FastISS-Enhanced RISC-V VP++: Latest Results in High-Performance Interpreter-Based Simulation*

Manfred Schlägl, Daniel Große
Institute for Complex Systems, Johannes Kepler University, Linz, Austria
manfred.schlaegl@jku.at, daniel.grosse@jku.at

Abstract

In this extended abstract, we first summarize the work presented in [1] and then report the most recent performance measurement results. In [1] we presented an enhanced, open-source, SystemC-based *RISC-V VP++* that significantly improves the performance of interpreter-based Instruction Set Simulation while preserving the approach's comprehensibility and adaptability. The enhancements include a *Dynamic Basic Block Cache* (DBBCache) and a *Load/Store Cache* (LSCache) to accelerate instruction processing and memory operations. Our latest results across Linux-based workloads demonstrate substantial performance gains of $9.97\times$ over the original *RISC-V VP++* and $1.83\times$ over the *Spike* reference simulator, with peak execution performance reaching $\sim$440 *Million Instructions per Second* (MIPS). The enhanced *Virtual Prototype* (VP) is publicly available as open-source on GitHub.

1 Motivation

*Virtual Prototype*s (VPs) are high-level, executable models of *Hardware* (HW) platforms that can run unmodified production *Software* (SW) [2, 3]. A typical major use-case of VPs is the acceleration of early design space exploration *before* physical HW is built. To ensure this acceleration is effective, VPs must be easy to create and understand. This is achieved by using high level languages like C++, domain specific standardized libraries like SystemC (IEEE 1666 [4]) [5, 6, 7], and abstraction of communication details with *Transaction Level Modeling* (TLM) [8]. For the same reason, although less efficient than dynamic binary translation methods, interpreter-based *Instruction Set Simulator*s (ISSs) are often preferred for simulating processors in VPs due to their ease of implementation, comprehensibility, and adaptability. Adding new instructions to an interpreter ISS typically involves straightforward modifications to the decoder and the instruction execution logic. However, their performance limitations often become apparent in later stages when VPs are used for interleaving HW and SW development or as reference models for verification.

The techniques presented in this work aim to significantly improve the performance of interpreter-based ISS implementations without sacrificing their comprehensibility or adaptability. Specifically, they include: (i) the ***Dynamic Basic Block Cache* (DBBCache)**, which generates an alternative representation of the executed code, the *Dynamic Basic Block Graph* (DBBG), to efficiently cache data needed for instruction processing by the ISS, and (ii) the ***Load/Store Cache* (LSCache)**, which allows direct translation of in-simulation virtual addresses to host system memory addresses, to speed up load and stores on data memory. In our evaluation, we compare these optimizations using 12 Linux-based benchmark workloads, achieving up to 442.77 *Million Instructions per Second* (MIPS) and a significant average performance increase, by a factor of 9.97 over the original *RISC-V VP++* [9] and 1.83 over the efficient RISC-V reference simulator *Spike* from *RISC-V International* [10]. We also showcase that these optimizations retain comprehensibility and adaptability of the VP by implementing the *RISC-V half-precision floating-point extension* (Zfh) in both the original and optimized VP, with no significant differences observed.

*This work has partially been supported by the LIT Secure and Correct Systems Lab funded by the State of Upper Austria.

2 Related Work

There are a number of open-source RISC-V simulators, such as *RISC-V VP++* [9], *RISC-V VP* [11], *RISC-V-TLM* [12], *Spike* [10], *QEMU* [13], *RV8* [14] or *DBT-RISE* [15]. Earlier versions of *RISC-V VP++*, its predecessor *RISC-V VP*, and *RISC-V-TLM* are all SystemC VPs with a non-optimized, interpreter-based ISS. *Spike* comes with an already highly efficient caching interpreter-based ISS, and is therefore chosen as a comparison in our evaluation. *QEMU*, *RV8* and *DBT-RISE* use dynamic binary translation. Although dynamic binary translation offers higher performance, this work focuses on interpreter-based techniques for the reasons motivated before. Commercial VPs, such as Synopsys Virtualizer, Siemens Vista and SIM-V from MachineWare are closed source, and thus exclude the qualities of *comprehensibility* and *adaptability* emphasized in this paper. To the best of our knowledge, the optimized VP resulting from this work has the highest performance interpreter-based ISS, among the SystemC-based, Linux-enabled VPs currently available as open-source.

3 Contribution

This paper considers the open-source *RISC-V VP++* [9]. The VP provides extensive capabilities such as running Linux and interactive graphical applications [16], comes with support for the *RISC-V Vector Extension* (RVV) [17] and *Capability Hardware Enhanced RISC Instructions* (CHERI) [18], and is used for advanced verification ap-

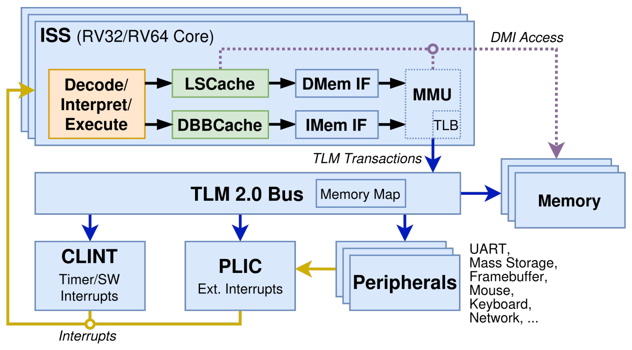

Figure 1: *RISC-V VP++* Architecture

proaches [19, 20, 21]. Its architecture is outlined in Figure 1. The blue blocks show the components included in the original *RISC-V VP++*. The added and modified components presented in this paper are highlighted in green and orange, respectively. The VP includes interpreter-based ISSs for RISC-V in 32-bit (RV32) and 64-bit (RV64) configurations, and is capable of simulating multiple cores as indicated by the stacked ISS components in Figure 1. The ISSs also include optional support for a *Memory Management Unit* (MMU) to realize *Virtual Memory Management* (VMM). A TLM-based bus links the ISSs, memory, and peripherals. The *CLINT* and *PLIC* provide timer and interrupt functionality. The VP offers configurations, from small platforms without VMM, e.g., for bare-metal SW, to larger platforms with VMM capable of running OSes such as Linux.

The first optimization integrated into the ISS of *RISC-V VP++* is the **DBBCache**, highlighted in green in Figure 1. The DBBCache is an execution-driven cache that accelerates instruction processing by dynamically identifying, storing, and reusing *Dynamic Basic Block*s (DBBs) at run time. In contrast to *Static Basic Block*s (SBBs), DBBs are formed dynamically based on the actual execution flow and *Control Flow Changes* (CFCs), such as taken branches, jumps, and trap or interrupt events. A DBB starts at the target *Program Counter* (PC) of a CFC and ends at the first subsequent instruction that causes another CFC.

During execution, the DBBCache incrementally builds a DBBG, including the DBBs and representing the observed dynamic execution paths. Each ISS instance maintains a private DBBCache, reflecting its own execution and memory context. The cache is organized in *Blocks*, each corresponding to one DBB and containing a variable number of *Entries*. An *Entry* stores the instruction word and predecoded execution information required by the ISS. *Blocks* are created for the initial entry point and whenever a CFC occurs. A hash map indexed by *Block* start PCs is maintained to locate *Blocks*, but is only consulted on cache misses.

Instruction fetch and decode are unified in the DBBCache. On a cache hit, the next *Entry* of the active *Block* is returned directly. On a miss, the instruction is fetched from memory, decompressed if necessary, decoded, and stored as a new *Entry* in the current *Block*. This allows subsequent executions to bypass repeated decode work on cache hits.

CFCs are explicitly reported by the ISS to the DBBCache. For taken branches and static jumps, each *Entry* contains a direct link to the successor *Block*. On first occurrence, the target PC is resolved via the *Block* hash map or by creating a new *Block*; the link is then stored for future executions. This enables fast switching between *Blocks* and allows multiple branch instructions within a single *Block*, increasing the average *Block* length. Dynamic jumps, including indirect calls and returns, are handled using a small per-*Block* associative cache that maps previously observed target PCs to successor *Blocks*. Trap and interrupt entries are handled similarly, using a dedicated cache to efficiently reuse the starting *Blocks* of exception handlers. Returns from traps or interrupts are not cached; instead, execution continues in a temporary dummy *Block* until the next CFC, avoiding filling the cache with *Blocks* unlikely to be reused.

To maintain coherency in the presence of self-modifying code or virtual memory changes, the DBBCache implements a lightweight coherence mechanism. RISC-V fence.i and fence.vma instructions are reported to the DBBCache and used to increment a global coherence counter. Each *Block* tracks the counter value at which it was last validated. On mismatch, the *Block* is checked against instruction memory and selectively updated if required, and affected control-flow links are reset.

Based on the high hit rates achieved by the DBBCache, further optimizations are enabled in the instruction-processing part of the ISS, highlighted in orange in Figure 1. Two particularly relevant optimizations are the use of computed gotos and a fast-path execution mode. With computed gotos, *Block Entries* store direct jump targets to the instruction implementations, eliminating dispatch overhead. The fast-path mode is used when *Blocks* are known to be coherent, minimizing control logic in the common case. Together, these optimizations significantly reduce interpreter overhead while preserving full functional correctness.

The second optimization integrated into the ISS is the **LSCache**, highlighted in green in Figure 1. While fast instruction interpretation is essential, high ISS performance also critically depends on efficient data flow to and from memory. In SystemC-based VPs with VMM, memory accesses typically involve address translation and either *Direct Memory Interface* (DMI) handling or full TLM transactions, introducing significant overhead. The LSCache aims to eliminate this execution path for as many load/store instructions as possible. It exploits the fact that in-simulation memory is commonly backed by contiguous host memory and therefore DMI-capable. For such memory regions, the LSCache directly translates in-simulation page addresses to host system memory addresses, enabling memory accesses via simple pointer dereferencing.

The cache is organized as a direct-mapped cache with 256 entries. Each entry stores a mapping from an in-simulation page start address to the corresponding host system memory page obtained via DMI. For each load or store, the cache checks whether the accessed page is present and valid. On a cache hit, the host memory address is computed from the cached host address (corresponding to the in-simulation page base) plus the page offset, completely bypassing the data memory interface, address translation, and DMI. On a miss, the access is handled via the regular data memory

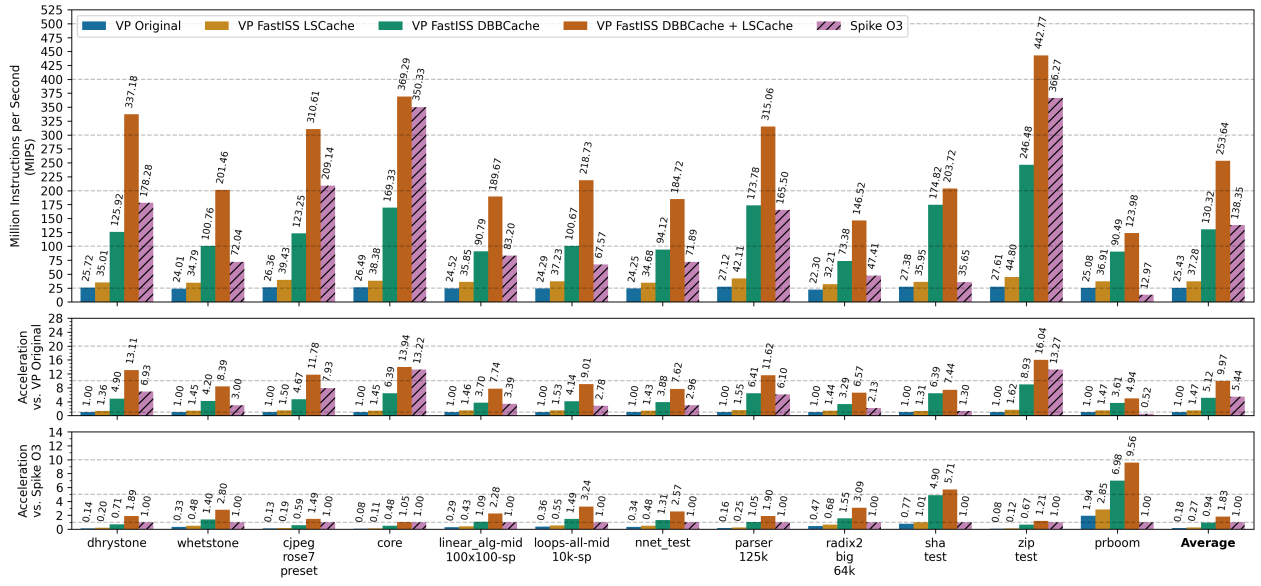
Figure 2: Results of the Workload Performance Measurements and Comparison of the Simulators

interface; if DMI is used, the resulting host page address is inserted into the cache for future accesses.

The LSCache is agnostic to the use of VMM and supports both virtual and physical in-simulation addresses. A single cache miss is sufficient to enable fast-path accesses for all subsequent loads and stores to the same 4 KiB page. To ensure correctness, changes to the virtual memory setup are reported via the RISC-V fence.vma instruction, after which all cache entries are invalidated.

4 Preserved Comprehensibility and Adaptability

To demonstrate preserved comprehensibility and adaptability, we implement support for RISC-V's Zfh extension in both the original *RISC-V VP++* (*VP Original*) and the optimized VP (*VP DBBCache + LSCache*). Both implementations change the same number of code lines, 530. 271 of these changes are related to the instruction decoding of Zfh. However, since decoding is not affected by the optimizations, there are no differences between the implementations, as expected. The remaining 259 changes are related to the interpretation and execution of Zfh. Here, we observe 68 differences, 2 for each of the 34 newly introduced instructions. These differences are related to the replacement of the case distinction by computed gotos. Specifically, the C++ case and break constructs are replaced by appropriately named macros, OP__CASE and OP__END. Overall, the original structure of the code is preserved through these macros, leading us to conclude that the presented optimizations do not have a significant negative impact on the comprehensibility or adaptability of the VP's ISS.

5 Latest Performance Results

We now discuss the latest results of performance improvements of the introduced DBBCache and LSCache compared to the original *RISC-V VP++* and the *Spike* simulator. The measurement setup is identical to that used in [1]. All measurements were performed on a host system with an Intel® Core™ i7-10700 8-core processor running at 2.9 GHz, with 128 GiB RAM. The simulated *RV64* system is based on *Linux 6.9.0* [22], created by *buildroot-2023.08.2* [23] using the *GCC* compiler in version 13 [24].

Figure 2 presents the latest results of our performance measurements and comparisons in three bar charts. The X-axis common to all charts shows the 12 selected workloads and the simulators examined. The top chart shows absolute results in MIPS obtained by taking the median of multiple measurement runs. The two charts below show the same results as acceleration factors relative to the original *RISC-V VP++* (middle) and the *Spike* simulator (bottom). As for the set of selected workloads, we use (i) *Dhrystone* [25] and *Whetstone* [26], targeting integer and floating-point, respectively, (ii) all 9 workloads from *CoreMark®-PRO* [27], which target integer, floating-point and also the memory subsystem, and (iii) a two minute demo run of *PrBoom*, a Linux port of a classic game [28], rendering 350x250 images in a in-memory framebuffer, targeting integer, but also *Operating System* (OS) interaction. As for the simulators, we compare (Figure 2, left to right) (i) the original *RISC-V VP++*, (ii) three optimized VPs with different combinations of DBBCache and LSCache enabled, and (iii) the *Spike* simulator, built with *GCC* optimization level 3.

Compared to the results reported in [1], we observe further performance improvements. The peak performance increases from 406.97 to 442.77 MIPS (*zip test* workload). Relative to the original *RISC-V VP++*, the optimized VP now achieves an average speedup of 9.97×, compared to 8.98× reported in [1]. When compared against *Spike*, the average speedup increases to 1.83×, up from 1.65×. Furthermore, while *Spike* still exceeded the execution performance of the optimized VP for the *core* workload in [1], the optimized VP now consistently outperforms *Spike* across all evaluated workloads.

These performance gains are primarily enabled by moving the tracking of executed clock cycles from instruction execution into the DBBCache. Instead of adding the cycle count of each instruction to the ISS cycle counter during execution, partial cycle sums relative to the start of a DBB are precomputed for each instruction and stored in the corresponding *Block Entries*. As a result, the ISS cycle counter only needs to be updated on *Block* transitions. The current number of executed cycles can be reconstructed at any time by combining the ISS cycle counter with the partial sum stored in the active *Entry*.

6 Conclusions

In this extended abstract, we have summarized the improved *RISC-V VP++* presented in [1] and its two most important ISS optimizations, DBBCache and LSCache. The latest performance measurements of the further optimized VP show peak execution performance of 442.77 MIPS, with average speedups of 9.97× over the original *RISC-V VP++* and 1.83× over *Spike*, consistently outperforming *Spike* across all evaluated workloads. The optimized VP is publicly available as open-source on GitHub[1].

[1]https://github.com/ics-jku/riscv-vp-plusplus

References

[1] M. Schlägl and D. Große, "Fast interpreter-based instruction set simulation for virtual prototypes," in *Design, Automation and Test in Europe Conference*, pp. 1–7, 2025.

[2] T. De Schutter, *Better Software. Faster!: Best Practices in Virtual Prototyping*. Synopsys Press, March 2014.

[3] R. Leupers, G. Martin, R. Plyaskin, A. Herkersdorf, F. Schirrmeister, T. Kogel, and M. Vaupel, "Virtual platforms: Breaking new grounds," in *Design, Automation and Test in Europe Conference*, pp. 685–690, 2012.

[4] "IEEE standard for standard SystemC language reference manual," 2023.

[5] D. Große and R. Drechsler, *Quality-Driven SystemC Design*. Springer, 2010.

[6] V. Herdt, D. Große, and R. Drechsler, *Enhanced Virtual Prototyping: Featuring RISC-V Case Studies*. Springer, 2020.

[7] M. Hassan, D. Große, and R. Drechsler, *Enhanced Virtual Prototyping for Heterogeneous Systems*. Springer, 2022.

[8] OSCI, *OSCI TLM-2.0 Language Reference Manual*, 2009.

[9] M. Schlägl, C. Hazott, and D. Große, "RISC-V VP++: Next generation open-source virtual prototype," in *Workshop on Open-Source Design Automation*, 2024.

[10] "Spike RISC-V ISA simulator." https://github.com/riscv/riscv-isa-sim, 2026.

[11] V. Herdt, D. Große, H. M. Le, and R. Drechsler, "Extensible and configurable RISC-V based virtual prototype," in *Forum on Specification and Design Languages*, pp. 5–16, 2018.

[12] M. Montón, "A RISC-V SystemC-TLM simulator," 2020.

[13] "QEMU a generic and open source machine emulator and virtualizer." https://www.qemu.org, 2026.

[14] "RV8." https://github.com/larkmjc/rv8, 2026.

[15] "DBT-RISE." https://github.com/Minres/DBT-RISE-Core, 2026.

[16] M. Schlägl and D. Große, "GUI-VP Kit: A RISC-V VP meets Linux graphics - enabling interactive graphical application development," in *ACM Great Lakes Symposium on VLSI*, pp. 599–605, 2023.

[17] M. Schlägl, M. Stockinger, and D. Große, "A RISC-V "V" VP: Unlocking vector processing for evaluation at the system level," in *Design, Automation and Test in Europe Conference*, pp. 1–6, 2024.

[18] M. Schlägl, A. Hinterdorfer, and D. Große, "A RISC-V CHERI VP: Enabling system-level evaluation of the capability-based CHERI architecture," in *Asia and South Pacific Design Automation Conference*, 2026.

[19] C. Hazott and D. Große, "Relation coverage: A new paradigm for hardware/software testing," in *IEEE European Test Symposium*, pp. 1–4, 2024.

[20] M. Schlägl and D. Große, "Single instruction isolation for RISC-V vector test failures," in *IEEE/ACM International Conference on Computer-Aided Design*, pp. 156:1–156:9, 2024.

[21] C. Hazott, F. Stögmüller, and D. Große, "Using virtual prototypes and metamorphic testing to verify the hardware/software-stack of embedded graphics libraries," *Integr.*, vol. 101, 2025.

[22] "The Linux kernel archives." https://kernel.org, 2026.

[23] "Buildroot." https://www.buildroot.org, 2026.

[24] "GCC the GNU compiler collection." https://gcc.gnu.org, 2026.

[25] "Dhrystone benchmark version 2.1." https://www.netlib.org/benchmark/dhry-c, 2026.

[26] "C converted Whetstone double precision benchmark version 1.2." https://www.netlib.org/benchmark/whetstone.c, 2026.

[27] "The EEMBC CoreMark-PRO processor benchmark." https://www.eembc.org/coremark-pro, 2026.

[28] "PrBoom." https://prboom.sourceforge.net/, 2026.