A RISC-V CHERI VP: Enabling System-Level Evaluation of the Capability-Based CHERI Architecture

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

Andreas Hinterdorfer
Institute for Complex Systems
Johannes Kepler University
Linz, Austria
ahinterd@gmail.com

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

***Abstract*—Despite decades of mitigation efforts, memory corruption bugs remain a dominant source of security vulnerabilities. CHERI, a capability-based architecture, directly targets this problem by replacing traditional pointers with Capabilities that encode bounds, permissions, and tamper-protection tags. However, CHERI represents a significant architectural intervention that impacts not only the processors core, but the entire Hardware/Software platform. System-level evaluation methods, such as *Virtual Prototype*s (VPs), have shown to be highly valuable for exploring, validating, and optimizing such complex Hardware/Software systems.**

This paper introduces the first open-source SystemC/TLM-based CHERI-enhanced RISC-V VP. The VP comes with support for *Virtual Memory Management* (VMM) and is capable of executing complex software stacks, such as the general purpose and memory-safe CheriBSD operating system. A verification using TestRIG demonstrates the VP's robustness, passing 2.15 million test cases. A case study with CheriBSD and 10 representative, demanding benchmark workloads highlights the VP's capability to simulate complex CHERI-enabled systems and to provide valuable insights for Hardware/Software co-design.

The CHERI-enhanced VP, along with the Software used in our case study is available as open-source on GitHub.

I. Introduction

Memory corruption bugs are one of the oldest problems in computer security and are still a major issue today. According to the MITRE ranking, memory corruption bugs are considered one of the top three most dangerous *Software* (SW) vulnerabilities in 2024 [1]. Also, the Chromium project reports that around 70% of their security bugs are memory safety problems [2]. Although modern type-safe programming languages, like Rust, attempt to address this problem, it is economically unrealistic to expect that the billions of lines of existing C and C++ code can be replaced in the short or mid-term future. To address these persistent challenges, alternative approaches are required that enhance the security of existing memory-unsafe languages like C and C++ without necessitating their wholesale replacement.

One such approach is *Capability Hardware Enhanced RISC Instructions* (CHERI), a project started in 2010 by the University of Cambridge as part of the *Clean Slate Trustworthy Secure Research and Development* (CTSRD) initiative [3]. CHERI is a HW/SW-semantics co-design project that extends conventional *Instruction Set Architecture*s (ISAs), compiler and *Operating System*s (OSs) with new architectural features to enable fine-grained memory protection and highly scalable SW compartmentalization [4]. A central concept of CHERI is the replacement of conventional pointers by so called *Capabilities*. While conventional pointers contain only the address they point to, *Capabilities*, beside this, also encode access permissions, address bounds and a *Tag* flag as protection against tampering. Each time a *Capability* is used to access memory (instruction fetch, load, or store), its properties are checked, and if the check fails, the access is rejected. On the SW side, *Capabilities* are automatically generated and handled by a CHERI-aware compiler. Explicit source code modifications are only necessary in exceptional cases, like in low-level system SW. One example of a practical use-case for *Capabilities* is the protection against out-of-bound arrays accesses in C. *Capabilities* pointing to an array contain the bounds of the array, which are checked by *Hardware* (HW) on each access, thus eliminating out-of-bound accesses – a common source of buffer overflow attacks in C programs. However, while *Capabilities* serve as the foundational building blocks, many other concepts of CHERI are built on top of them and also involve significant complexity.

Recently, the primary reference platform for CHERI was shifted from the MIPS architecture to RISC-V [5], [6], a relatively young, highly modular and open-standard ISA. RISC-V specifies very compact and simple base ISAs (RV32I for 32-bit and RV64I for 64-bit Integer) that can be extended by a variety of optional standard extensions, for example Multiplication/Division (M), Floating Point (F, D), Vector (V), only to name a few. This scalability allows RISC-V to be utilized across a wide range of applications, from compact controllers to high-performance application processors. Its openness and modularity make RISC-V a particularly suitable foundation for novel approaches like CHERI.

However, unlike other RISC-V extensions, CHERI-RISC-V represents a significant architectural intervention that impacts not only the processor core but the entire HW/SW plat-

form. For instance, in the processor core, CHERI-RISC-V not only introduces instructions with new functionality but also modifies the behavior of existing instructions and the overall instruction processing. On the HW platform side, components such as caches, buses, and memory require adjustments to support the additional *Tag* flag. On the SW side, compilers, OSs, and low-level systems SW demand substantial attention.

One approach to tackle this complexity is the use of *Virtual Prototype*s (VPs), as we will demonstrate in this paper. VPs are executable system-level models of entire HW platforms which can run unmodified production SW [7] and enable early exploration, validation, and optimization of complex HW and SW systems [8]. These models are industry-proven and are typically implemented in SystemC/TLM, a standardized class library for C++ (IEEE 1666) [9]–[11]. In this paper, we consider the open-source, SystemC/TLM-based *RISC-V VP++* [12], which was selected for its extensive feature set [13]–[20]. The VP supports RISC-V RV32 and RV64 in both single- and multicore configurations, with optional support for *Virtual Memory Management* (VMM). Additionally, *RISC-V VP++* offers multiple ready-to-use HW platform configurations, ranging from small microcontroller-based systems for bare-metal SW to application processor-based systems capable of running complex OSs like Linux [21].

Contributions:

1) We introduce the first open-source SystemC/TLM-based VP with VMM support, fully implementing CHERI-RISC-V as defined in version 9 of the CTSRD CHERI specification [22]. The implementation of the CHERI-enhanced VP is the result of a thorough analysis of the 500+ pages specification, the formal CHERI RISC-V Sail model [23], and the operation of *RISC-V VP++*.
2) We discuss the process used for verification of the CHERI-enhanced VP using TestRIG with *Direct Instruction Injection* (DII) [24] a framework for automated testing of RISC-V implementations using *Random Instruction Generation* (RIG) and the *RISC-V Formal Interface* (RVFI) standard [25]. Based on this, we demonstrate the robustness of the VP with 2.15 million passed test cases.
3) In our case studies, we demonstrate the potential of the CHERI-enhanced VP through practical evaluations. Using CheriBSD [26], we show that complex OSs with VMM can be successfully executed on the VP. Based on this, to highlight the value for system-level evaluation, we present results of detailed measurements from selected benchmark workloads.

We like to mention, that there are currently significant efforts underway to industrialize CHERI and create an officially ratified RISC-V CHERI extension [27]. However, this process is still ongoing. In this work, we focus on the stable CTSRD CHERI specification in version 9.

The CHERI-enhanced VP, along with the SW used in our case study is available on GitHub[1].

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

II. Related Work

Our work addresses the limitations of current approaches for CHERI-RISC-V system evaluation. *Register Transfer Level* (RTL) models, while precise, are too slow and resource-intensive for efficient system-level SW development and exploration.

Machine emulators like *QEMU* [28], [29] offer much higher performance but lack the accuracy required for CHERI and are not standardized (e.g., no SystemC/TLM support), limiting their integration into industrial workflows. Similarly, while gem5 is a widely used and flexible architecture simulator with RISC-V support [30], no CHERI-RISC-V extension is available for it, preventing its direct use for CHERI system-level evaluation.

Formal models such as Sail [23], [31] offer precise semantics but are not intended for system-level modeling and cannot run complete software stacks or interact with realistic peripheral models.

To the best of our knowledge, no commercially available tool provides an open, accurate, and extensible SystemC/TLM-based CHERI VP that supports VMM and is capable of running full operating systems like CheriBSD.

With this work, we close that gap by introducing the first open-source CHERI-enhanced RISC-V VP that combines system-level performance and accuracy with industrially relevant standards.

III. Central Concepts of CHERI

A core concept of CHERI is replacing traditional pointers with *Capabilities*. Unlike regular pointers, which store only an integer address, *Capabilities* include a *Tag* bit for tamper protection, access permissions, and address bounds, along with additional metadata like flags and object type. To minimize overhead, bounds are compressed using a floating-point-like technique, enabling *Capabilities* on CHERI-RISC-V RV64 to be just 128-bits wide – only twice the size of a regular pointer. Unlike other properties of a *Capability*, the *Tag* bit, responsible for tamper protection, is handled independently from the directly accessible portion of the *Capability*. Supporting *Tag* bits requires platform-wide modifications, including caches, buses, and memory.

Each time, a *Capability* is used to access the memory (fetch, load, store), the operation has to be *authorized* by the HW. The HW checks if (i) the *Capability* is valid (*Tag* bit is set), (ii) the access is permitted according to the *Capability* access permissions, and (iii) the address is within the *Capability* bounds. If one of these checks fail, the memory access is rejected by the HW. This process is used not only for data accesses but also for protecting the control flow. For this, CHERI extends the *Program Counter* (PC) to a *Program Counter Capability* (PCC), which is used to authorize all instructions fetches.

To illustrate the protection provided by *Capabilities*, Listing 1 shows a very simple C program containing a not uncommon buffer overflow vulnerability. The program allocates an array `str` with room for 10 characters on the stack and passes

```
1  void main(void) {
2    char str[10];           /* allocate str (10 characters) on the stack */
3    gets(str);              /* read characters from stdin to str */
4    printf("%s\n", str);    /* output the read characters */
5  }
```

Listing 1: Simple C program containing a buffer overflow vulnerability

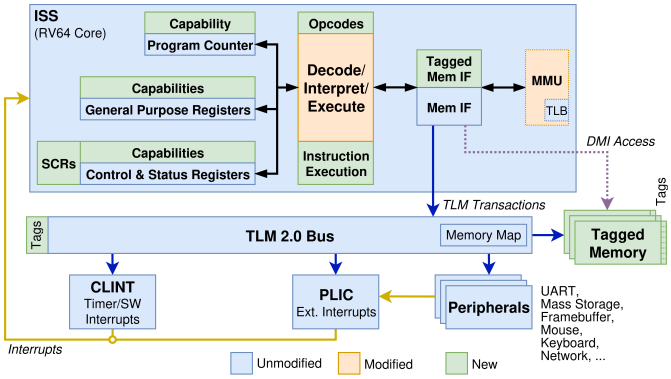

Fig. 1: Architectural Overview of the CHERI-enhanced VP

the array as argument to the function `gets`. The function `gets` reads an arbitrary number characters from the standard input until a line end is detected and stores the read characters as elements in the given `str` array. On a typical architecture, `str` is a regular pointer. Feeding the program with more than 10 characters, leads to a buffer overflow of `str`, which can be exploited by attackers to modify the programs memory space. In contrast, on a *Capability*-based architecture like CHERI, as soon as `gets` attempts to store more than 10 characters in `str`, the HW detects the out-of-bounds access and triggers a trap, allowing the OS to terminate the process. The buffer overflow, and with this a potential attack is successfully prevented.

Notably, no modifications to the program code are required to gain these protections – compiling with a CHERI-aware compiler is sufficient.

While CHERI-RISC-V significantly modifies the ISA, it remains capable of executing standard non-CHERI-aware RISC-V software, referred to as *Integer Pointer Mode* in this paper. Mode switching is managed via the *F* flag in the PCC: when set, execution is in *Capability Mode*; when cleared, it is in *Integer Pointer Mode*.

IV. Enhancing RISC-V VP++ with CHERI

The architecture of the CHERI-enhanced VP is outlined Fig. 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 implements an interpreter-based *Instruction Set Simulator* (ISS) for RV64 and also include optional support for a *Memory Management Unit* (MMU) to realize VMM. A *Transaction Level Modeling* (TLM)-based bus links the ISSs, memory, and peripherals. The *CLINT* and *PLIC* provide timer and interrupt functionality. Memory can be accessed either via TLM transactions, like other peripherals, or via *Direct Memory Interface* (DMI), where accesses are realized directly via pointers provided by SystemC. The former allows tracing of memory accesses on the TLM bus, while the latter provides higher simulation performance.

In the following sections, we provide a structured overview of the integration process of CHERI-RISC-V into *RISC-V VP++*, focusing on the components highlighted in green and orange in Fig. 1. For the integration process, we primarily relied on the CHERI-RISC-V specification [22]. Any ambiguities or missing details were resolved using the formal CHERI RISC-V Sail model [23].

A. Capabilities

The first step in the integration process is the introduction of *Capabilities*, which are implemented according to the CHERI-RISC-V specification, including respective data structures, compression/decompression functions and various helper methods. After that, all entities that can hold an address (i.e. a pointer) are extended to *Capabilities*. This modifications affects (i) the PC, (ii) all 32 general purpose registers, and (iii) all *Control and Status Register*s (CSRs) that may hold an address. The affected components are highlighted by the green blocks on the left side of the ISS, shown in Fig. 1.

B. Special Capability Registers (SCRs)

As mentioned in Section III, CHERI-RISC-V continues to support execution in *Integer Pointer Mode*. To distinguish between CSRs in *Integer Pointer Mode* and their *Capability*-extended variants in *Capability Mode*, the CSRs are referred to as *Special Capability Register*s (SCRs) in *Capability Mode*. For example, the *MTVEC* and *STVEC* (*Trap-Vector Base-Address Register*) CSRs, which hold the addresses for trap handling in machine and supervisor-mode, are extended to corresponding *MTCC* and *STCC* (*Trap Code Capability*) SCRs.

Furthermore, CHERI-RISC-V defines some special registers, that are only available as SCRs. These include for example the previously mentioned *Capability* extended PC, the PCC, and a *Capability* used for non-capability-aware load and store operations, the *Default Data Capability* (DDC). The SCRs is shown as green block left to the CSRs block in Fig. 1. Overall, we extend 9 CSRs to SCRs and add 5 new SCRs, including the PCC.

C. Instructions Processing (Decode/Interpret/Execute)

In this section, we examine the instruction processing, represented by the orange block *Decode/Interpret/Execute* and the green blocks *Opcodes* and *Instruction Execution* in Fig. 1. The instruction processing flow is presented in Fig. 2. Similar, as in Fig. 1, blue blocks show the components included in the original *RISC-V VP++*, and the added and modified components are highlighted in green and orange, respectively.

The instruction processing in *RISC-V VP++* is structured in four stages. First, a new instruction is **fetched** (*a*) from the memory location pointed to by PC position. The read instruction word is then **decoded** (*b*) using a decision tree in which individual bits and bit fields of the instruction (the encoding) are examined successively until an operation is identified. As result, we get a unique *opId* from the *opId Table*

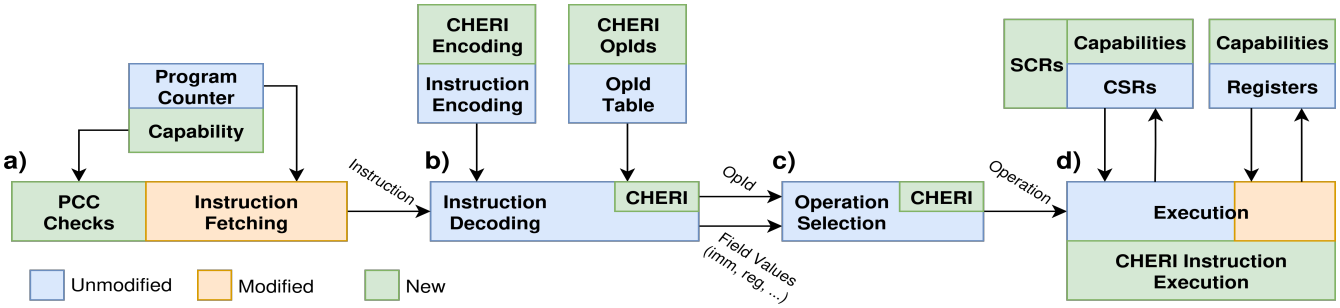

Fig. 2: Instruction Processing in the CHERI-enhanced VP

and the field values of the instructions (e.g. used registers, immediate values, ...). Next, the to be executed operation implementation is **selected** (*c*) using a case distinction on the *opId*. The relevant fields are selected and passed to the concrete operation implementation. Finally, the concrete operation implementation is **executed** (*d*), applying all associated side effects, i.e., updates to registers, CSRs, memory and peripherals. We now proceed to discuss the required modifications to each of the four stages to enable the integration of CHERI-RISC-V.

a) Instruction Fetching: To integrate CHERI into the instruction fetch process, several modifications are required to enforce its *Capability*-based security model. Similar to loads and stores, fetches must also be authorized by a *Capability* – in this case, by the *Capability*-extended PCC introduced in Section IV-A. However, since both CHERI-RISC-V and RISC-V support compressed 16-bit instructions, authorizations must be performed on a 16-bit basis. Consequently, the fetch process is divided in up to two stages. First, a *Capability*-authorized fetch is performed for a 16-bit half-word using the PCC. If this half-word is identified as a compressed instruction, it is passed directly to decoding. Otherwise, an additional PCC-authorized fetch is performed for the next higher 16-bit half-word. The two half-words are then combined to form a full 32-bit instruction word, which is subsequently passed to decoding.

b) Instruction Decoding: To support CHERI, the instruction decoder must be extended to handle the new instructions introduced by the CHERI-RISC-V. In total, 99 new non-compressed instructions are added. To support the new instructions in the decoder, we add two new components: (i) the *CHERI Encoding*, which extends the existing *Instruction Encoding*, and (ii) the *CHERI* opIds, which extends the *opId Table*. Implementation efficiency is enhanced, and the risk of errors is minimized by leveraging automated code generation for both extensions. All 99 CHERI instruction use existing RISC-V encoding formats (R-, I-type or S-type), and most instructions also share a common opcode and are further distinguished by additional fields in the instruction format. This structure allows for an efficient extension of the existing decoder decision tree by adding new *opId*-cases for the CHERI instructions. In addition to the 99 non-compressed instructions, CHERI-RISC-V modifies the behavior of 21 compressed instructions. When decoded in *Integer Pointer Mode*, these instructions retain their original behavior, but when decoded in *Capability Mode* (as determined by the PCC mode flag), they perform CHERI-specific operations. This functionality is implemented by introducing additional mode checks during the decompression of compressed instructions in the decoder.

c) Operation Selection: As previously described, this stage receives the *opId* and all field values of the instruction. The operation selection is implemented as a case distinction on the *opId* (C++ switch case). For each case, the relevant field values are selected, and the execution of the operation is delegated to the execution stage. To support CHERI-RISC-V, the case distinction is extended with the new CHERI *opIds*.

d) Execution: In this final stage, the actual execution of the selected operation (*opId*) takes place. To support CHERI, this block is extended with implementations for the 99 new CHERI-RISC-V instructions. Each implementation is based on the CHERI specification, with formal CHERI RISC-V Sail model serving as a reference for cases where the specification is ambiguous. In addition to the 99 new instructions, indicated by the green CHERI *Instruction Execution* block in Fig. 2, CHERI also modifies the behavior of several existing instructions. This is indicated by the partially orange section in the *Execution* block in Fig. 2. All integer load and store instructions are now mode-dependent, using the passed on *Capability* in *Capability Mode*, or the DDC in *Integer Pointer Mode*. Control-flow instructions, such as branches and jumps, are also affected. In *Capability Mode*, the target address must be authorized by the PCC and passed on *Capabilities*.

This concludes our discussion of orange block *Decode/Interpret/Execute* and the green blocks *Opcodes* and *Instruction Execution* shown in Fig. 1.

D. Memory Interface and Memory Management Unit (MMU)

Major parts of the logic for RISC-V load/store instructions are actually implemented in the *Memory Interface* (Mem IF) of the VP, represented as the blue block on right left side of the ISS in Fig. 1. For CHERI-RISC-V, we extend the Mem IF with the new component Tagged Mem IF, shown as green block above the Mem IF. This new component introduces methods to support the newly added CHERI-RISC-V *Capability* load/store instructions. Additionally, since CHERI modifies the behavior of existing load/store instructions, the register used as the address in the standard RISC-V ISA must now be interpreted as a *Capability*. This *Capability* is used to authorize the operation, requiring several checks to be performed before each load/store operation.

Also the optional MMU, shown as orange block near the memory interfaces, needs some modifications: (i) The MMU is extended to support CHERI *Tags*, and (ii) the *Page Table Entries* (PTEs) are extended by 5 new CHERI-RISC-V related flags, to control the behavior of *Capability* stores (two flags) and *Capability* loads (three flags) on a per-page basis.

E. Tagged Memory and Tags in SystemC/TLM Transactions

As already highlighted in the Section I, CHERI-RISC-V not only affects the ISA of a processor, but has also effects on other parts of the HW platform. In CHERI, each memory location capable of holding a *Capability* is associated with a 1-bit *Tag* that tracks the validity of the *Capability*. This requires a so called *tagged memory architecture* where the *Tag* bit is atomically bound to each *Capability*-aligned memory word, ensuring that non-*Capability* operations, such as byte-level stores, clear the *Tag*. To support this in our VP, we implement a new Tagged Memory Module, which replaces the memory module of the original VP and is shown as green block on the left side of Fig. 1. The module tracks the *Tags* using an array of Booleans, indexed by access addresses divided by the size of a *Capability*.

As shown in the center of Fig. 1, the VP's ISS, memory and peripherals are connected as modules via a SystemC/TLM bus. In TLM, bus communication is abstracted via transactions, which are realized as function calls carrying a standardized transaction object. This transaction object contains all information related to the transaction, e.g., the type (read or write), the target address and the payload data. However, a TLM transaction object has no representation of a CHERI *Tag*. To address this, we utilize the TLM extension mechanism defined in the SystemC standard [9], which allows user-defined objects to be attached to transaction objects. Based on this mechanism, we implement a custom TLM extension to carry the *Tag*, and extend the Tagged Mem IF and the Tagged Memory Module. With this, these CHERI-aware modules are now able to exchange *Tags* in addition to data over the unmodified TLM bus. Meanwhile, modules that are unaware of the *Tag*-extension simply ignore it and continue to function as before, ensuring compatibility with existing SystemC/TLM components.

V. Verification with TestRIG

In this section, we give a brief overview on the verification process used for our CHERI-enhanced VP including some major results metrics.

The complexity of CHERI-RISC-V makes manual testing infeasible, necessitating automated tools for thorough verification. For our verification we use TestRIG, which was introduced together with the DII technique in [24]. TestRIG is a framework for automated testing of RISC-V implementations using *Random Instruction Generation* (RIG) and the RVFI standard [25]. Orchestrated by the so called *Verification Engine* (VEngine), generated instruction sequences are fed directly into connected cores via DII, execution result traces are extracted using the RVFI, and then, the results are compared for differences.

To be able to use TestRIG in our verification process, the CHERI-enhanced VP is further extended with support for DII and RVFI. The VP is then integrated with a simulator generated from the formal CHERI RISC-V Sail model, serving as a reference platform in a CHERI TestRIG setup. This setup was used iteratively throughout the development process of our CHERI-enhanced VP, proving to be highly valuable by isolating numerous bugs and effectively demonstrating the practical use of TestRIG. Some particularly challenging and interesting findings included the detection of missing alignment checks in specific execution paths and the identification of incorrect handling of integer values in *Capability* registers. In final tests, the presented state of our CHERI-enhanced VP successfully **passes 2.15 million test cases** in approximately **30 hours**, executing a total number of **~1.85 billion instructions**.

VI. Case Study: CheriBSD with Benchmarks

In this case study, we demonstrate the potential of the CHERI-enhanced VP through practical evaluations with CheriBSD, a CHERI-enabled variant of FreeBSD – a mature, full-featured, and widely-deployed operating system. We show, that the CHERI-enhanced VP is able to run complex CHERI-enabled general purpose OSs with VMM and based-on applications. Furthermore, we illustrate that we can derive valuable assessments for the design of CHERI-based HW/SW systems, by comparing the execution of CheriBSD and 10 selected benchmark workloads compiled without and with support for CHERI on our VP.

To ensure comparability, the toolchains for both, the non-CHERI and CHERI builds are created using cheribuild [32] which is provided by the CTSRD project. These toolchains are then used to build CheriBSD, as well as the necessary CHERI-enabled low-level bootloader OpenSBI-CHERI [33], both with and without CHERI support. At this point, we can boot the CHERI enabled CheriBSD on our VP. On an AMD® Ryzen 7 PRO 6850U 8-core processor running at 2.7 GHz, the system boot – measured from the start of the simulation to the point where the init process of CheriBSD was loaded from a root file system on a RAM disk and starts its execution – takes approximately 18 seconds. Examining the detailed runtime statistics provided by the VP, we observe that by this point, the VP has executed approximately 324 million instructions and performed around 21 million data loads and 63 million data stores. Additionally, the Tagged Memory Module contains 655k valid *Tags* at this stage, corresponding to valid *Capabilities*. The system continues its boot process and eventually provides a console. From this point onward, the user can interact with the system (e.g. run programs) in near real-time.

Next, we analyze the behavior of a selected set of benchmark workloads, each compiled with and without support for CHERI. To account for any optimizations possibly not yet ported to the CHERI LLVM compiler [34], and with this improve comparability, we disable all compiler optimizations (i.e. O0). The workloads are each executed on a CheriBSD system without and with CHERI support, running on the exact

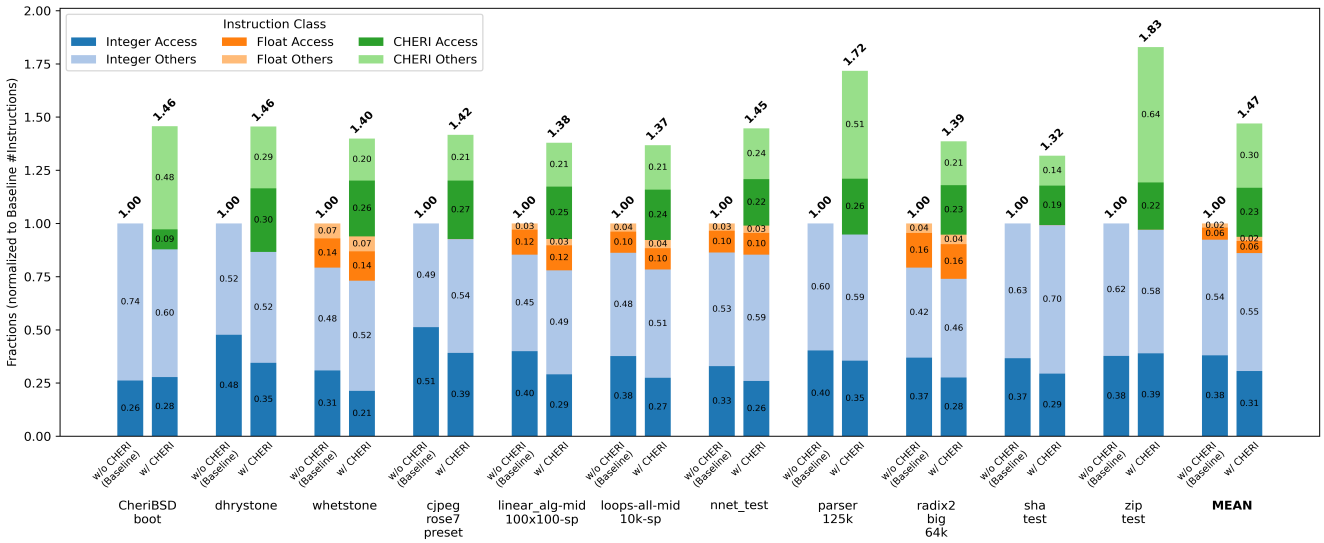

Fig. 3: Distribution of executed Instructions for Workloads with and without CHERI (normalized to their respective baselines)

same CHERI-enhanced VP model. We again leverage the runtime statistics provided by the VP, this time to examine the distribution of executed instructions for workloads with and without CHERI. All runs are repeated three times, and the median of the measurements (based on the number of executed instructions) is selected as the final outcome. The results are visualized in Fig. 3. The X-axis shows the workloads along with the mean value across all workloads, divided into two setups: *without CHERI* as the baseline, and *with CHERI*. The selected workload set includes (a) the system boot process of CheriBSD itself (as described above), (b) the well-known *Dhrystone* [35] and *Whetstone* [36] benchmarks, and (c) 8 workloads (excluding *core*) from *CoreMark®-PRO* [37]. All executed instructions are classified in (i) all integer related instructions including privileged instructions (*Integer* in blue), (ii) all floating-point related instructions (*Float* in orange), and (iii) all newly introduced CHERI instructions (*CHERI* in green). All instruction classes are further divided in load/store/atomic (*Access* in dark) and all other instructions (*Others* in light). All shown values for a workload are normalized to the respective number of executed instruction in the baseline.

As expected, the number of *Float* instructions remains very consistent between CHERI and non-CHERI executions. For *Integer Access* instructions, a decrease is observed in most cases, except for the *CheriBSD boot* and *zip test* workloads. This reduction, however, is clearly outweighed by the additional number of *CHERI Access* instructions. The *Integer Others* category remains roughly the same, though a slight increase is noticeable on average (*MEAN*). Overall, CHERI introduces significant overhead, with an average increase of 1.47 across all workloads, reaching as high as 1.83 for *zip test*. It is also worth noting, that *CoreMark®-PRO* actually consists of 9 workloads. However, the missing workload *core*, when built for CHERI-RISC-V, encounters a *Tag* violation during execution, which is also reproducible on the QEMU-CHERI emulator [29]. This suggests that *core* may either be one of the cases requiring a manual port to CHERI or that its implementation contains a latent bug exposed only by CHERI – an interesting case that will be investigated in future work.

Overall, the presented results indicate that, although CHERI offloads significant aspects of memory protection to HW, it also introduces notable overhead in SW. Further investigations and combined efforts in improving both HW and SW are necessary to address these challenges. This clearly highlights the value of system-level approaches capable of simulating complex CHERI-based HW/SW systems, such as our CHERI-enhanced VP.

VII. Conclusions

In this paper, we introduced the first open-source CHERI-RISC-V enhanced SystemC/TLM-based VP, enabling system-level evaluation of CHERI-RISC-V-based HW/SW systems. Our verification with TestRIG validated the VP's robustness, by passing **2.15 million test cases**. Our case study with CheriBSD and 10 well-established benchmark workloads demonstrated the VP's ability to simulate complex CHERI-based systems. Notably, running CheriBSD exercises the complete CHERI trust and protection chain, demonstrating the correct interaction of instruction set features, memory management, privilege control, and capability enforcement in the VP. The VP measurements reveal an average instruction overhead of **1.47** for CHERI-enabled workloads. These results underscore the importance of system-level approaches, like our CHERI-enhanced VP, in effectively simulating and analyzing complex CHERI-based HW/SW systems.

For future work, we plan to enhance the verification of CHERI-based implementations by adapting *RVVTS* [38], a test framework that has proven successful for testing the complex RISC-V vector extension.

The CHERI-enhanced VP, along with the Software used in our case study is available on GitHub.

Acknowledgments

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

REFERENCES

[1] MITRE, "Cwe/sans top 25 most dangerous software errors," 2024. [Online]. Available: https://cwe.mitre.org/top25/

[2] The chromium projects - memory safety. [Online]. Available: https://www.chromium.org/Home/chromium-security/memory-safety/

[3] Ctsrd - rethinking the hardware-software interface for security. [Online]. Available: https://www.cl.cam.ac.uk/research/security/ctsrd/

[4] R. N. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera, "Cheri: A hybrid capability-system architecture for scalable software compartmentalization," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 20–37.

[5] A. Waterman and K. Asanović, *The RISC-V Instruction Set Manual; Volume I: Unprivileged ISA*, SiFive Inc. and CS Division, EECS Department, University of California, Berkeley, 2019.

[6] ——, *The RISC-V Instruction Set Manual; Volume II: Privileged Architecture*, SiFive Inc. and CS Division, EECS Department, University of California, Berkeley, 2019.

[7] 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*, 2012, pp. 685–690.

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

[9] "IEEE 1666-2023 standard for standard SystemC language reference manual." [Online]. Available: https://doi.org/10.1109/IEEESTD.2023.10246125

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

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

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

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

[14] M. Schlägl, J. Reichhardt, and D. Große, "ProtoLens: dynamic transaction visualization in virtual prototypes," in *Forum on Specification and Design Languages*, 2025, pp. 1–8.

[15] C. Hazott and D. Große, "Boosting SW development efficiency with function lifetime diagrams," in *IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems*, 2025, pp. 99–104.

[16] 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.

[17] C. Hazott and D. Große, "LLM-assisted metamorphic testing of embedded graphics libraries," in *Forum on Specification and Design Languages*, 2025, pp. 1–10.

[18] 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*, 2024, pp. 1–6.

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

[20] ——, "DSA monitoring framework for HW/SW partitioning of application kernels leveraging VPs," in *IEEE Design and Verification Conference and Exhibition Europe*, 2023, pp. 34–41.

[21] 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*, 2023, pp. 599–605.

[22] R. N. M. Watson, P. G. Neumann, J. Woodruff, M. Roe, H. Almatary, J. Anderson, J. Baldwin, G. Barnes, D. Chisnall, J. Clarke, B. Davis, L. Eisen, N. W. Filardo, F. A. Fuchs, R. Grisenthwaite, A. Joannou, B. Laurie, A. T. Markettos, S. W. Moore, S. J. Murdoch, K. Nienhuis, R. Norton, A. Richardson, P. Rugg, P. Sewell, S. Son, and H. Xia, "Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 9)," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-987, Sep. 2023. [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-987.pdf

[23] "CHERI RISC-V Sail model," https://github.com/CTSRD-CHERI/sail-cheri-riscv.

[24] A. Joannou, P. Rugg, J. Woodruff, F. A. Fuchs, M. van der Maas, M. Naylor, M. Roe, R. N. M. Watson, P. G. Neumann, and S. W. Moore, "Randomized testing of RISC-V CPUs using direct instruction injection," *IEEE Design and Test*, vol. 41, no. 1, pp. 40–49, 2024.

[25] "RISC-V formal verification framework," https://github.com/YosysHQ/riscv-formal, 2025.

[26] "CheriBSD - capability enabled, unix-like operating system that extends freebsd to take advantage of capability hardware," https://www.cheribsd.org, 2025.

[27] Cheri alliance. [Online]. Available: https://cheri-alliance.org/

[28] "QEMU a generic and open source machine emulator and virtualizer," https://www.qemu.org, 2025.

[29] "QEMU-CHERI," https://github.com/CTSRD-CHERI/qemu, 2025.

[30] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[31] "Riscv sail model," https://github.com/riscv/sail-riscv.

[32] "cheribuild.py - a script to build cheri-related software," https://github.com/CTSRD-CHERI/cheribuild, 2025.

[33] "Opensbi-cheri," https://github.com/CTSRD-CHERI/opensbi, 2025.

[34] "LLVM-CHERI," https://github.com/CTSRD-CHERI/llvm, 2025.

[35] "Dhrystone benchmark version 2.1," https://www.netlib.org/benchmark/dhry-c, 2025.

[36] "C converted Whetstone double precision benchmark version 1.2," https://www.netlib.org/benchmark/whetstone.c, 2025.

[37] "The EEMBC CoreMark-PRO processor benchmark," https://www.eembc.org/coremark-pro, 2025.

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