2026 Design, Automation & Test in Europe Conference (DATE 2026)

CISim: ISA-Agnostic Custom Instruction Simulation for General-Purpose Processor

Xiaoyu Hao†, Sen Zhang, Liang Qiao, Jun Shi, Junshi Chen*, ‡, Hong An*, ‡
School of Computer Science and Technology, University of Science and Technology of China

***Abstract*—Pre-RTL ISA-agnostic simulators have been established for designing heterogeneous systems, but few of them are suitable for evaluating a general-purpose processor (GPP) with custom instructions (CIs). MosaicSim [1], a state-of-the-art ISA-agnostic simulator, still has several limitations for CI design and simulation. First, it shows inaccuracy in simulating GPPs due to an oversimplified performance model. Second, as designed for kernel simulation, it lacks support for running complex real-world benchmarks. Third, it cannot evaluate fine-grained irregular CIs due to the lack of the ability to represent or define them in benchmarks. To this end, we propose CISim, a new ISA-agnostic simulation framework containing an offloader that generates and integrates CIs into benchmarks, along with a simulator capable of executing benchmarks with CIs. Evaluations show that CISim is accurate by validating against Gem5 [2] and achieves higher accuracy than MosaicSim. A case study evaluating CI exploration methods highlights the strength and flexibility of CISim.**

***Index Terms*—CPU Simulator, Custom Instruction, ISA-Agnostic Simulation, Performance Modeling**

I. Introduction

Extending a base general-purpose instruction set architecture (ISA) with custom instructions (CIs) improves both performance and energy efficiency of processors. The rapid evolution of domain algorithms and applications drives a need for pre-RTL simulators in CI design. Currently, using existing CPU simulators [2]–[6], designed for real-world ISAs, is the primary method for exploring and evaluating ISA extensions. As another trend, some works [1], [7], [8] aim to build ISA-agnostic simulators to guide specialized hardware design, which are decoupled from legacy ISAs and eliminate the need to simulate ISA-dependent characteristics of workloads [9].

Building a simulator should address several challenges, including accuracy, simulation speed, the ability to execute a wide range of workloads, and flexibility [4]. When designing CIs for general-purpose processors (GPPs), the traditional compiler-simulator paradigm faces the issue of requiring additional effort besides modifying simulators, such as inevitable modifications to compilers and benchmarks. Although ISA-agnostic simulators have shown strength in designing specialized hardware, few of them can be used for CI simulation. Gem5-Aladdin [8] and Gem5-Salam [7] are not entirely ISA-agnostic, relying on Gem5 [2] to simulate GPPs, as they are designed for accelerators in heterogeneous systems. Only MosaicSim [1] provides an ISA-agnostic GPP model, but it poses three limitations in addressing the above challenges for CI simulation. First, it provides insufficient accuracy due to an oversimplified performance model of GPP, which may produce misleading results in CI design. Second, it only simulates kernel-based benchmarks and lacks support for complex benchmarks whose code regions of interest to simulate are hard to represent as kernels. Third, MosaicSim is unsuitable for evaluating GPPs with irregular CIs because it cannot represent or provide an interface to define them in benchmarks, particularly given that such CIs can be located anywhere within a benchmark.

This work was supported by the Strategic Priority Research Program of Chinese Academy of Sciences (Grant No.XDB0500102), Laoshan Laboratory (No.LSKJ202300305). *Corresponding authors: Junshi Chen and Hong An. †Email: haoxiaoyu@mail.ustc.edu.cn. ‡Also with Laoshan Laboratory, China.

To this end, we establish a new ISA-agnostic simulation framework based on LLVM [10] toolchain for **C**ustom **I**nstruction **Sim**ulation (CISim) of GPPs. CISim includes an offloader that automatically customizes benchmarks with CIs, which provides a proper representation for custom and corresponding data-moving instructions in LLVM IR. It eliminates the need to modify compilers or benchmarks, providing an easy-to-use interface for defining new instructions. CISim also has a cycle- and trace-driven simulator that uses LLVM IR as the ISA, allowing evaluation of the GPP performance and extended ability to run benchmarks with new CIs. CISim can execute complex real-world benchmarks and achieve high accuracy through a detailed performance model. We implement an LLVM-based basic block vector (BBV) collection method, working with Simpoint [11] to select representative code chunks, and provide another LLVM pass to instrument and collect traces for selected chunks used in simulation. Our contributions can be summarized as:

- We introduce CISim, a framework that uses an offloader to automatically generate and integrate CIs into benchmarks in LLVM IR and a GPP simulator cooperating with the offloader to run real-world benchmarks with CIs, bridging the gap between CI representation and simulation.
- Evaluation shows CISim achieves low IPC errors of 5.0%, 6.4%, and 11.4% when validated against Gem5 on three 2-/4-/6-wide out-of-order cores using benchmarks from SPEC2006 [12] and SPEC2017 [13]. Also, CISim achieves lower IPC errors than MosaicSim using MachSuite [14] on these cores.
- We present a case study that utilizes CISim to evaluate CI exploration methods, demonstrating the strength of CISim and yielding two observations.

II. Background and Motivation

A. Tightly Integrated Custom Instruction

Fig. 1 shows the tight integration of a custom functional unit (FU) to run CIs. This approach requires modifications to

978-3-9826741-1-7/DATE26/© 2026 EDAA
Authorized licensed use limited to: University of Science & Technology of China. Downloaded on June 15,2026 at 07:27:05 UTC from IEEE Xplore. Restrictions apply.

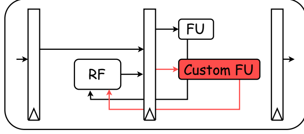

Fig. 1: A tightly integrated custom FU to run CIs.

various pipeline stages to recognize and handle new custom opcodes [15]–[17]. Implementation in the host core's pipeline is ideal for low-latency operations such as simple combinational or short multi-cycle CIs, e.g, ARM Custom Instruction [18]. More complex instructions with high latency or those that are supposed to run in parallel with the host processor may be suitably implemented as coprocessors, e.g., RoCC [19]. In this paper, we focus on CIs whose latency can be represented as a single value and identified by fusing operations within a basic block. For larger code regions, works [20], [21] propose generating benchmarks targeting hot paths comprising multiple basic blocks and even loops. They require modeling an accelerator to run such regions that include branch and memory operations, which is not the problem this paper aims to solve.

B. Current Simulation Approach and Limitation

1) Compiler-Simulator Approach: Evaluating CI with this approach requires modifying simulators, compilers, and source code of benchmarks, causing significant effort beyond the architectural design from researchers. For example, Gem5 [2] is one of the most popular and influential simulators for studying processor architectures, providing a domain-specific language to define new instructions. However, it requires the application to be compiled into a binary executable, making it inevitable to extend mature compilers like GCC [22] or LLVM and modify the source code with new instructions. To solve this, TDG [23], [24] proposes a graph-based approach to model various accelerators, relying on critical path analysis, but it is inaccurate in contention modeling [25].

2) ISA-agnostic Simulator: ISA-agnostic simulators [1], [7], [8] typically use LLVM IR as the ISA. MosaicSim [1] is a state-of-the-art simulator designed for heterogeneous systems, modeling performance for both GPPs and accelerators. However, it has three limitations for evaluating a GPP with CIs.

(1) MosaicSim is not accurate enough due to the absence of an instruction cache (ICache), unmodeled pipelined FU behaviour, an oversimplified instruction fetch strategy, and an inappropriate branch predictor training method that trains the branch predictor immediately after predictions are made, rather than during commit. **(2)** It only simulates benchmarks represented as well-defined kernels, restricting MosaicSim's usage to the GPP design that requires simulating complex benchmarks, whose code regions of interest can cross several basic blocks with complex control-flow and nested function calls, making them hard to extract as kernels. Running benchmarks with billions of instructions can also be problematic with MosaciSim, particularly without supporting an acceleration mechanism like Simpoint [11], due to the absence of a tool that can collect BBV and instruction traces of a selected chunk at the LLVM IR level. **(3)** It cannot customize benchmarks with CIs without representing them properly. Representing new instructions as function calls in LLVM IR is a straightforward idea, which has been used previously to represent coarse-grained accelerators [7], [8], but it faces challenges for irregular CIs. These CIs may appear anywhere in an application and contain a few operations that are part of a basic block, which can hardly be extracted and defined as functions manually. Representing CIs as functions raises a problem that their arguments and return values should be meaningful in simulation as instructions. CIs should be designed under architectural constraints, including register ports and instruction bit width, which limit the number of operands of instructions. However, a beneficial CI may have a larger number of operands than the limit, so special-purpose registers (SPRs) [26] and data-moving instructions [27] should be introduced. A proper representation is required for custom and corresponding data-moving instructions that encode both scalar general-purpose registers (GPRs) and SPRs, which is not possible with existing simulators.

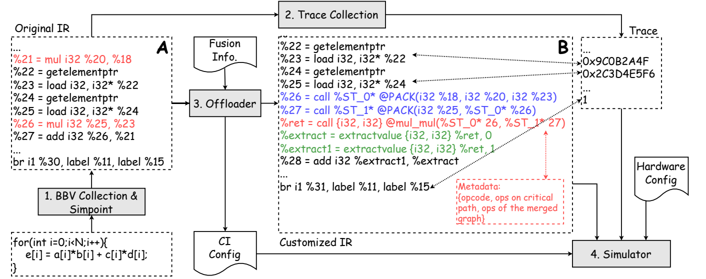

Fig. 2: The workflow of CISim. In this example, two `muls` are fused as a CI. A and B are original and customized IR, respectively, for a core allowing three source operands at most.

III. CISIM

A. Overview

Fig. 2 illustrates the workflow of CISim, where the shaded boxes represent four key steps that are detailed below.

1) Preparation: The process begins by compiling an application to LLVM IR and applying optimization passes. In step 1, we perform an instrumenting pass to collect BBVs as input to Simpoint, which is optional. In step 2, we collect traces for a selected code chunk or the entire application using another instrumenting pass, which counts the number of executed instructions and records traces for a specific interval. Both collections of BBV and trace require the instrumented IRs to be compiled into executables and run on a real machine.

2) Offloader: In step 3, the offloader generates CIs based on fusion information indicating which instructions to fuse. In our example, two multiply instructions `mul` (%21 and %26) in the original IR (Fig. 2A) are fused as a new CI `mul_mul`, which has four inputs and two outputs. Considering the limitations of instruction bit width or allowed register ports per instruction of the host core, encoding all operands in an instruction can be hard. Therefore, we assume that this CI accesses a SPR with the aid of data-moving instructions. Fig. 2B shows the

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on June 15,2026 at 07:27:05 UTC from IEEE Xplore. Restrictions apply.

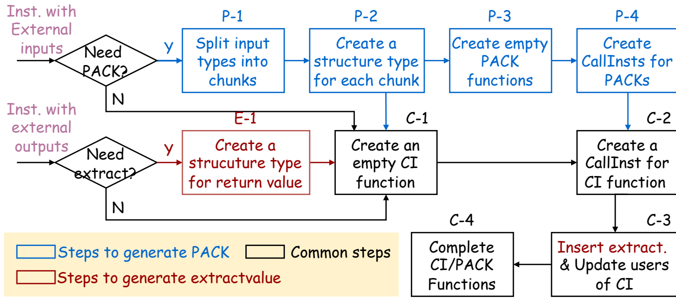

Fig. 3: Detailed steps for generating customized IR.

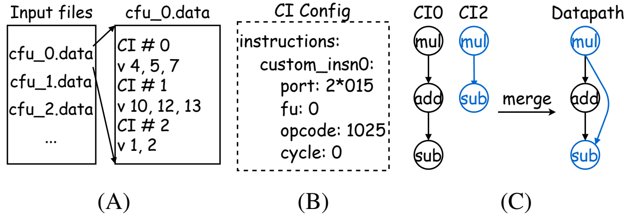

Fig. 4: (A) Input configuration files to the offloader (Fusion Info. in Fig. 2). (B) CI configuration template output by the offloader. (C) Merged datapath for CIs of different operations.

customized IR. The offloader inserts two `PACK` instructions to pack operands into two structures that represent the explicit transfer of data from scalar GPRs to a SPR. Two structures, each containing part of the operands, are then passed into the function `mul_mul`. Since LLVM allows only one return value per instruction, the return value of this CI is also a structure. The IR `extractvalue` is used to extract values from the returned structure, which represents data movements from a SPR (%ret) to two GPRs (%extract and %extract1).

3) Simulator: In step 4, the simulator runs benchmarks with/without CIs, requiring static IR, the collected trace, and two configuration files: one for hardware parameters and general instructions, and another for CIs. It begins at the basic block where the trace starts, requiring reading and assigning values from the trace to static IR to drive the simulation, like in Fig. 2B, a branch decision is read to decide which basic block to run next. CISim only allows fusing arithmetic instructions and never changes the order of memory and branch instructions, so trace collection is required once for different CI configurations.

B. BBV and Trace Collection

1) BBV Collection: BBV is a vector of how many times each basic block is executed of an application, which is collected within fixed intervals, e.g., every 100 million instructions. To collect BBV, we insert a function to count how many times a basic block has executed at the beginning of each basic block. We insert another function at the end of each basic block to count the number of executed instructions in the current interval and dump BBV to a file when reaching the interval end.

2) Trace Collection: We count the number of executed instructions at the end of every basic block and check at the beginning of each basic block to see if the interval has been reached. We insert the collecting functions before each corresponding IR to collect load/store addresses and branch/switch decisions, as well as names of basic blocks at the beginning of each block. Collecting functions use a flag to check if the collection is enabled each time they are called. The flag is set to be true within the target interval only. Basic block names are dumped to support indirect function calls, which are invoked via function pointers that are unknown in static IR. We also record the basic block name at the interval's start to indicate where to start a simulation. To avoid the cold-start simulation, CISim supports collecting additional traces for warming up before the selected chunk.

C. Offloader

1) Input and Output: The input to the offloader, specifying the fusion information, is shown in Fig. 4A. It consists of a list of files, each defining a custom FU and CIs that execute on it. A CI is defined in two lines. The first line, e.g., CI # 0, indicates that this is a new instruction, and the second line, starting with a letter v, is a list of unique static IDs representing operations to fuse. One can use an LLVM pass to identify which instructions to fuse and generate this configuration easily. Static IDs are assigned by traversing the IR hierarchy in the order of Module, Function, Basic Block, and Instruction.

Besides customized IR, the offloader also outputs a CI configuration template (CI Config in Fig. 2) as input to the simulator. Fig. 4B shows its format, where each CI has four fields (the same as general instructions). We use the representation proposed by [28] to indicate the port usage of an instruction. A data-moving instruction's latency is specified in the cycle field. A CI's cycle is calculated dynamically in simulation using operations on the critical path of its dataflow graph, which are stored in IR's metadata. Metadata also includes opcodes of new instructions automatically assigned by our offloader.

2) Datapath Generation: Fig. 4A shows CIs with a different number of operations that run on a custom FU. To share hardware resources among them, as shown in Fig. 4C, the offloader merges dataflow graphs of CIs to generate a shared datapath using the method proposed in [27]. CIs defined in a file execute on the same custom FU, which is specified in the FU field, but they are represented using different function names. Operations of the merged graph are also stored in metadata. These operations, as well as operations on CI's critical path, are for further use in estimating power/area.

3) Code Generation: The idea of automatically customizing IR was previously used by researchers to discover CIs [29]. However, their implementation creates redundant basic blocks in benchmarks, preserving both original and customized basic blocks, which causes incorrect functionality and prevents benchmarks from running on a simulator. We improve it by preserving only customized basic blocks. Fig. 3 shows the implementation steps. It first recognizes the instruction sequence to fuse as a CI according to the fusion information, and identifies instructions with external inputs and outputs. After that, steps from *P-1* to *P-4* generate `PACK`s, if a CI has more input operands than a predefined threshold. If a CI has more than one return value, step *E-1* is required to create a structure

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on June 15,2026 at 07:27:05 UTC from IEEE Xplore. Restrictions apply.

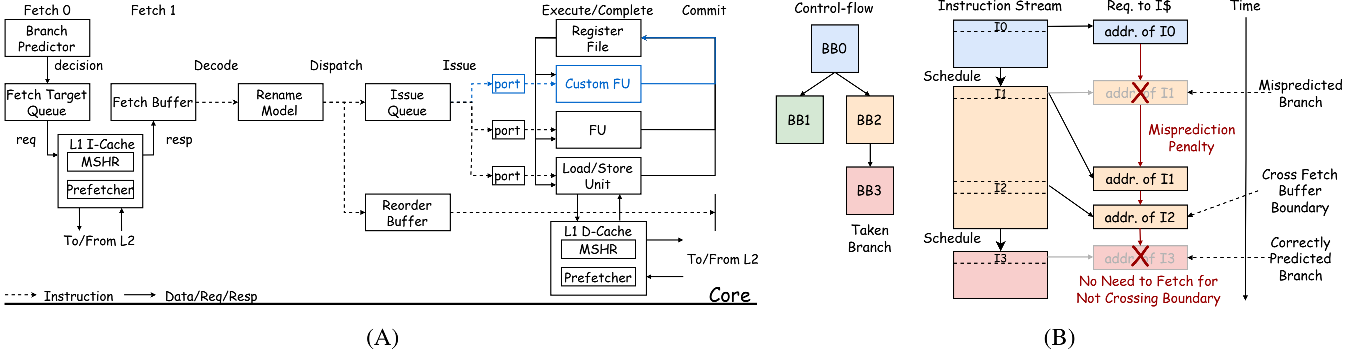

Fig. 5: (A) The pipeline organization and key components of the cycle-driven simulator. (B) A fetch is triggered when the next PC crosses the fetch buffer boundary. A new basic block is scheduled when reaching a terminator instruction. We model the misprediction penalty before sending a fetch request. This figure assumes `I3` is in the same fetch buffer as `I2`.

type for output values, and `extractvalues` are inserted to extract values in step *C-3*. Steps from *C-1* to *C-4* generate CI functions. More details can be found in [29].

This implementation benefits CI simulation from three aspects. First, it generates custom and data-moving instructions with correct data dependencies, which is the foundation of simulation. Second, it builds dependencies between `PACK`s to abstract the case of sequentially transferring data from multiple GPRs to specific positions of a single SPR, with each transfer preserving the SPR's previous data at other positions. As shown in Fig. 2B, the output of `PACK %26` is passed to `PACK %27`. We improve it by omitting this dependency if required to move data to separate SPRs. Third, it preserves correct semantics for data-moving. A CI function directly returns a structure instead of a pointer because extracting values from a pointer introduces other instructions like `load`, which is not consistent with the semantics of moving data between GPRs and SPRs, as well as lacks recorded addresses in the trace. In contrast, `PACK` returns pointers to structures allocated using `malloc`, since they are used in CI functions and are never executed in simulation.

D. Performance Modeling

The simulator is implemented based on the microarchitecture shown in Fig. 5A. It has eight pipeline stages: Fetch 0, Fetch 1, Decode, Rename, Dispatch, Issue, Execute/Complete, and Commit. Decode is an empty stage for future extension. Complete is used to mark the end of an instruction execution.

1) Fetch & Schedule: The fetch contains two stages. Fetch 0 sends fetch requests to the ICache from the fetch target queue (FTQ). Fetch 1 waits and receives responses from ICache and fetches instructions into a fetch queue. Since traces are collected in a real machine, traces for those mispredicted branches are unavailable, and CISim runs only basic blocks on taken branches. Fig. 5B illustrates how CISim schedules basic blocks and generates fetch requests based on the control flow, where BB0, BB2, and BB3 are three basic blocks to execute. The schedule determines which basic block to run next based on the result of a terminator. It also converts static IR to dynamic instructions by reading and assigning traces, such as an address for a load, and builds data dependencies between instructions. After a basic block is scheduled, CISim splits it into several fetch requests according to the fetch buffer size and puts them into FTQ. In our example, BB0 and BB3 each have one request, and BB2 has two requests. Fetch 0 sends a fetch request when the next PC is going to cross the fetch buffer boundary, such as `I1` and `I2`. If a branch is predicted correctly and its PC falls within the fetch buffer, no fetch request is generated for the first PC of the target basic block, like `I3` of BB3. Fetch 0 schedules the next basic block before sending a fetch request, but only if the previous PC corresponds to a terminator, e.g., `I1`. `I2` invokes no schedule as its predecessor is not a terminator.

2) Branch Prediction: Fetch 0 uses decisions from the branch predictor to determine if a misprediction occurs. Since CISim cannot simulate mispredicted branches, we propose modeling the penalty before sending a fetch request to ICache. In Fig. 5B, BB1 is mispredicted to be taken, so a penalty occurs before sending a request (address of `I1`) to fetch BB2. In this case, CISim stalls fetch, waits until the branch decision is made, and then additionally stalls a predefined cycle.

3) Rename Model: LLVM IR uses an unlimited set of virtual registers, so the rename is not required, but we still need to manage physical register usage. The conventional approach frees a register after writing its related architectural register. Without architectural registers, it is hard to know when to free a physical register, so we use an analytical method to calculate register requirements. During scheduling, we assign unique dynamic IDs to instructions, each corresponding to a unique destination operand. After rebuilding data dependencies with these IDs, we can determine register requirements by counting unique IDs among both destination and source operands of instructions in the reorder buffer (ROB). For example, we calculate the number of allocated integer registers as $N_{reg_int}^{used} = N_{dst_int}^{all} + N_{src_int}^{prev}$. $N_{dst_int}^{all}$ is the number of registers allocated to integer destination operands of instructions in ROB, which can be easily managed. $N_{src_int}^{prev}$ is the number of registers consumed by source operands (of instructions in ROB) produced by instructions previously committed. To avoid repeatedly traversing ROB, we manage a map<op, count>, whose size is $N_{src_int}^{prev}$. The op is an operand ID. We insert items or increment counts after an instruction is renamed, and

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on June 15,2026 at 07:27:05 UTC from IEEE Xplore. Restrictions apply.

TABLE I: GPP Out-of-Order Cores

Core	Configuration
OOO2	2-wide pipe., 64-entry ROB, 32-entry IQ, 1 ALU, 1 FPU, 1 iMul/iDiv, 1 fpMul/fpDiv, 1 BRU, 1 LD/ST
OOO4	4-wide pipe., 192-entry ROB, 64-entry IQ, 3 ALU, 2 FPU, 1 iMul/iDiv, 1 fpMul/fpDiv, 1 BRU, 2 LD/ST
OOO6	6-wide pipe., 512-entry ROB, 128-entry IQ, 4 ALU, 3 FPU, 2 iMul/iDiv, 2 fpMul/fpDiv, 2 BRU, 2 LD/ST
Memory	
L1I/L1D	32KB / 8-way / 3-cycle hit lat
L2	1MB / 16-way / 8-cycle hit lat
L3	8MB / 16-way / 34-cycle hit lat
DRAM	25GB/s BW / 118-cycle hit lat

decrement counts and remove items whose count decreases to zero when it is committed, both using this instruction's source operands produced by instructions that have left the ROB.

4) Issue & Execution: We use a simplified model for execution ports. For the instruction in Fig. 4B, we do not decompose it into two uops, each using one port, but instead, we assume it always has only one uop and occupies one of these ports for two cycles. FUs can be configured fully pipelined or not.

5) Function Call: A call instruction can be a normal function call, a CI, or a data-moving instruction. CISim identifies new instructions by checking if their metadata is not empty, and then parses the metadata to calculate the latency of CIs and obtain their opcode. CISim treats each custom or data-moving instruction as a single operation, never executing its function body. CISim uses a stack to manage general function calls. It pushes or pops a return address into/out of the stack when a `call`/`ret` instruction is being scheduled. When our simulator detects an indirect function, it gets the first basic block of the function based on its name recorded in the trace.

6) Cache and DRAM Model: CISim employs a cache and DRAM model similar to MosaicSim [1], but CISim supports ICache. We also realize fetch-directed prefetcher [30] in ICache and stride prefetcher [31] in DCache.

IV. Evaluation

A. Accuracy

1) Methodology: Validating the accuracy of CISim faces two challenges. First, microarchitectures of real-world processors have become increasingly complex, making it hard to model their behaviors without microarchitectural details. Second, CISim uses LLVM IR as the ISA, and the ISA difference often results in errors, such as validating it against a CISC x86 machine, despite LLVM IR being a load-store architecture. We therefore validate CISim by comparing instruction per cycle (IPC) with the Gem5 [2] RISC-V simulator, which has a similar pipeline organization to CISim. The accuracy is validated by configuring both CISim and Gem5 with the same architectural parameters for three out-of-order (OOO) cores as specified in Table I. All cores use the same memory system configuration, a TAGE [32] branch predictor with an 8-cycle flush penalty, a 64-byte fetch buffer, and one port for each FU. Validations use C/C++ benchmarks from SPEC2006 [12] and SPEC2017 [13]. We skip the first 10B instructions and report results for the next 10B instructions with the reference inputs.

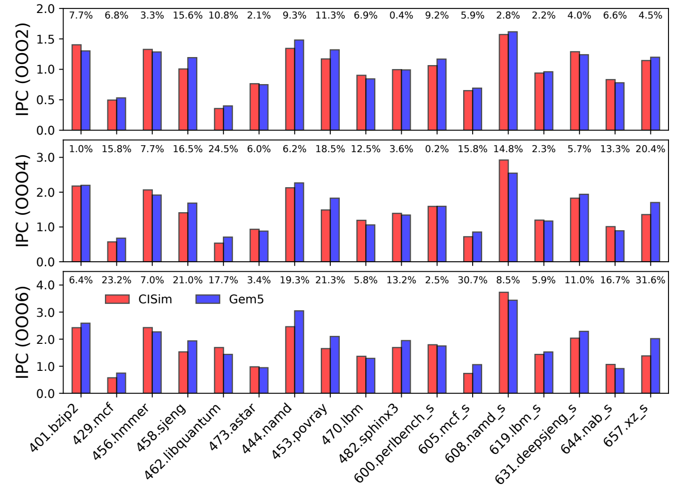

Fig. 6: IPC estimation of CISim and Gem5 on three cores.

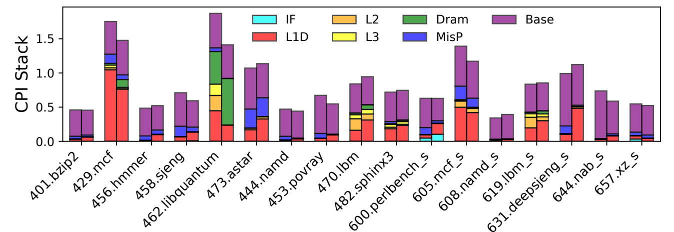

Fig. 7: CPI stacks of CISim (left) and Gem5 (right) on OOO4. IF is the penalty in instruction fetching due to ICache miss, and MisP is the penalty of branch misprediction. L1D, L2, L3, and DRAM are components of cycles spent on accessing L1 data, L2, L3 caches, and DRAM. The Base is the rest.

2) Result: Fig. 6 shows the IPC per benchmark achieved by CISim (left bars) and Gem5 (right bars) on three cores. The absolute errors of IPC are shown at the top of the bars. CISim achieves geometric mean errors of 5.0%, 6.4%, and 11.4% on OOO2, OOO4, and OOO6, respectively. CISim underestimates performance (lower IPC) on some benchmarks, contradicting the intuition that missing components in CISim may lead to overestimation, like those for address translation (e.g., TLB). It is due to the RISC-V "C" extension, which reduces much pressure on the frontend by fetching 16-bit instructions. Even though we have disabled it during compilation, compressed instructions are still present in pre-compiled external libraries. CISim cannot simulate functions in external libraries and will skip them since it is unable to convert them into the format of LLVM IR and collect their traces. To investigate the source of errors, we present CPI stacks of CISim (left) and Gem5 (right) on OOO4 in Fig. 7. It shows that CISim captures a similar breakdown to Gem5, indicating that the accuracy of CISim is not achieved by overestimating one component and underestimating another excessively.

3) Comparison with MosaicSim: Since MosaicSim only simulates kernels, we compare CISim with its GPP model

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on June 15,2026 at 07:27:05 UTC from IEEE Xplore. Restrictions apply.

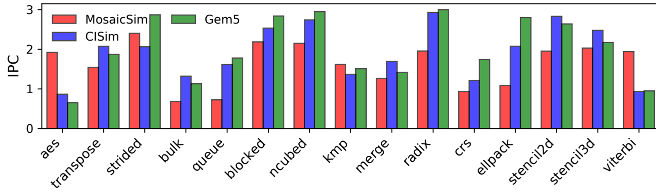

Fig. 8: IPC of MosaicSim, CISim, and Gem5 on OOO4.

using MachSuite [14] and report results of kernels only. Fig. 8 shows the estimated IPC of them, as well as Gem5 used as the reference. MosaicSim achieves a geometric mean error of 29.4%, while CISim achieves a lower error of 11.5%. CISim also achieves lower error rates on OOO2 (8.7% vs. 46.7%) and OOO6 (13.4% vs. 22.5%). Sources of errors in MosaicSim are discussed previously in Section II-B2.

CISim has a higher error rate using MachSuite than SPEC. CISim and Gem5 cover identical code regions (kernels) when running MachSuite. In contrast, when running SPEC, the same number of instructions may correspond to different code regions due to the ISA difference. This explains the divergent results.

4) Running Overhead: The simulation speed of CISim is 250 KIPS for the OOO4 core on an Intel Xeon CPU E5-2695 v4, slightly slower than MosaicSim's 280 KIPS, due to its more complex but more reliable performance model. To run SPEC with 10B instructions, CISim requires an average of 500 MB of runtime physical memory and tens of GB of disk storage (hundreds of MB for MachSuite).

B. Case Study: Evaluation for CI Exploration Methods

Automated CI exploration is important and well-studied [33]. Discovered CIs can be located anywhere and contain various operations, which pose difficulties in evaluating their benefits, but CISim provides a promising platform for them. Three works are selected to run, including Novia [27], CIExplorer [29], and MaxClique [34]. We revise them a little to generate inputs for our offloader. All experiments are performed using 200 million instructions selected by Simpoint and assume each CI's latency is the critical path of its dataflow graph, without considering any techniques that can reduce this latency. Custom FUs are fully pipelined. Both CI and data-moving instructions encode at most three source operands, and the latter require one cycle.

Observation #1: There seems to be no simple principle for exploring CIs that can benefit all cores. Fig. 9 shows the speedup achieved by all three methods on three cores in the top, and shows the coverage of dynamic custom instructions in the middle. Note that only CIExplorer can identify different CIs on three cores, thereby achieving different coverage shown with different suffixes, e.g., *-OOO2*. MaxClique has proven that for RISC in-order cores, a CI can achieve higher speedup with more operations, but this principle does not apply to OOO cores. CIExplorer produces performance improvements, but with the increase of instruction-level parallelism (ILP) of cores, its benefits decrease. Moreover, pursuing higher coverage of CIs does not always benefit performance. As shown in the middle, Novia and MaxClique have higher coverage, but they

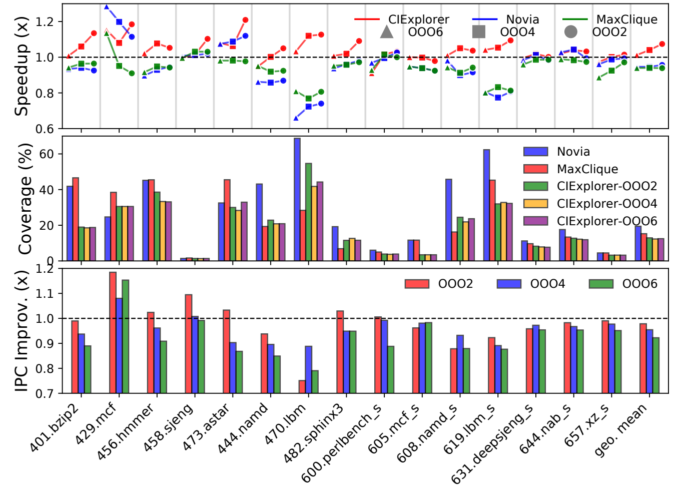

Fig. 9: The top shows speedups of three exploration methods on three cores. The middle shows the dynamic instruction coverage. The bottom shows IPC improvements of CIExplorer.

decrease the performance of all cores, because their identified CIs have too many operations and introduce too many data-moving instructions, obstructing operations that could run in parallel. Assuming two consecutive instructions without data dependency, `I1` and `I2`, each of them can start running when its operands are ready in an OOO core. However, after fusion, one must wait for additional operands from the other, particularly when these operands are from memory and cause miss penalties, thereby decreasing ILP. The more fused operations in cores of higher ILP, the larger the limitation. CIExplorer works best by using an ILP-aware cost model for host cores, but it still falls short in modeling this behavior, as it exploits no dynamic information about the branch and memory.

Observation #2: Be careful when fusing operations for improving throughput. The bottom of Fig. 9 shows the IPC of CIExplorer normalized to cores without CIs, revealing no improvement for most benchmarks. Put simply, `I1` and `I2` can be committed in parallel with an IPC of 2, but after fusion, IPC decreases to 1 (only one CI left). CIs can improve performance by reducing pipeline pressure and instruction count, but may limit ILP as discussed above. A possible way to enhance both performance and throughput is to fuse operations in a chain [35] and avoid fusing parallel operations, preferably with the help of an ILP-aware cost model and dynamic information.

V. Conclusion

We propose CISim for ISA-agnostic custom instruction simulation, intending to replace no simulator, but rather to validate an idea that may take a step toward addressing the inflexibility of the current simulation paradigm. We demonstrate that CISim is reliable by validating CISim against Gem5 on three out-of-order cores. CISim also achieves lower error rates than MosaicSim. After validation, we present a case study that applies CISim to evaluate various CI exploration methods, providing two key observations that demonstrate its strengths.

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on June 15,2026 at 07:27:05 UTC from IEEE Xplore. Restrictions apply.

REFERENCES

[1] O. Matthews, A. Manocha, D. Giri, M. Orenes-Vera, E. Tureci, T. Sorensen, T. J. Ham, J. L. Aragón, L. P. Carloni, and M. Martonosi, "Mosaicsim: A lightweight, modular simulator for heterogeneous systems," in *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2020, pp. 136–148.

[2] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj *et al.*, "The gem5 simulator: Version 20.0+," *arXiv preprint arXiv:2007.03152*, 2020.

[3] T. E. Carlson, W. Heirman, and L. 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*, 2011, pp. 1–12.

[4] D. Sanchez and C. Kozyrakis, "Zsim: Fast and accurate microarchitectural simulation of thousand-core systems," *ACM SIGARCH Computer architecture news*, vol. 41, no. 3, pp. 475–486, 2013.

[5] T. Austin, E. Larson, and D. Ernst, "Simplescalar: An infrastructure for computer system modeling," *Computer*, vol. 35, no. 2, pp. 59–67, 2002.

[6] N. Hardavellas, S. Somogyi, T. F. Wenisch, R. E. Wunderlich, S. Chen, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk, "Simflex: A fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture," *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, no. 4, pp. 31–34, 2004.

[7] S. Rogers, J. Slycord, M. Baharani, and H. Tabkhi, "gem5-salam: A system architecture for llvm-based accelerator modeling," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 471–482.

[8] Y. S. Shao, S. L. Xi, V. Srinivasan, G.-Y. Wei, and D. Brooks, "Co-designing accelerators and soc interfaces using gem5-aladdin," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

[9] Y. S. Shao and D. Brooks, "Isa-independent workload characterization and its implications for specialized architectures," in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2013, pp. 245–255.

[10] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[11] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," *ACM SIGPLAN Notices*, vol. 37, no. 10, pp. 45–57, 2002.

[12] "Spec cpu 2006," https://www.spec.org/cpu2006/.

[13] "Spec cpu 2017," https://www.spec.org/cpu2017/.

[14] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks, "Machsuite: Benchmarks for accelerator design and customized architectures," in *2014 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2014, pp. 110–119.

[15] A. Dolmeta, S. Di Matteo, E. Valea, M. Carmona, A. Loiseau, M. Martina, and G. Masera, "Tyrca: A risc-v tightly-coupled accelerator for code-based cryptography," in *2025 Design, Automation & Test in Europe Conference (DATE)*. IEEE, 2025, pp. 1–7.

[16] J. Oppermann, B. M. Damian-Kosterhon, F. Meisel, T. Mürmann, E. Jentzsch, and A. Koch, "Longnail: High-level synthesis of portable custom instruction set extensions for risc-v processors from descriptions in the open-source coredsl language," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2024, pp. 591–606.

[17] M. Damian, J. Oppermann, C. Spang, and A. Koch, "Scaie-v: an open-source scalable interface for isa extensions for risc-v processors," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 169–174.

[18] J. Yiu, "Innovate by customized instructions, but without fragmenting the ecosystem," https://armkeil.blob.core.windows.net/developer/Files/pdf/white-paper/arm-custom-instructions-without-fragmentation-whitepaper.pdf.

[19] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz *et al.*, "The rocket chip generator," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, vol. 4, pp. 6–2, 2016.

[20] S. Kumar, N. Sumner, V. Srinivasan, S. Margerm, and A. Shriraman, "Needle: Leveraging program analysis to analyze and extract accelerators from whole programs," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 565–576.

[21] S. Kumar, W. N. Sumner, and A. Shriraman, "Spec-ax and parsec-ax: extracting accelerator benchmarks from microprocessor benchmarks," in *2016 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2016, pp. 1–11.

[22] "Gcc, the gnu compiler collection," https://gcc.gnu.org/.

[23] T. Nowatzki, V. Govindaraju, and K. Sankaralingam, "A graph-based program representation for analyzing hardware specialization approaches," *IEEE Computer Architecture Letters*, vol. 14, no. 2, pp. 94–98, 2015.

[24] T. Nowatzki and K. Sankaralingam, "Analyzing behavior specialized acceleration," *ACM SIGPLAN Notices*, vol. 51, no. 4, pp. 697–711, 2016.

[25] C. Bai, J. Huang, X. Wei, Y. Ma, S. Li, H. Zheng, B. Yu, and Y. Xie, "Archexplorer: Microarchitecture exploration via bottleneck analysis," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 268–282.

[26] R. E. Gonzalez, "Xtensa: A configurable and extensible processor," *IEEE micro*, vol. 20, no. 2, pp. 60–70, 2000.

[27] D. Trilla, J.-D. Wellman, A. Buyuktosunoglu, and P. Bose, "Novia: A framework for discovering non-conventional inline accelerators," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 507–521.

[28] A. Abel and J. Reineke, "uops. info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 673–686.

[29] X. Hao, S. Zhang, L. Qiao, Q. Jiang, J. Shi, J. Chen, H. An, X. Tang, H. Shu, and H. Yuan, "Ciexplorer: Microarchitecture-aware exploration for tightly integrated custom instruction," in *Proceedings of the 39th ACM International Conference on Supercomputing*, 2025, pp. 975–990.

[30] G. Reinman, B. Calder, and T. Austin, "Fetch directed instruction prefetching," in *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, 1999, pp. 16–27.

[31] J. W. Fu, J. H. Patel, and B. L. Janssens, "Stride directed prefetching in scalar processors," *ACM SIGMICRO Newsletter*, vol. 23, no. 1-2, pp. 102–110, 1992.

[32] A. Seznec and P. Michaud, "A case for (partially) tagged geometric history length branch prediction," *The Journal of Instruction-Level Parallelism*, vol. 8, p. 23, 2006.

[33] E. Hussein, B. Waschneck, and C. Mayr, "Automating application-driven customization of asips: A survey," *Journal of Systems Architecture*, vol. 148, p. 103080, 2024.

[34] A. K. Verma, P. Brisk, and P. Ienne, "Rethinking custom ise identification: A new processor-agnostic method," in *Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, 2007, pp. 125–134.

[35] A. Sharifian, S. Kumar, A. Guha, and A. Shriraman, "Chainsaw: Von-neumann accelerators to leverage fused instruction chains," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–14.

Authorized licensed use limited to: University of Science & Technology of China. Downloaded on June 15,2026 at 07:27:05 UTC from IEEE Xplore. Restrictions apply.