-offs between fidelity, flexibility , and simulation speed.

The execution-driven methodology emulates the behavior of real CPUs within the performance model, such as fetching, decoding, scheduling and executing instructions. This approach is inherent to RTL models [23–26, 43] and is also implemented in many software simulators [1–4]. By coupling functional execution with performance modeling, this approach captures microarchitecture-dependent dynamic behaviors, such as speculative execution and wrong-path effects, thereby offering high fidelity. However, this accuracy comes at the cost of significant complexity, increased error-proneness, and reduced simulation speed.

In contrast, the trace-driven methodology decouples the functional model from the performance model by replaying the pre-generated traces of instructions including architectural information such as instruction semantics, instruction addresses, memory accesses, and branch outcomes [5, 6, 8, 10, 45]. These traces are often generated using instrumentation tools like Pin [46], DynamoRIO [47], and Valgrind [48], or obtained from public pre-generated traces [10, 35, 36]. This decoupling affords higher flexibility, enabling researchers to focus on microarchitectural optimization. However, this flexibility often comes at the cost of reduced fidelity, as traces lack dynamic microarchitecture-dependent information.

3. Challenge

Agile performance evaluation requires rapid feature prototyping, support for extensive workloads, and fast simulation. To meet these goals at the RTL level, trace-driven simulation offers a promising approach by decoupling performance and functional models and supporting trace-based workloads. However, integrating trace-driven simulation into existing execution-driven CPU RTL models introduces non-trivial challenges. Publicly available traces often vary in trace format, lack information such as instruction encodings, and are sometimes generated from different instruction sets.

3.1. Trace-driven RTL Integration

Transforming a complex execution-driven CPU RTL model into a trace-driven implementation presents unique challenges compared to building an RTL model from scratch or driving individual RTL modules independently. In addition to supplying stimuli to existing RTL modules, a trace-driven model must precisely control the instruction flow based on external traces while maintaining the original performance behavior.

3.2. Trace-driven Performance Discrepancies

Trace-driven simulation inherently suffers from fidelity loss due to the lack of necessary information. This gap stems from two primary sources: intentional abstraction and dynamic information omission. First, to balance confidentiality and storage overhead, conventional traces often omit critical details such as operand values and instruction opcodes. Second, static traces fail to capture dynamic execution states, such as wrong-path instructions and page table walks, which only emerge during runtime. The absence of these microarchitectural side effects prevents the accurate replication of execution-driven behaviors, potentially leading to significant performance discrepancies.

3.3. Trace Compatibility

Trace-driven approaches can bypass the limitations of simulated peripheral environments, thereby enhancing the coverage of supported workloads. However, due to confidentiality constraints, instruction source code is often unavailable for publicly accessible trace files [10, 35, 36]. Another scenario

involves target applications that require evaluation but have not been adapted to the target instruction set, rendering direct assessment infeasible.

Instruction sets share commonalities but also exhibit significant differences, which hinder direct trace porting. For example, differences in general-purpose register conventions, instruction encodings and sizes, PC alignment rules, and the range of direct branch instructions all impose constraints on cross-instruction-set trace evaluation. These challenges are particularly pronounced for RTL models, which typically lack sufficient abstraction capabilities.

4. TraceRTL Design

To enable agile performance evaluation of RTL designs, we first propose a trace-driven simulation methodology at the RTL level (§ 4.1) while preserving high performance accuracy (§ 4.2). Building on this, we introduce TraceBridge, a trace transformation method that enhances compatibility by enabling the replay of traces from different formats and instruction sets (§ 4.3).

4.1. Trace-Driven Microarchitecture Design

We decompose the CPU into core components and describe how each component is driven by the trace. Interfaces, defined asthe set of I/O signals between modules, can be driven to control the module's behavior. By driving the key interfaces with the information in traces, TraceRTL replaces the functional model with external traces while maintaining the original performance behavior. This section describes the design of trace-driven integration to meet its objectives: (1) driving RTL modules with external instruction traces, (2) enforcing the CPU model to conform to the trace instruction flow, and (3) identifying and mitigating performance discrepancies inherent in trace-driven simulation.

4.1.1. Trace-Driven RTL Modules

Our key insight is that hardware module interfaces can be classified into functional interfaces, which determine what each instruction computes and where execution proceeds next (e.g., arithmetic, branching, or exception handling), and performance-sensitive interfaces, which determine how efficiently the instruction stream is realized (e.g., branch prediction, cache access, and memory prefetching). Based on this distinction, we analyze key module behaviors and drive performance-sensitive modules using external instruction traces, preserving original performance characteristics without requiring full functional execution.

Branch predictor. The branch predictor's performance-critical interfaces primarily include two types: training and prediction. The predictor is trained on the committed branch outcomes. Therefore, instructions on the mis-speculated path, which are flushed from the pipeline, leave no side effects. By substituting the branch outcomes with trace information, which includes branch direction and target, we are able to stimulate the training process. For prediction, the predictor takes the current program counter (PC) and branch history to generate the next instruction fetch request. While prediction is at speculative stage, the PC and history for correct-path instructions are consistent between execution-driven and trace-driven simulations.

Instruction fetch. The instruction fetch unit obtains fetch requests from the branch predictor and retrieves instructions from the traces. We propose an *interval match mechanism* to simulate the fetch bandwidth, as shown in Fig. 2. A fetch request typically specifies a contiguous instruction interval I_{bp} defined by starting and ending addresses. The fetch unit forwards the request to TraceReader that extracts a continuous sequence of trace instructions I_{trace}. Instructions in I_{if}, which are common to both I_{bp} and I_{trace}, are then sent to subsequent pipeline stages for execution. When the starting address does not match the beginning of the trace, I_{if} is empty, preventing any instructions from being fetched. Consequently, the impact of instructions on the mis-predicted path cannot be modeled. § 4.2.1 presents a refined design to address this limitation.

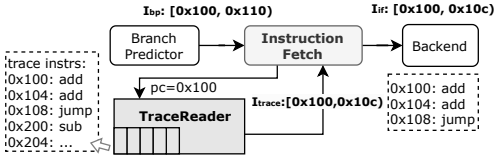


Figure 2. Trace-driven instruction fetch with interval match mechanism.

Out-of-order backend. The backend relies on instruction encodings to stimulate decoding, register renaming, dynamic scheduling and execution. These encodings are directly supplied from the trace. Alternatively, a more aggressive approach is to provide the results of the decoding directly to drive renaming and scheduling, although this is beyond the scope of this work. Particular units like the FDivSqrt operation may need optional data for accurate execution latency.

Cache hierarchy. Cache behavior is mainly influenced by access addresses. Instruction addresses are derived from fetch requests generated by the branch predictor. Data addresses, on the other hand, are dynamically calculated from the operands, which are invalid in trace-driven simulation. Therefore, memory access addresses should be included in traces to model memory behavior. Special modules like the indirect memory access prefetcher need extra information.

Memory management unit. The virtual-to-physical address translation and page-table walk require in-memory page table entries (PTEs) that are typically absent in traces [49]. We employ a dynamic page table generation approach: For each instruction in the trace, we traverse the page tables using its virtual address. If a required PTE is invalid, a new page is allocated, and the corresponding PTE is initialized. This process continues recursively until reaching the leaf page, which is initialized with the physical address in traces.

4.1.2. Trace-Controlled Instruction Flow

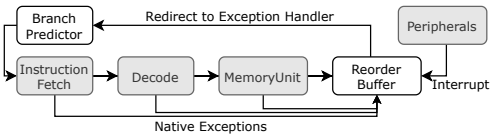
TraceRTL controls the instruction flow by managing branch instructions, interrupts, and exceptions, while ensuring processor compliance by instruction stream correctness checks.

Branch instruction. We replace the branch execution unit's outcomes with target and conditional result recorded in the trace to control the programs' instruction flow.

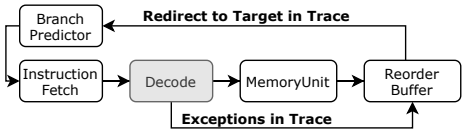
Exception and interrupt. Traps, including exceptions like page faults and interrupts like timer interrupts, may be triggered by programs, devices, and operating system. Traps affect control flow and pipeline redirection, as illustrated in Fig. 3(a). These are intercepted and re-injected according to the trace. Specifically, trace-recorded exceptions are triggered as illegal instructions, redirecting to the target in trace, as illustrated in Fig. 3(b). This design ensures that exceptions are preserved without relying on full functional execution.

Table 1. Key CPU module behaviors and their corresponding trace-driven stimuli in TraceRTL.

Module	Key Behavior	Trace-Driven Stimulus
Branch Predictor	Prediction uses PC and history; Training uses committed branch outcomes.	Use current PC and history for prediction; Use branch outcomes from trace for training.
Instruction Fetch	Fetch request defines instruction interval; Wrong-path instructions.	Apply interval match mechanism to simulate fetch bandwidth; Generate wrong paths on mismatch.
Instruction Execution	Decode, rename, schedule, and execution.	Use instruction encoding and optional operand from trace.
Instruction Flow	Branch instruction outcome; Exception and interrupt redirect the pipeline.	Intercept branch outcomes/exception generation; Support redirect; Flow check.
Cache Hierarchy	Access cache by addresses.	Memory address from trace; Instruction address from branch predictor; Optional data from trace.
MMU	Virtual-to-physical address translation; Access memory for page table entries.	Construct page table according to the address from trace.



(a) Native exceptions and interrupt triggered by the CPU pipeline and peripherals.



(b) Intercept the native exceptions and trigger trace exceptions as illegal instruction.

Figure 3. Exception and interrupt management in TraceRTL.

Instruction stream check. A fundamental requirement of trace-driven simulation is that the performance model must be guided by the trace, a key aspect of which is to ensure its execution adheres to the provided instruction stream. We capture the processor's actual instruction stream through committed instructions and compare it against the trace. The differences in the streams indicate implementation flaws in the RTL model itself or trace-driven framework.

4.1.3. Overall

In summary, TraceRTL provides a general and adaptable framework for trace-driven RTL performance evaluation. It is designed to evolve naturally with RTL designs, require minimal effort across microarchitectural iterations, remain applicable across diverse microarchitectures, and flexibly support various performance optimizations.

Extending TraceRTL to new architectures. We summarize the trace-driven transformation methodology in Table 1. TraceRTL employs an interface-based modification strategy that reduces modification overhead while accommodating variations in module design. The processor module partitioning methodology is universal across different microarchitectures, making TraceRTL a reusable and microarchitecture-agnostic framework for RTL performance evaluation. The specific modifications may vary depending on processor-specific designs. For instance, in-order processors commonly fetch one or two instructions per cycle, which does not require the interval match described in § 4.1.1. In contrast, some high-performance processors may fetch instructions spanning two intervals per cycle, thus necessitating two interval-match operations. For CPU-driven accelerators, such as matrix units, the necessary execution information can also be recorded into trace instructions and dispatched accordingly.

Applicability for microarchitecture features. TraceRTL is particularly advantageous for evaluating functionally complex yet performance-critical features (§ 7.2). Beyond functionality, it captures fine-grained timing effects that are difficult to model accurately at higher abstraction levels. For example, variations in microarchitectural timing may critically affect the overall performance (§7.4). It can also evaluate microarchitectural optimizations in the same way as conventional trace-driven simulators (e.g., branch prediction, prefetching, replacement, memory dependence prediction). With additional trace information, TraceRTL can be extended to model advanced optimizations, such as value prediction (with execution results) and indirect memory prefetching (with memory values).

4.2. Trace-driven Performance Discrepancy Mitigation

To achieve high accuracy, trace-driven simulation should strive to mimic the behaviors of execution-driven simulation. This section details our methodology for bridging this gap by enhancing trace-driven simulation of the frontend fetch unit through wrong-path simulation, refining execution latency via operand and opcode provisioning, and maintaining MMU fidelity through dynamic page table construction.

4.2.1. Fetch: Wrong-Path Simulation

Out-of-order processors may execute instructions that are later discarded due to events like branch mispredictions. These instructions, although executed, are flushed by pipeline redirect operations, preventing them from affecting the architectural state of the CPU, such as the register file or memory.

Wrong-path instructions' performance impact, particularly on the cache hierarchy, cannot be ignored. The impact on the cache can be categorized into **prefetching** and **pollution**, leading to positive and negative effects. Fig. 4 presents a code example divided into three sections: (1) Code1, executed unconditionally before the branch; (2) Code2, located within one branch; and (3) Code3/Code4, placed outside the branch's influence, further categorized into the proximate Code3 and

the distant Code4. Upon a mispredicted branch, Code2 is executed, and if its execution is swift, Code3 may follow. Once the branch is resolved, speculatively fetched instructions of Code2 and Code3 are discarded, with Code2 potentially polluting the cache and Code3 prefetching the cache.

```
*b = 1;  /* Code1 */
a = *b;
if (a == 0)
  a = *c;/* Code2 */
a = *d;  /* Code3 */
a = *e;  /* Code4 */
```

Figure 4. Code example demonstrating wrong-path instruction generation.

We statistically analyze the number and addresses of memory instructions on both correct and wrong paths in the out-of-order processor XiangShan, focusing on instructions sent to the load pipeline. These addresses are aligned to cache-line size. We categorize the address space into three types: (1) exclusive-arch-path: only accessed by correct path instructions, (2) exclusive-wrong-path: only accessed by wrong-path instructions and (3) overlapped: accessed by both paths. As shown in Fig. 5, we found that most of the address space falls into type(1) and (3). Therefore, we can tentatively draw a rough conclusion that *prefetching has the predominant influence.*

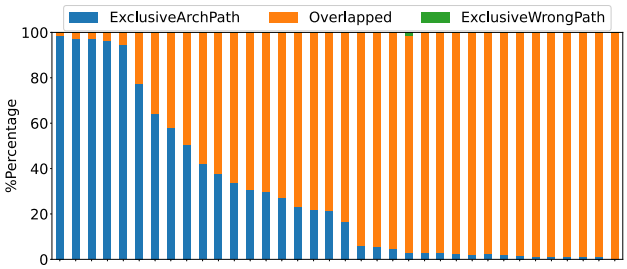


Figure 5. Percentage of memory interval weighted by load access times for the SPEC CPU2017. Each bar represents a sub-benchmark, sorted according to "ExclusiveArchPath".

Based on the observation, we focus on simulating the prefetching influence by *taking the instructions at correct path as wrong-path instructions*. The process involves the following steps: (1) When a branch misprediction occurs, we check whether the fetch request's starting address exists in traces within a fixed instruction window; (2) If it exists, the instructions in the trace are sent to subsequent pipeline stages as wrong-path instructions. The instruction fetch unit is blocked for simplification. These instructions are not discarded from the traces; (3) Once the branch instruction is resolved and the pipeline is redirected, the correct fetch request is issued.

4.2.2. Execution: Instruction Opcode Provisioning

Conventional trace-driven simulators often operate with partial instruction encodings. Instruction encoding has two types of information: instruction opcode for functionality like ADD and SUB, register indices for instruction dependency and out-of-order scheduling. Explicit instruction opcodes are frequently

Algorithm 1 Dynamic Page Table Construction

```
1: procedure InstructionWalk(instList)
2:     for inst in instList do
3:         if inst's PC valid then
4:             PageWalk(inst.VirtualPC, inst.PhysicalPC)
5:         end if
6:         if inst's memory address valid then
7:             PageWalk(inst.VirtualAddr, inst.PhysicalAddr)
8:         end if
9:     end for
10: end procedure
11: procedure PageWalk(va, pa)
12:     pageBase = PageTableRootAddr
13:     for level := 0 to MaxLevel do
14:         pteAddr = getPteAddr(va, level, pageBase)
15:         pte = readPageTable(pteAddr)
16:         if pte not valid then
17:             if level == MaxLevel-1 then
18:                 newPte = genPte(pa)      ▷ Leaf page arrived
19:             else
20:                 newPte = genPte(AllocatePage())
21:             end if
22:             writePageTable(pteAddr, newPte)
23:         end if
24:         pageBase = pte.ppn << 12
25:     end for
26: end procedure
```

abstracted or omitted for confidentiality concerns and software simulators' highly abstracted microarchitecture designs. Consequently, instead of providing detailed opcodes, trace instructions are categorized into coarse-grained functional groups: (1) control flow (unconditional direct, conditional direct, and indirect jumps); (2) memory access (loads and stores); and (3) computation (integer and floating-point).

Our work focuses on quantifying the performance modeling deviations induced by this loss of fine-grained opcodes. Specifically, we investigate how substituting precise opcodes with coarse-grained categories impacts simulation fidelity. This analysis aims to isolate the impact of operation abstraction from other simulation variables, providing a quantitative understanding of the accuracy trade-offs in abstracted trace modeling.

4.2.3. Execution: Operand Provisioning

Some operations are implemented in a blocking manner and their execution cycles are variable depending on the operands, like division, floating-point division and square-root. This type of performance error is always neglected and simulators often implement them with fixed latency.

Although these instructions are relatively few, their long execution cycles and low degree of concurrency amplify their performance impact. To achieve more accurate simulation for these types of instructions, we record their operands in traces.

4.2.4. MMU: Dynamic Page Table Construction

User-space programs use virtual addresses, which must be translated to physical addresses by the memory management unit (MMU) before accessing the cache or main memory. In the MMU, the virtual address first consults the L1 translation lookaside buffer (TLB). If L1 TLB hits, the physical address

is obtained directly. In case of L1 TLB miss, the virtual address will be sent to a larger L2 TLB or hardware page table walker to traverse the memory-resident page tables to find the physical address corresponding to the virtual address, which involves multiple memory accesses, especially in hypervisor environments. Page table caches are used to speed up page table walks. In summary, the hit rates of TLB and page table cache, as well as page table walker's memory latency, are crucial for MMU-sensitive programs.

To simulate the behavior of MMU and minimize the modifications on RTL modules, we need to provide a self-consistent page table for the MMU. However, traces typically contain only the physical and virtual addresses, but not the page table [49]. Therefore, we employ a dynamic page table generation method, as illustrated in Algorithm 1. By iterating over each instruction in the traces and traversing the page tables based on the virtual address, we allocate new page frames and initialize the invalid corresponding page table entries, until reaching the leaf page. The leaf entry is then initialized with the corresponding physical address. After dynamically generating the page table, when a TLB miss occurs, the memory-resident page tables are traversed.

4.3. Trace Compatibility with TraceBridge

We introduce a trace transformation methodology, TraceBridge, to bridge the incompatibilities in trace formats and instruction sets. To support trace-driven simulation, the trace must contain at least three categories of information: (1) primary instruction type, including branch types, computation, and memory operations; (2) execution guidance, including PC, branch target and conditional result, and memory address; (3) register dependencies to model instruction-level parallelism. Such information is typically included in the trace format of dynamic instrumentation tools [49] and publicly available traces [10, 35, 36], where fine-grained semantic information such as instruction opcodes are sometimes missing.

TraceBridge retains the key information from the trace, transforming its format to be compatible with the target model by refining the execution semantics. However, trace-driven RTL models pose additional low-level challenges due to their rich details: (1) *instruction correspondence and register semantics*; (2) *difference in instruction encoding size and program counter (PC) alignment constraints*; (3) *variations in branch offset ranges.*

The primary principle of TraceBridge is to maintain performance semantics consistency. This ensures that the performance characteristics of the original program are reflected in the target architecture. For confidentiality, public traces omit instruction encodings [10] or provide instruction categories [36]. To address this, we observe that an instruction can encompass multiple performance semantics, which fall into four types: ⟨Load, Computation, Store, Branch⟩. To maintain performance semantics consistency, we map each individual performance semantic to its corresponding instruction(s) in the target ISA. A single x86 instruction, which may encompass multiple micro-operations, is translated into an equivalent sequence of RISC-V instructions. For instance, the x86 RET instruction is mapped to two RISC-V instructions (LOAD and JR), and x86 memory accesses exceeding the width of a single RISC-V instruction are decomposed into multiple instructions to preserve the access range. The necessary mapping results in instruction inflation, which is analyzed in § 7.1. In the case of missing opcodes, compute instructions are mapped to representative types such as [F]ADD, [F]MUL, and CONVERT due to limited information in the traces. For ISAs with flag mechanism, such as x86, spare registers can be employed to establish inter-instruction dependencies. Furthermore, special handling for architecturally significant registers, like the return address register, guarantees the correct correspondence between x86 call/return operations and their RISC-V counterparts.

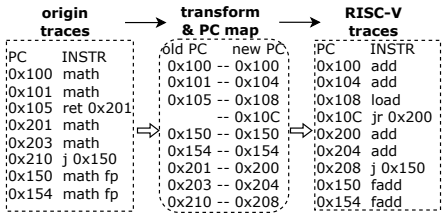


Figure 6. Example of trace transformation, consisting of PC conversion and instruction encoding mapping

To resolve differences in instruction size, PC alignment, and instruction inflation, we reorganize PCs in the traces to conform to RISC-V requirements. As illustrated in Fig. 6, we collect all instruction PCs and sequentially reassign new addresses based on RISC-V encoding size. When a PC gap is detected (e.g., from 0x105 to 0x150), the current PC is updated accordingly. A mapping from original PCs to RISC-V PCs is then constructed, and branch target addresses are updated using this mapping.

While the x86 ISA supports larger offset ranges for direct branch instructions than RISC-V, we observe that branch target computation mainly occurs in two modules: the pre-decoding unit at the fetch stage and the branch execution unit. By overriding the computation result with the target recorded in the traces, we effectively support larger branch offset ranges in the trace-driven RISC-V model.

Overall. TraceBridge provides a methodology to evaluate the microarchitectural behavior of mature, real-world software ecosystems (e.g., Google workloads) on an emerging hardware ecosystem (e.g., RISC-V). Admittedly, TraceBridge is unable to eliminate all performance discrepancies caused by inherent cross-ISA differences and missing execution information in traces, such as instruction semantics and application binary interfaces(ABIs). Furthermore, while the high-level methodology is consistent, the specific rules should adapt for source and target ISAs. For example, x86 and RISC-V differ in the number of general-purpose registers. Consequently, when translating to x86, some registers may map to memory (i.e., register spilling). It is also constrained by information missing from the trace, forcing a simplified instruction remapping, which inevitably introduces performance errors. However, according to our evaluation of missing RISC-V opcodes (§ 7.1), the accuracy is above 99% (0.95% error for SPECint2017) for early-stage performance exploration.

5. Put It All Together

TraceRTL improves the performance evaluation workflow by optimizing stages such as workload preparation, prototyping, and performance simulation. As illustrated in Fig. 7, a typical iterative workflow based on TraceRTL is employed to perform agile performance evaluation. The workflow involves the following steps: ① **Trace Preparation**: Program traces for the benchmarks or target applications are prepared

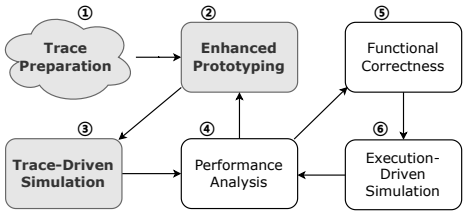


Figure 7. Agile performance evaluation workflow with TraceRTL. The workflow comprises two loops: a trace-driven loop ②→③→④→② and an execution-driven loop ④→⑤→⑥→④.

for subsequent performance evaluation. Each trace represents a program segment. Traces can be generated using a variety of tools, including dynamic instrumentation tools like Pin [46] and DynamoRIO [47], instruction-level simulators like QEMU [50] , and publicly available traces such as Google workload traces [36] and Qualcomm workload traces [35]. TraceRTL can be combined with additional techniques such as SimPoint [40] to further shorten simulation time, while also avoiding the overhead and complexity of booting. ② **Prototyping**: New microarchitectural features can be prototyped on a RTL model without full implementation, as shown in § 7.2. ③ **Trace-Driven Simulation**: The trace-formatted program segments are replayed in trace-driven simulation, yielding performance results of the CPU model. ④ **Performance Analysis**: The performance results and program behaviors are analyzed to identify performance bottlenecks. These insights inform subsequent iterations and guide prototype refinement. ⑤ **Functional Correctness**: When the design meets expected performance targets, the efforts invested in prototype development can be seamlessly carried over. TraceRTL supports compile-time mode switching between execution-driven and trace-driven simulation, enabling smooth transition to functional validation. ⑥ **Execution-Driven Simulation**: Further performance analysis and iteration are conducted through execution-driven.

TraceRTL facilitates an agile and accurate RTL-level microarchitecture design exploration process. Rather than replacing existing architectural simulators, TraceRTL serves as a complementary and reinforcing component that enhances RTL performance exploration and bridges the gap between high-level models and real RTL behavior. It targets a distinct sweet spot in the accuracy-productivity trade-off, preserving the ground-truth RTL model and accepting manageable maintenance overhead to achieve substantially higher accuracy, with comparable or potentially lower (Palladium/FPGA) simulation cost. By enabling direct performance evaluation on real RTL implementations, TraceRTL empowers architects to broaden application coverage and identify microarchitectural bottlenecks that high-level simulators may overlook.

6. Evaluation

We conduct evaluations to address two key questions:

1. Can we mitigate trace-driven simulation's performance inaccuracies (§ 6.2)?
2. Does TraceRTL achieve high performance accuracy (§6.3)?

To address these questions, we compare the performance of the original RTL model, TraceRTL, and the state-of-the-art simulator gem5 [4].

Table 2. Target system configuration.

Component	Description
Branch Predictor	uBTB, BTB, TAGE-SC, ITTAGE, RAS
Fetch/Decode/Rename Width	8/6/6
RoB/LoadQueue/StoreQueue	160/72/64
Integer/Float Register File	224/192
ALU/FMA/FDivSqrt unit	4/4/2/
Load/Store unit	3/2
L1 ICache	64KB, 4-way, 256-set
L1 DCache	64KB, 8-way, 128-set
L2 Cache	1MB, 8-way, 512-set, 4-bank
L3 Cache	16MB, 16-way, 4096-set, 4-bank
L1 ITLB/DTLB	48-entry, fully-associative
L2 TLB	2048-entry, 8-way, 32-set
DRAM	DRAMsim3, 8GB, DDR4-3200

6.1. Experimental Setup

Target System. We evaluate TraceRTL by altering an open-source high-performance RISC-V processor, XiangShan [26, 38], into a trace-driven model. TraceRTL introduces low implementation overhead while preserving RTL fidelity. It reuses the original RTL and drives existing modules by intercepting inputs and outputs. The modifications consist of three primary components. First, the simulation environment, implemented primarily in C++, manages trace file loading, instruction stream validation, and page table generation. Second, the TraceRTL module, written in Chisel, retrieves traces via the DPI and supplies instructions to the processor. Third, interface connections and execution guidance are applied to existing processor modules. The first two components are microarchitecture-agnostic, whereas the third requires tighter coupling with specific microarchitectural details. Specifically, the microarchitecture-specific modifications account for fewer than 450 LOC (lines of code). Nevertheless, the modification methodology remains portable across diverse processor designs.

XiangShan, implemented in Chisel [13], is a tape-out ready superscalar out-of-order processor. Its latest third-generation, Kunminghu, achieves a clock frequency of 3GHz and SPECint2006 score exceeding 15/GHz, demonstrating its capability as a platform for exploring high performance microarchitecture designs. We use the default configuration of XiangShan, as shown in Table 2. We take the original XiangShan's performance as the ground truth.

gem5 is widely used for CPU microarchitecture exploration and is often referenced as the ground truth in some simulator works [8, 51, 52] for its rich details. We use the XS-gem5 [53] as the baseline, which has been carefully calibrated to XiangShan through over 1,200 git commits and more than 60,000 lines of source code additions since July 2022, including XiangShan-specific adjustments.

Simulation Speed. XS-gem5 achieves a simulation speed of around 35kHz. As TraceRTL is directly derived from the original RTL model, it inherently shares a comparablesimulation speed and benefits from hardware-accelerated emulation tools. The simulation speeds are both around 6.5kHz using Verilator [14] and around 1.4MHz on Cadence Palladium, which is 40× faster than XS-gem5.

Workloads. We use SPEC CPU2006 [54] and SPEC CPU2017 [55] benchmark suites. We compare the benchmark

scores between XiangShan, TraceRTL and XS-gem5. The complete execution of SPEC CPU benchmarks takes a very long time in software simulation. A set of representative program segments are generated by sampling the SPEC CPU benchmarks using SimPoint [40]. Each segment consists of 20M instructions for warm-up and 20M instructions for performance sampling. To limit simulation time, more than 30% weight of the program segments are included for each application. NEMU [56], an instruction-level simulator, is employed to execute these segments and generate trace files to feed into TraceRTL. Both XiangShan and XS-gem5 are functionally verified against NEMU, guaranteeing they share the same execution flow.

6.2. Trace-Driven Performance Discrepancies

For the first time, we can evaluate the performance impact of the trace-driven simulation on an accurate high-performance RTL processor and the effectiveness of measures to mitigate its performance errors. We quantify the performance errors arising from wrong-path simulation, memory management unit behaviors, operand and opcode absence.

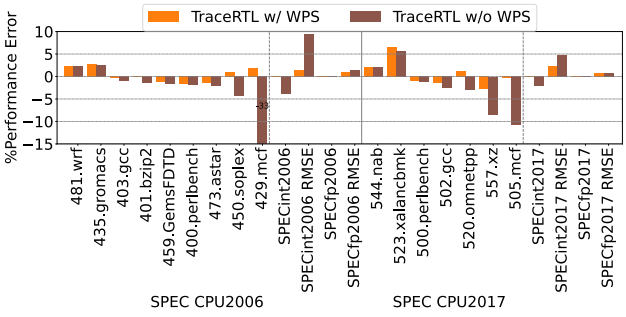


Figure 8. Performance errors of TraceRTL w/ and w/o wrong-path simulation on SPEC CPU2006 and SPEC CPU2017.

6.2.1. Wrong-path Simulation.

We adopt the mechanism detailed in § 4.2.1 to model wrong-path effects. For comparison, we also consider the basic approach where the instruction fetch halts upon encountering a mis-prediction, detailed in § 4.1.1. Fig. 8 illustrates SPEC CPU2006's and SPEC CPU2017's performance differences with and without simulating wrong-path instructions' effect, containing the sub-benchmarks whose "w/o WPS" errors are more than 1%, benchmarks' overall performance errors and RMSE (root mean squared error) metric. Although the overall performance impact of neglecting wrong paths is relatively small (-3.91% and -0.18% for SPECint2006 and SPECfp2006, -2.17% and 0.14% for SPECint2017 and SPECfp2017), certain benchmarks, such as 429.mcf and 450.soplex on SPEC CPU2006 and 505.mcf and 557.xz on SPEC CPU2017, exhibited substantial performance degradation. Our results demonstrate that simulating the impact of wrong-path instructions effectively mitigates these programs' performance discrepancies, reducing the overall performance error to 0.14% for SPECint2006 and 0.13% for SPECint2017. The RMSE of SPECint2006 and SPECint2017 falls from 9.56% and 4.87% to 1.38% and 2.38%.

6.2.2. Instruction Opcode Provisioning

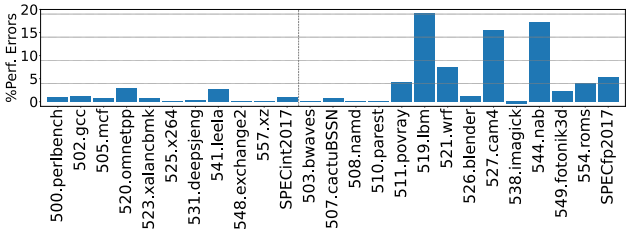
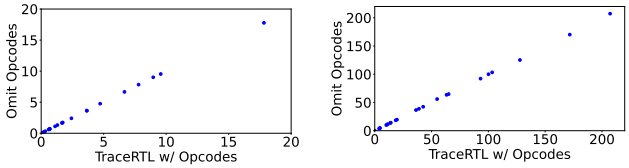


Figure 9. Performance errors of TraceRTL on SPEC CPU2017 when omitting computation instruction opcodes.



(a) Branch predictor MPKI. (b) Data cache MPKI.

Figure 10. BPU and data cache MPKI comparison between TraceRTL w/ and w/o computation instruction opcodes on SPEC CPU2017 benchmarks. Each point represents one sub-benchmark.

Coarse-grained opcode abstraction is common in trace-driven simulators without detailed execution unit modeling, or in applications that directly provide traces without instruction encoding. To quantify the performance deviations, we implemented a controlled mapping scheme within the TraceRTL framework. Specifically, the diverse array of complex computational opcodes are collapsed into a simplified set of generic operations: integer addition/multiplication (ADD/MUL) and floating-point addition/multiplication (FADD/FMUL).

The results across the SPEC CPU2017 demonstrate that the impact of opcode abstraction varies significantly between workload types. As shown in Fig. 9, SPECint2017 exhibits high resilience to coarse-grained semantic mapping, maintaining a negligible average error of 0.95%. In contrast, SPECfp2017 shows a much higher sensitivity, with the average error rising to 5.30% and peaking at 19.29% in 519.lbm. The results suggest that while coarse-grained opcode traces are sufficient for evaluating general-purpose integer architectures, they may introduce unacceptable fidelity loss for floating-point heavy workloads.

Despite the divergence, the coarse-grained abstraction effectively preserves the control-flow and memory-access characteristics of the workloads. As illustrated in Fig. 10, the MPKI metrics of branch predictor and data cache remain highly consistent between the abstracted traces and normal TraceRTL. In summary, coarse-grained opcode abstraction has limited impact on integer compute-intensive applications, frontend modules (branch prediction and instruction fetch), and memory-access related research. It is well-suited for studies where the target workloads or modules have a weak correlation with floating-point operations.

6.2.3. Uncertain-latency Operations

To model the execution latency of uncertain-latency operations, represented by floating-point division and square root (FDivSqrt), we adopt the approach that supplies operands, detailed in § 4.2.3. For comparison, we also evaluated a baseline configuration where the FDivSqrt is replaced with a fixed-latency dummy unit, with latencies varying based on the operation type and data width. As shown in Fig. 11, which

contains sub-benchmarks whose "fixed-latency" errors exceed 0.5%, the fixed-latency model resulted in overall performance errors of -1.65% and -1.22% on SPECfp2006 and SPECfp2017, respectively, with significant deviations for sub-benchmarks such as gromacs and zeusmp in SPECfp2006, and 521.wrf, 527.cam4, and 544.nab in SPECfp2017. By providing operands, we are able to improve the accuracy of performance for these applications.

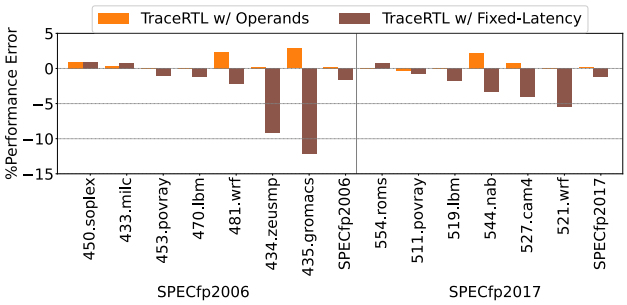


Figure 11. Performance error of simulating FDivSqrt with operand-dependent vs. fixed latency on SPECfp2006 and SPECfp2017.

6.2.4. Memory Management Unit

To evaluate the performance impact of the MMU, we employ the dynamic page table (Dynamic PT) approach detailed in § 4.2.4. For comparison, we also simulate an ideal L1 TLB which always hits and a page table walker with fixed-latency of 15 cycles. As shown in Fig. 12, which contains sub-benchmarks whose "Ideal L1TLB" errors are more than 3%, the ideal MMU introduces average performance discrepancies of 6.19% and 2.35% on SPEC CPU2017 int and fp, with 11 out of 23 benchmarks experiencing performance discrepancies exceeding 3%. When simulating a page table walker with fixed memory latency, 4 out of the 23 benchmarks have errors greater than 3%. In contrast, when simulating the actual MMU behavior, the overall performance overhead decreases to 0.13% and 0.14% for SPEC 2017 int and fp, and only 1 out of 23 benchmarks exhibits a performance error greater than 3%.

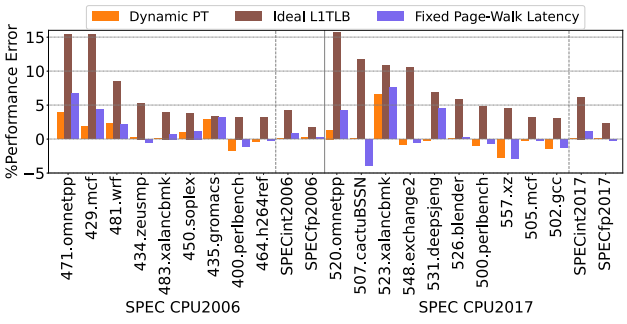


Figure 12. Performance error of simulating the MMU using different strategies on SPEC CPU2006 and SPEC CPU2017.

6.3. Overall Performance Accuracy

We evaluate the performance accuracy of TraceRTL and XS-gem5 on SPEC CPU2006 and SPEC CPU2017, with original XiangShan as the ground truth, as shown in Fig. 13.

Overall. TraceRTL achieves significantly high accuracy in overall performance. For RMSE metric, TraceRTL achieves 1.45% and 1.00% on SPECint2006 and SPECfp2006, compared to 9.85% and 19.44% for XS-gem5. Similarly, the RMSE of SPECint2017 and SPECfp2017 of TraceRTL are 2.38% and 0.67%, compared to 8.02% and 22.53% of XS-gem5.

Sub-benchmarks. TraceRTL exhibits high accuracy at both the overall and sub-benchmark levels. For XS-gem5, on SPEC CPU2006, 11 out of 29 sub-benchmarks have errors greater than 10%, and 14 out of 29 have errors greater than 3%. Similarly, on SPEC CPU2017, 7 out of 23 sub-benchmarks have errors greater than 10%, and 13 out of 23 have errors greater than 3%. These discrepancies can be attributed to the diversity of program characteristics, which makes it challenging to perfectly calibrate. In contrast, by inheriting rich details, TraceRTL effortlessly achieves high accuracy. TraceRTL achieves performance accuracy such that only 1 out of 29 on SPEC CPU2006 and 1 out of 23 on SPEC CPU2017 has an error greater than 3%.

7. Case Studies

In this section, we present case studies to demonstrate how TraceRTL facilitates agile performance evaluation:

1. **Trace Compatibility**: Using TraceBridge, we evaluate x86-based Google workload traces on the RISC-V XiangShan CPU (§ 7.1).
2. **Prototyping**: We use TraceRTL to quickly evaluate the performance impact of adopting a two-stage address translation MMU (§ 7.2) and a new floating-point unit(§ 7.3).
3. **Performance Sensitivity Accuracy**: We compare the accuracy of performance impact between TraceRTL and XS-gem5 at frontend, backend and memory (§ 7.4).

7.1. Trace Compatibility: Google Workload Traces

We evaluate datacenter workloads, the x86-based Google workload traces [36] from warehouse-scale computer workloads on the RISC-V high performance processor XiangShan to show the feasibility of TraceBridge described in § 4.3. Google workload traces consist of multiple trace groups, each containing many trace files. For each group, we select the longest trace and apply the SimPoint [40] for sampling. Applying SimPoint directly to the transformed traces can avoid errors caused by instruction inflation.

While TraceBridge maintains semantic consistency, it introduces the overhead of instruction inflation. We analyze this inflation across both static and dynamic dimensions, considering instruction count and size, as shown in Fig. 14. The inflation ratios for static and dynamic instruction counts remain stable within a narrow range, from 1.09 for arizona to 1.19 for yankee. The dynamic instruction size, an indicator of instruction cache pressure, exhibits an inflation ratio ranging from 0.95 for arizona to 1.20 for bravo.a, with 9 out of 12 applications staying within a 10% inflation margin.

We provide a Top-down [57] breakdown analysis of performance bottlenecks for both Google workload traces and SPECint2017, sorted by IPC, as shown in Fig. 15. While only 3 out of 10 SPECint2017 sub-benchmarks exhibit memory-bound

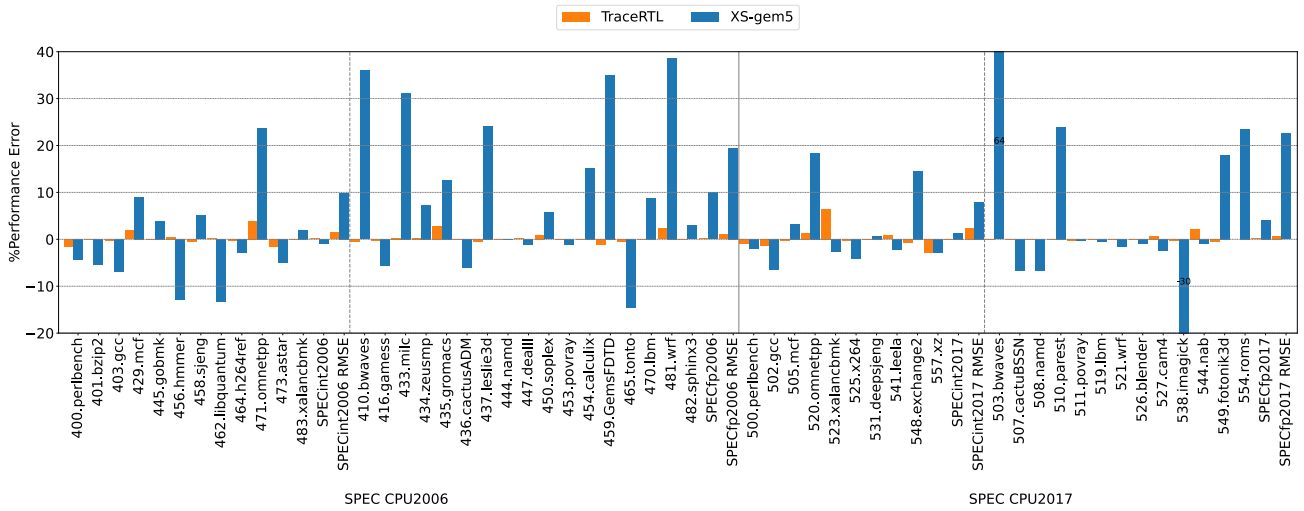


Figure 13. Performance error of TraceRTL and XS-gem5 on SPEC CPU2006 and SPEC CPU2017, using the execution-driven XiangShan as the baseline.

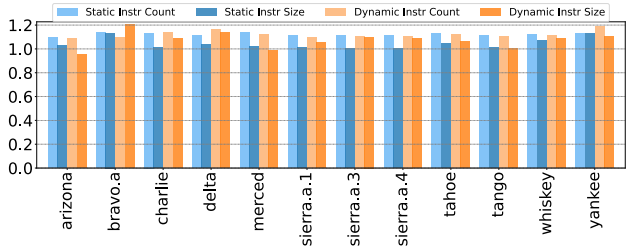


Figure 14. Instruction inflation rate of TraceBridge on Google workload traces.

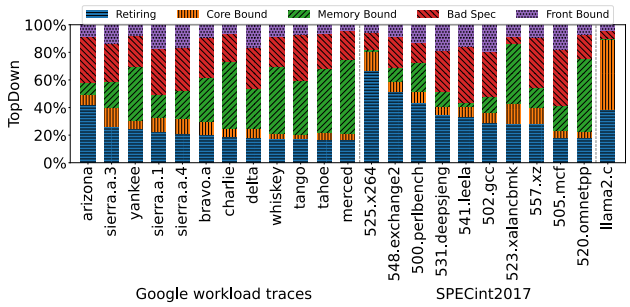


Figure 15. Top-down breakdown comparison between Google workload traces, SPECint2017 , and llama2.c.

over 20%, 8 out of 12 Google workload traces demonstrate this characteristic, with 6 reaching approximately 40%. These results highlight memory access as the primary performance bottleneck, underscoring the importance of memory optimization for warehouse-scale computing systems. TraceBridge introduces dynamic instruction size and count expansion, which primarily affects front-end and core-bound performance categories. However, since these two factors account for relatively small proportions in Google workload traces, TraceBridge has limited impact through expansion effects. Although coarse-grained instruction encoding may potentially affect floating-point workloads, the analysis in § 6.2.2 shows that it preserves accurate instruction streams and cache behavior. This indicates that the impact on memory-bound and bad-speculation categories is also minimal.

TraceRTL also streamlines porting workloads by leveraging the well-developed QEMU. It takes less than 30 minutes to compile llama2.c [58] and generate program traces by QEMU. As shown in Fig. 15, these traces are simulated on TraceRTL, and, unlike Google workload traces and SPECint2017, exhibit distinct core-bound behaviors.

7.2. Prototyping #1: Memory Management Unit

TraceRTL enables efficient prototyping and performance evaluation of complex microarchitectural modules. As a case study, we examine two-stage address translation, a key mechanism for supporting virtual machines through memory virtualization defined in the RISC-V Hypervisor extension [59].

Evaluating this module is non-trivial due to its reliance on privileged operations, complex control and status registers (CSRs), and software-managed page tables. Additionally, its performance impact is significant: address translation may trigger multiple memory accesses to page table. For instance, the RISC-V Sv39 scheme requires 3 memory accesses, while the virtualized, two-stage Sv39-Sv39x4 scheme requires up to 15 memory accesses that increase the translation latency.

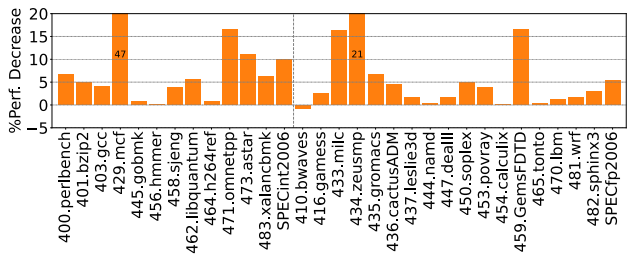


Figure 16. Performance decrease estimation when adopting two-stage address translation on SPEC CPU2006.

TraceRTL allows performance evaluation of such designs before full functional implementation is complete. By (1) directly

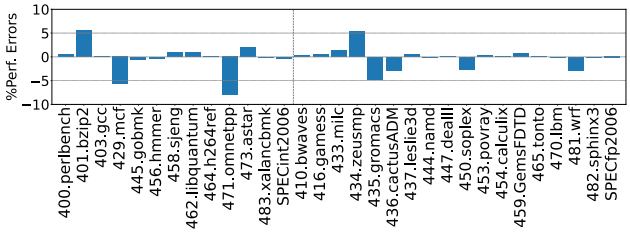


Figure 17. Performance error of TraceRTL on SPEC CPU2006 under KVM virtualization.

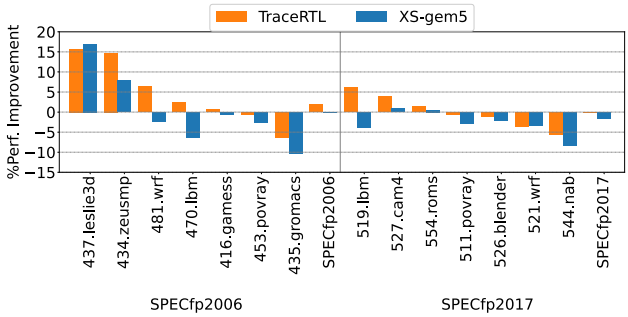


Figure 18. Performance improvement when adopting pipelined FDivSqrt on SPECfp2006 and SPECfp2017.

providing the page table following the two-stage translation scheme and (2) adding a standalone host page table walker which performs guest-physical-address to host-physical-address translation, we enable the MMU to perform the two-stage Sv39-Sv39x4 scheme, thereby obtaining the performance results of two-stage address translation. Fig. 16 illustrates the performance changes of the TraceRTL under normal address translation and two-stage address translation modes on SPEC CPU2006. The two-stage address translation results in a performance degradation of 9.99% for SPECint2006 and 5.27% for SPECfp2017. Among the 29 sub-benchmarks, 10 have a degradation exceeding 5%. In summary, TraceRTL simplifies the requirements for functional correctness and software modifications, providing a robust development platform for exploration around MMU.

To evaluate the accuracy, we compare TraceRTL-based Hypervisor against fully-functional XiangShan Hypervisor on SPEC CPU2006 under KVM virtualization. As shown in Fig. 17, TraceRTL achieves high accuracy, with performance errors below 1% for 19 of 23 sub-benchmarks. The overall error is 0.32% for SPECint2006 and 0.23% for SPECfp2006.

7.3. Prototyping #2: FDivSqrt Unit

TraceRTL enables the implementation of dummy execution units with configurable latency behavior without complex behavioral modeling. For instance, implementing a functional FDivSqrt unit in RTL entails substantial effort, as the implementation in XiangShan exceeds 2,400 LOC and requires extensive verification. To evaluate the pipelined design [60] without incurring such overhead, alternative modeling approaches are necessary. XS-gem5 adopts cycle-accurate modeling for execution units and requires more than 40 LOC of modifications. In contrast, TraceRTL enables dummy implementation with configurable latency in fewer than 10 LOC of modifications, eliminating verification overhead. Figure 18 shows the performance impact of replacing two blocking FDivSqrt units with a pipelined version across benchmarks where TraceRTL changes exceed 0.5%. Notable discrepancies between XiangShan and XS-gem5 appear in SPEC CPU2006 wrf and SPEC CPU2017 lbm. Given the differences in performance accuracy, TraceRTL results are considered more reliable.

7.4. Performance Sensitivity Accuracy

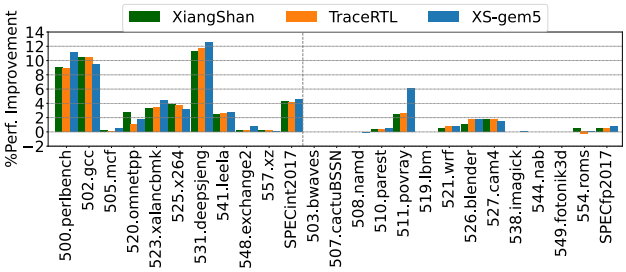
In addition to the performance accuracy of the processor model, the performance sensitivity to microarchitectural modifications is also important. To evaluate the performance sensitivity accuracy to microarchitectural modifications, we adjust key configurations in the frontend, backend, and memory subsystem. For the frontend, we compare the performance impact of different branch target buffer (BTB) sizes—specifically, from 1024 to 2048 (default) entries. For the backend, we vary the number of floating-point units FMA from 2 to 4 (default). For the memory subsystem, we evaluate performance with the best-offset prefetcher in the L2 cache both disabled and enabled (default).

As shown in Fig. 19, we compare the performance variations of XiangShan, TraceRTL and XS-gem5 on SPEC CPU2017 benchmarks under microarchitectural modifications mentioned above. The performance trends observed on TraceRTL closely match those of XiangShan better than those of XS-gem5. For instance, when enlarging BTB size, sub-benchmarks such as 500.perlbench, 502.gcc, and 511.povray exhibit similar trends between TraceRTL and XiangShan. When increasing the number of the FMA, sub-benchmarks like 507.cactuBSSN, 508.namd, and 519.lbm show consistent behavior. When adopting the best-offset prefetcher, sub-benchmarks including 500.perlbench and 507.cactuBSSN also demonstrate analogous performance improvements.

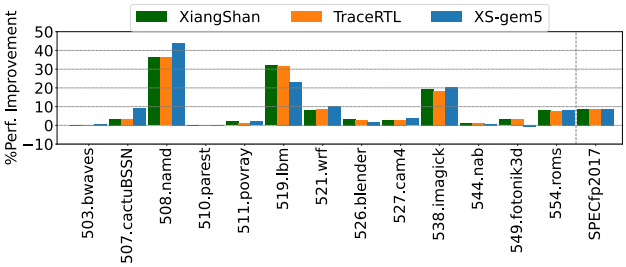
We analyze the notable performance errors of XS-gem5 and find that its prefetching subsystem is considerably more complex and finely tuned, yet lacks clear calibration against the RTL design. This mismatch diminishes the observable performance gains from new prefetchers such as best-offset. The observation highlights the fundamental calibration challenge and motivates the design of TraceRTL: while a model may overfit to the baseline configuration to reproduce similar overall performance, its performance trends for specific microarchitectural features may diverge significantly.

8. Related Work

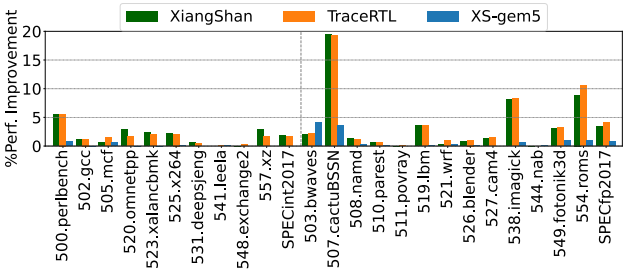
Trace-Driven Model Transformation. Prior work has explored employing trace-based methods to directly control RTL modules' behavior for functional verification, coverage analysis, and performance validation [61, 62]. These works use traces to drive separate RTL modules and the main challenge lies in the generation of traces. Some works collect the traces generated by CPU RTL models for coverage analysis [63]. In contrast, TraceRTL, centered on the whole CPU RTL model, addresses the challenges of design space exploration at the RTL level. Given that achieving high performance accuracy is both a fundamental requirement and a persistent challenge, TraceRTL provides a solution that not only supports prototyping but also enables the execution of workloads in trace form. Trace-driven methodology can be used to improve existing software simulators, such as the trace-driven gem5 mentioned at [64]. In contrast, TraceRTL enhances the RTL simulation to avoid extra model



(a) From 1024 to 2048 BTB entries on SPEC CPU2017.



(b) Increasing FMA number from 2 to 4 on SPECfp2017.



(c) Adopting L2cache best-offset prefetcher on SPEC CPU2017.

Figure 19. Performance improvements of microarchitectural modifications on XiangShan, TraceRTL and XS-gem5.

layers. Accel-Sim [65] adds a new frontend for GPGPU-Sim [66] to support trace-driven simulation. Unlike Accel-Sim's high-level GPU modeling, TraceRTL targets low-level RTL CPU models and addresses challenges of model calibration.

Trace-Driven Performance Inaccuracy. Previous works have investigated performance inaccuracy in trace-driven simulation, primarily focusing on the wrong-path simulation in single-core [67–69], multi-core [70, 71] and synchronization in multi-core simulation [72, 73]. Our methodology mainly focuses on prefetching influence of wrong paths by taking the instructions at the correct path as wrong-path instructions, to suit the RTL model and achieve high accuracy. Moreover, existing trace-driven simulators have a high level of abstraction , which may introduce performance errors thus masking some influencing factors. TraceRTL provides a platform for studying trace-driven simulation.

Error-Prone RTL Model. New RTL languages such as Bluespec SystemVerilog [12], Chisel [13], and SpinalHDL [39] provide high expressiveness and abstraction to reduce design errors. Assassyn [74] introduces a high-level abstraction for asynchronous event handling of pipelined architectures and can generate a calibrated C++ simulator. TraceRTL presents an orthogonal approach to utilizing a trace-driven methodology to decouple the functional and performance models of existing CPU models and expand the scope of workloads.

Trace Format Transformation. Prior work has explored trace format transformation, e.g., converting Arm traces into ChampSim-compatible format [7, 75]. However, ChampSim's high-level abstraction bypasses many low-level challenges, such as differences in instruction semantics, encoding size, PC alignment, and branch offset range, which become critical when executing traces on RTL models.

9. Conclusion

We propose TraceRTL, a methodology to bring trace-driven simulation to the CPU RTL model to facilitate agile performance evaluation. We evaluate TraceRTL by integrating it into XiangShan, achieving high accuracy of 99.87% and 99.86% on SPECint2017 and SPECfp2017. We propose a trace transformation strategy, TraceBridge, and evaluate x86 Google workload traces on the RISC-V XiangShan. TraceRTL mitigates the benchmarking gap between software simulators and RTL design, supports both an RTL-based performance exploration workflow and seamless integration with simulator-driven flows, serving as a bridge from early-stage exploration to last-mile RTL evaluation.

Ethical Statement

No ethical approval was required for this study, as it did not involve human or animal subjects.

Funding

This work was supported by the National Natural Science Foundation of China (Grant No. 62090022, 62090023, 62172388) and the Strategic Priority Research Program of Chinese Academy of Sciences (Grant No. XDA0320000, XDA0320300).

Declaration of competing interests

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data Availability Statements

The data supporting the findings of this study are openly available in XiangShan at `https://github.com/OpenXiangShan/XiangShan/tree/dev-tracertl`.

Credit authorship contribution statement

Zifei Zhang: Conceptualization; Project administration; Methodology; Validation; Investigation; Data curation; Formal Analysis; Writing – original draft. Yinan Xu: Methodology; Writing – review & editing; Kaichen Gong: Software; Validation; Investigation. Sa Wang: Writing; Visualization. Dan Tang: Supervision; Funding acquisition; Resources. Yungang Bao: Supervision; Funding acquisition; Resources; Writing – review & editing.

References

1. Doug Burger and Todd M. Austin. The simplescalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25(3):13–25, June 1997. doi:10.1145/268806.268810.
2. Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, November 2005. doi:10.1145/1105734.1105747.
3. N.L. Binkert, R.G. Dreslinski, L.R. Hsu, K.T. Lim, A.G. Saidi, and S.K. Reinhardt. The m5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, 2006. doi:10.1109/MM.2006.82.
4. Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011. doi:10.1145/2024716.2024718.
5. Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/2063384.2063454.
6. Daniel Sanchez and Christos Kozyrakis. Zsim: fast and accurate microarchitectural simulation of thousand-core systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, page 475–486, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2485922.2485963.
7. Nathan Gober, Gino Chacon, Lei Wang, Paul V. Gratz, Daniel A. Jimenez, Elvira Teran, Seth Pugsley, and Jinchun Kim. The championship simulator: Architectural simulation for education and competition, 2022. URL: https://arxiv.org/abs/2210.14324, arXiv:2210.14324.
8. Hossein Golestani, Rathijit Sen, Vinson Young, and Gagan Gupta. Calipers: a criticality-aware framework for modeling processor performance. In *Proceedings of the 36th ACM International Conference on Supercomputing*, ICS '22, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3524059.3532390.
9. Tony Nowatzki, Jaikrishnan Menon, Chen-Han Ho, and Karthikeyan Sankaralingam. Architectural simulators considered harmful. *IEEE Micro*, 35(6):4–12, 2015. doi:10.1109/MM.2015.74.
10. Cbp2025 simulator framework. https://ericrotenberg.wordpress.ncsu.edu/cbp2025-simulator-framework/, 2025.
11. Sizhuo Zhang, Andrew Wright, Thomas Bourgeat, and Arvind. Composable building blocks to open up processor design. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-51, page 68–81. IEEE Press, 2018. doi:10.1109/MICRO.2018.00015.
12. Thomas Bourgeat, Clément Pit-Claudel, Adam Chlipala, and Arvind. The essence of bluespec: a core language for rule-based hardware design. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 243–257, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3385412.3385965.
13. Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*, pages 1216–1225, 2012. doi:10.1145/2228360.2228584.
14. Verilator. Verilator user's guide. https://www.veripool.org/guide/latest/, 2026.
15. Haoyuan Wang and Scott Beamer. Repcut: Superlinear parallel rtl simulation with replication-aided partitioning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 572–585, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3582016.3582034.
16. Kexing Zhou, Yun Liang, Yibo Lin, Runsheng Wang, and Ru Huang. Khronos: Fusing memory access for improved hardware rtl simulation. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, page 180–193, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3613424.3614301.
17. Haoyuan Wang, Thomas Nijssen, and Scott Beamer. Don't repeat yourself! coarse-grained circuit deduplication to accelerate rtl simulation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, ASPLOS '24, page 79–93, New York, NY, USA, 2025. Association for Computing Machinery. doi:10.1145/3622781.3674184.
18. Mahyar Emami, Thomas Bourgeat, and James R. Larus. Parendi: Thousand-way parallel rtl simulation. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '25, page 783–797, New York, NY, USA, 2025. Association for Computing Machinery. doi:10.1145/3676641.3716010.
19. Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanović. Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, page 29–42. IEEE Press, 2018. doi:10.1109/ISCA.2018.00014.
20. Sagar Karandikar, Albert Ou, Alon Amid, Howard Mao, Randy Katz, Borivoje Nikolić, and Krste Asanović. Fireperf: Fpga-accelerated full-system hardware/software performance profiling and co-design. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 715–731, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3373376.3378455.
21. Mahyar Emami, Sahand Kashani, Keisuke Kamahori, Mohammad Sepehr Pourghannad, Ritik Raj, and James R Larus. Manticore: Hardware-accelerated rtl simulation with static bulk-synchronous parallelism. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, pages 219–237, 2023. doi:10.1145/

3623278.3624750.

22. Fares Elsabbagh, Shabnam Sheikhha, Victor A Ying, Quan M Nguyen, Joel S Emer, and Daniel Sanchez. Accelerating rtl simulation with hardware-software co-design. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 153–166, 2023. doi:10.1145/3613424.3614257.
23. Christopher Celio, David A Patterson, and Krste Asanovic. The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-167*, 2015. URL: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-167.html.
24. Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. Sonicboom: The 3rd generation berkeley out-of-order machine. In *Fourth Workshop on Computer Architecture Research with RISC-V*, volume 5, pages 1–7, 2020. URL: https://people.eecs.berkeley.edu/~krste/papers/SonicBOOM-CARRV2020.pdf.
25. Christopher Celio, Pi-Feng Chiu, Borivoje Nikolic, David A Patterson, and Krste Asanovic. BOOMv2: an open-source out-of-order RISC-V core. In *First Workshop on Computer Architecture Research with RISC-V (CARRV)*, 2017. URL: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-157.pdf.
26. Kaifan Wang, Jian Chen, Yinan Xu, Zihao Yu, Zifei Zhang, Guokai Chen, Xuan Hu, Linjuan Zhang, Xi Chen, Wei He, Dan Tang, Ninghui Sun, and Yungang Bao. XiangShan: An Open-Source Project for High-Performance RISC-V Processors Meeting Industrial-Grade Standards . In *2024 IEEE Hot Chips 36 Symposium (HCS)*, pages 1–25, Los Alamitos, CA, USA, August 2024. IEEE Computer Society. URL: https://doi.ieeecomputersociety.org/10.1109/HCS61935.2024.10665293, doi:10.1109/HCS61935.2024.10665293.
27. Chen Chen, Xiaoyan Xiang, Chang Liu, Yunhai Shang, Ren Guo, Dongqi Liu, Yimin Lu, Ziyi Hao, Jiahui Luo, Zhijian Chen, Chunqiang Li, Yu Pu, Jianyi Meng, Xiaolang Yan, Yuan Xie, and Xiaoning Qi. Xuantie-910: A commercial multi-core 12-stage pipeline out-of-order 64-bit high performance risc-v processor with vector extension: Industrial product. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 52–64. IEEE, 2020. doi:10.1109/ISCA45697.2020.00016.
28. Chen Bai, Qi Sun, Jianwang Zhai, Yuzhe Ma, Bei Yu, and Martin DF Wong. Boom-explorer: Risc-v boom microarchitecture design space exploration framework. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9. IEEE, 2021. doi:10.1109/ICCAD51958.2021.9643455.
29. Siddharth Gupta, Yuanlong Li, Qingxuan Kang, Abhishek Bhattacharjee, Babak Falsafi, Yunho Oh, and Mathias Payer. Imprecise store exceptions. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3579371.3589087.
30. Moein Ghaniyoun, Kristin Barber, Yuan Xiao, Yinqian Zhang, and Radu Teodorescu. Teesec: Pre-silicon vulnerability discovery for trusted execution environments. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3579371.3589070.
31. Luming Wang, Xu Zhang, Songyue Wang, Zhuolun Jiang, Tianyue Lu, Mingyu Chen, Siwei Luo, and Keji Huang. Asynchronous memory access unit: Exploiting massive parallelism for far memory access. *ACM Trans. Archit. Code Optim.*, 21(3), September 2024. doi:10.1145/3663479.
32. Duo Wang, Mingyu Yan, Yihan Teng, Dengke Han, Haoran Dang, Xiaochun Ye, and Dongrui Fan. A transfer learning framework for high-accurate cross-workload design space exploration of cpu. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 1–9, 2023. doi:10.1109/ICCAD57390.2023.10323840.
33. Hideki Ando. Performance improvement by prioritizing the issue of the instructions in unconfident branch slices. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 82–94, 2018. doi:10.1109/MICRO.2018.00016.
34. Yinan Xu, Zihao Yu, Dan Tang, Guokai Chen, Lu Chen, Lingrui Gou, Yue Jin, Qianruo Li, Xin Li, Zuojun Li, Jiawei Lin, Tong Liu, Zhigang Liu, Jiazhan Tan, Huaqiang Wang, Huizhe Wang, Kaifan Wang, Chuanqi Zhang, Fawang Zhang, Linjuan Zhang, Zifei Zhang, Yangyang Zhao, Yaoyang Zhou, Yike Zhou, Jiangrui Zou, Ye Cai, Dandan Huan, Zusong Li, Jiye Zhao, Zihao Chen, Wei He, Qiyuan Quan, Xingwu Liu, Sa Wang, Kan Shi, Ninghui Sun, and Yungang Bao. Towards developing high performance risc-v processors using agile methodology. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1178–1199, 2022. doi:10.1109/MICRO56248.2022.00080.
35. Championship value prediction. https://microarch.org/cvp1/. Accessed: 2025-02-20.
36. Google workload traces version 2. https://console.cloud.google.com/storage/browser/external-traces-v2. Accessed: 2025-02-20.
37. Wei Su, Abhishek Dhanotia, Carlos Torres, Jayneel Gandhi, Neha Gholkar, Shobhit Kanaujia, Maxim Naumov, Kalyan Subramanian, Valentin Andrei, Yifan Yuan, and Chunqiang Tang. Dcperf: An open-source, battle-tested performance benchmark suite for datacenter workloads. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, ISCA '25, page 1717–1730, New York, NY, USA, 2025. Association for Computing Machinery. doi:10.1145/3695053.3731411.
38. OpenXiangShan. XiangShan. https://github.com/OpenXiangShan/XiangShan, 2020.
39. SpinalHDL. Scala based hdl. https://github.com/SpinalHDL/SpinalHDL, 2024.
40. Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, page 45–57, New York, NY, USA, 2002. Association for Computing Machinery. doi:10.1145/605397.605403.
41. Alen Sabu, Harish Patil, Wim Heirman, and Trevor E Carlson. Looppoint: Checkpoint-driven sampled simulation for multi-threaded applications. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 604–618. IEEE, 2022. doi:10.1109/HPCA53966.2022.00051.
42. Trevor E Carlson, Wim Heirman, Kenzo Van Craeynest, and Lieven Eeckhout. Barrierpoint: Sampled simulation

of multi-threaded applications. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 2–12. IEEE, 2014. doi:10.1109/ISPASS.2014.6844456.

43. Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, and John Koenig. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 4:6–2, 2016. URL: https://aspire.eecs.berkeley.edu/wp/wp-content/uploads/2016/04/Tech-Report-The-Rocket-Chip-Generator-Beamer.pdf.
44. Bruno Sá, Luca Valente, José Martins, Davide Rossi, Luca Benini, and Sandro Pinto. CVA6 RISC-V virtualization: Architecture, microarchitecture, and design space exploration. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2023. doi:10.1109/TVLSI.2023.3302837.
45. RISC-V community. Olympia. https://github.com/riscv-software-src/riscv-perf-model, 2026.
46. Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, page 190–200, New York, NY, USA, 2005. Association for Computing Machinery. doi:10.1145/1065010.1065034.
47. Derek Bruening, Evelyn Duesterwald, and Saman Amarasinghe. Design and implementation of a dynamic optimization framework for windows. In *4th ACM workshop on feedback-directed and dynamic optimization (FDDO-4)*, page 20, 2001.
48. Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, page 89–100, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1250734.1250746.
49. DynamoRIO. Dynamorio trace format. https://dynamorio.org/sec_drcachesim_format.html. Accessed: 2026-02-07.
50. Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, pages 10–5555. California, USA, 2005. URL: https://www.usenix.org/legacy/event/usenix05/tech/freenix/full_papers/bellard/bellard.pdf.
51. Santosh Pandey, Amir Yazdanbakhsh, and Hang Liu. Tao: Re-thinking dl-based microarchitecture simulation. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 8(2):1–25, 2024. doi:10.1145/3656012.
52. Muhammad E. S. Elrabaa, Ayman Hroub, Muhamed F. Mudawar, Amran Al-Aghbari, Mohammed Al-Asli, and Ahmad Khayyat. A very fast trace-driven simulation platform for chip-multiprocessors architectural explorations. *IEEE Transactions on Parallel and Distributed Systems*, 28(11):3033–3045, 2017. doi:10.1109/TPDS.2017.2713782.
53. OpenXiangShan. XS-gem5. https://github.com/OpenXiangShan/GEM5, 2020.
54. John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006. doi:10.1145/1186736.1186737.
55. James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. SPEC CPU2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 41–42, 2018. doi:10.1145/3185768.3185771.
56. OpenXiangShan. NEMU. https://github.com/OpenXiangShan/NEMU, 2019.
57. Ahmad Yasin. A top-down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 35–44, 2014. doi:10.1109/ISPASS.2014.6844459.
58. Andrej Karpathy. llama2.c: Inference Llama 2 in one file of pure C. https://github.com/karpathy/llama2.c. Accessed: 2026-02-07.
59. RISC-V. RISC-V Instruction Set Manual. https://github.com/riscv/riscv-isa-manual. Accessed: 2026-02-07.
60. Javier D. Bruguera. Low-latency and high-bandwidth pipelined radix-64 division and square root unit. In *2022 IEEE 29th Symposium on Computer Arithmetic (ARITH)*, pages 10–17, 2022. doi:10.1109/ARITH54963.2022.00012.
61. Vivekananda M Vedula, Jacob A Abraham, Jayanta Bhadra, and Raghuram Tupuri. A hierarchical test generation approach using program slicing techniques on hardware description languages. *Journal of Electronic Testing*, 19:149–160, 2003. doi:10.1023/A:1022885523034.
62. Lingyi Liu and Shobha Vasudevan. Efficient validation input generation in rtl by hybridized source code analysis. In *2011 Design, Automation & Test in Europe*, pages 1–6, 2011. doi:10.1109/DATE.2011.5763253.
63. Biruk Mammo, Jim Larimer, Matthew Morgan, Dave Fan, Eric Hennenhoefer, and Valeria Bertacco. Architectural trace-based functional coverage for multiprocessor verification. In *2012 13th International Workshop on Microprocessor Test and Verification (MTV)*, pages 1–5, 2012. doi:10.1109/MTV.2012.12.
64. Sotiris Apostolakis, Chris Kennelly, Xinliang David Li, and Parthasarathy Ranganathan. Necro-reaper: Pruning away dead memory traffic in warehouse-scale computers. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '25, page 689–703, New York, NY, USA, 2025. Association for Computing Machinery. doi:10.1145/3676641.3716007.
65. Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. Accel-sim: An extensible simulation framework for validated gpu modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 473–486, 2020. doi:10.1109/ISCA45697.2020.00047.
66. Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 163–174, 2009. doi:10.1109/ISPASS.2009.4919648.
67. Onur Mutlu, Hyesoon Kim, David N Armstrong, and Yale N Patt. An analysis of the performance impact of wrong-path memory references on out-of-order and runahead execution processors. *IEEE Transactions on Computers*, 54(12):1556–1571, 2005. doi:10.1109/TC.2005.190.

68. Stijn Eyerman, Sam Van den Steen, Wim Heirman, and Ibrahim Hur. Simulating wrong-path instructions in decoupled functional-first simulation. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 124–133. IEEE, 2023. doi:10.1109/ISPASS57527.2023.00021.
69. Bhargav Reddy Godala, Sankara Prasad Ramesh, Krishnam Tibrewala, Chrysanthos Pepi, Gino Chacon, Svilen Kanev, Gilles A Pokam, Daniel A Jiménez, Paul V Gratz, and David I August. Correct wrong path. *arXiv preprint arXiv:2408.05912*, 2024. URL: https://doi.org/10.48550/arXiv.2408.05912.
70. Resit Sendag, Ayse Yilmazer, Joshua J. Yi, and Augustus K. Uht. The impact of wrong-path memory references in cache-coherent multiprocessor systems. *Journal of Parallel and Distributed Computing*, 67(12):1256–1269, 2007. Best Paper Awards: 20th International Parallel and Distributed Processing Symposium (IPDPS 2006). URL: https://www.sciencedirect.com/science/article/pii/S0743731507000457, doi:10.1016/j.jpdc.2007.03.005.
71. R. Sendag, A. Yilmazer, J.J. Yi, and A.K. Uht. Quantifying and reducing the effects of wrong-path memory references in cache-coherent multiprocessor systems. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, pages 10 pp.–, 2006. doi:10.1109/IPDPS.2006.1639260.
72. Stephen R Goldschmidt and John L Hennessy. The accuracy of trace-driven simulations of multiprocessors. *ACM SIGMETRICS Performance Evaluation Review*, 21(1):146–157, 1993. doi:10.1145/166962.167001.
73. Karthik Sangaiah, Michael Lui, Radhika Jagtap, Stephan Diestelhorst, Siddharth Nilakantan, Ankit More, Baris Taskin, and Mark Hempstead. Synchrotrace: Synchronization-aware architecture-agnostic traces for lightweight multicore simulation of cmp and hpc workloads. *ACM Trans. Archit. Code Optim.*, 15(1), March 2018. doi:10.1145/3158642.
74. Jian Weng, Boyang Han, Derui Gao, Ruijie Gao, Wanning Zhang, An Zhong, Ceyu Xu, Jihao Xin, Yangzhixin Luo, Lisa Wu Wills, and Marco Canini. Assassyn: A unified abstraction for architectural simulation and implementation. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, ISCA '25, page 1464–1479, New York, NY, USA, 2025. Association for Computing Machinery. doi:10.1145/3695053.3731004.
75. Josué Feliu, Arthur Perais, Daniel A. Jiménez, and Alberto Ros. Rebasing microarchitectural research with industry traces. In *2023 IEEE International Symposium on Workload Characterization (IISWC)*, pages 100–114, 2023. doi:10.1109/IISWC59245.2023.00027.