ProtoLens: Dynamic Transaction Visualization in Virtual Prototypes

Tool Paper

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

Jonas Reichhardt
Institute for Complex Systems
Johannes Kepler University
Linz, Austria
jonas.reichhardt@jku.at

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

***Abstract*—Transaction-level debugging in *Virtual Prototype*s (VPs) remains challenging due to the sheer number and intricate nature of interactions between software and hardware components. This paper presents ProtoLens, the first open-source tool for dynamic visualization of *Transaction Level Modeling* (TLM) transactions in SystemC-based VPs. Integrated with the open-source RISC-V VP++, ProtoLens provides an interactive web front-end that displays architecture-aware transaction flows in real-time. It captures transaction data via a lightweight extension of the TLM bus and enriches it with peripheral-specific views through user-defined modules, so-called *Transaction View Module*s (TVMs). Additionally, ProtoLens supports integration with software debuggers, allowing synchronized transaction inspection and control of the simulation flow. This enables developers to efficiently analyze issues such as incorrect memory mappings, unexpected peripheral behavior, and to better understand the overall system architecture.**

Two case studies highlight the capabilities of ProtoLens: one demonstrates how it complements classical debugging in a bare-metal software example, and the other showcases its ability to reconstruct real-time graphics output from a Linux-based game.

I. Introduction

*Virtual Prototype*s (VPs) are abstract, executable models of embedded systems used to support early *Software* (SW) development, system validation, and architectural exploration before the underlying *Hardware* (HW) – starting from RTL – is developed or available. To enable these use cases effectively, VPs offer significantly faster simulation speed compared to cycle-accurate models, allowing developers to run complex SW scenarios and system tests efficiently [1]. To realize VPs, industry commonly employs virtual platforms – system-level models constructed using SystemC [2]–[6] and *Transaction Level Modeling* (TLM) [7] that accurately represent HW components and their interactions. To create a virtual platform, a set of key components must be modeled and integrated at a high level of abstraction. This includes developing SystemC/TLM models for processing elements (e.g. CPUs, accelerators), memories (volatile/non-volatile), peripheral devices, and interconnects such as buses. Each component must expose standardized TLM interfaces to enable transaction-based communication, allowing fast and accurate simulation of SW. However, there are two major challenges: First, SW execution may trigger a series of HW events – such as memory accesses, peripheral operations, or interrupts – that are difficult to follow and analyze without a clear representation of the underlying transactions. As a representative case, booting a lightweight Linux system on a VP up to the login prompt already results in approximately 2.5 million TLM transactions (excluding memory accesses)[1]. Second, many HW components in a virtual platform are accessed via memory-mapped I/O, making correct definition and interpretation of the memory map critical [8]. Errors such as overlapping address regions, incorrect peripheral offsets, or misaligned access sizes can lead to subtle and hard-to-detect bugs during SW execution which in the worst case crash the system unpredictably.

To tackle these challenges, this paper presents the open-source tool **ProtoLens**, available on GitHub[2]: For the modern, highly-configurable RISC-V VP++ [9], ProtoLens enables intuitive visualization and debugging of TLM transactions via an interactive web front-end. To achieve this, ProtoLens extracts memory maps from the virtual platform, and automatically generates an architecture graph from that information. Besides a generic transaction log, ProtoLens supports user-defined *Transaction View Module*s (TVMs) to enable peripheral-specific data visualization, and integrates with existing SW debuggers. Additionally, it enables users to control the simulation flow. At its core, ProtoLens hooks into the TLM bus of the virtual platform to capture the TLM transactions as they occur. The transactions are then visualized in the architecture graph using annotations and are then processed by the TVMs to produce user-defined output. Two case studies are presented to highlight the capabilities and advantages of ProtoLens. The first case study illustrates how ProtoLens complements classical debugging techniques by analyzing the behavior of a basic bare-metal SW example. The second case study shows that ProtoLens can handle data-intensive applications, such as the real-time reconstruction of graphics output from a VP running Linux and a 3D game, using transaction traces. Although the initial design of ProtoLens and the case studies are based on

979-8-3315-9379-7/25/$31.00 ©2025 IEEE

[1]Linux-6.10 measured on RISC-V VP++
[2]https://github.com/ics-jku/ProtoLens

RISC-V VP++, the modular architecture of ProtoLens allows it to be applied to other SystemC/TLM-based VPs that provide similar interfaces.

The paper is structured as follows: Section II reviews related work. Section III introduces relevant preliminaries, including SystemC/TLM, its transactions concept and RISC-V VP++, the current basis of ProtoLens. Section IV presents ProtoLens, followed by Section V, which demonstrates its application through case studies. The paper is concluded in Section VI.

II. Related Work

Visualization of SystemC designs was already targeted in it's early days. One of the first approaches was presented in [10]. However, this work modified the SystemC kernel, and considers low-level SystemC and not VPs using TLM communication. The approach has been later improved in [11], but still suffers from the second limitation.

In [12], further extended in [13], advanced HW/SW co-visualization techniques utilizing 3D rendering and virtual reality, have been proposed. The focus was on visualization of instantiated SystemC modules and their interconnections, the visualization of TLM transactions is not considered.

The paper [14] proposed the tool SycView for visualization and profiling of SystemC simulations, i.e. simulation timing diagrams on the level of SystemC processes are generated. Again TLM transactions are not supported.

More closely related to our work are methods towards the visualization of transactions streams created from TLM simulations. The *SystemC Verification Library* (SCV) [15] as well as the *Lightweight Transaction Recording for SystemC* (LWTR4SC) [16] provide implementations for recording TLM transactions into a text-based and binary format, respectively. SCViewer [17] and Surfer [18], [19] are open-source waveform viewers, that allow for visualizing transaction recordings. Recently, an approach leveraging dynamic runtime instrumentation of VPs to determine so-called function lifetime diagrams for visualizing HW/SW interactions in Surfer has been proposed [20]. However, no interaction with the SystemC simulation, debugging of SW running on a core, or user-defined visualizations of TLM data as provided by the proposed TVMs is possible.

Finally, there are also commercial virtual prototyping environments (e.g. Synopsys Virtualizer, Siemens EDA Vista, Cadence Helium Studio) which also offer visualization of TLM transactions, including transaction history, initiator/target relationships etc. However, these tools are proprietary and thus not freely available.

III. Preliminaries

This section provides essential background for the paper. Section III-A introduces VPs and SystemC/TLM with its concept of transactions. Section III-B presents RISC-V VP++, the current basis of ProtoLens, and key features used in our case studies.

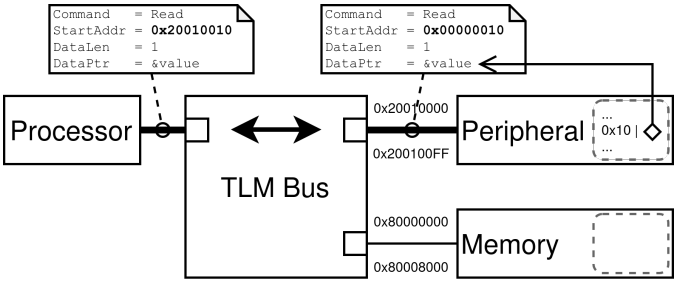

Fig. 1: Example: VP with TLM Read Transaction

A. VPs and SystemC/TLM

In general, VP are commonly used for sophisticated modeling and verification approaches like [21]–[28]. Today, VPs are predominantly created in SystemC/TLM, a standardized C++ class library that provides common building blocks and an event-driven simulation kernel to support the development and execution of HW simulations. HW system components, such as processors, buses and peripherals are called modules and implemented as C++ classes derived from the SystemC `sc_module` class. The two important aspects of such modules are behavior and communication. The behavior (e.g., the functionality of a peripheral) is described using SystemC processes and threads invoked by the simulation kernel, as well as methods triggered by communication. Communication is abstracted in SystemC/TLM using sockets and transactions. Sockets can either be configured as initiator of transactions (e.g. processor accessing a peripheral), or as target for transactions (e.g. peripheral accessed by a processor). Transactions are the data structures exchanged via sockets and describe the access itself. The most important attributes of a transaction are: (i) the command, which can be either a read or write; (ii) the start address of the access; (iii) a pointer to the data to be transferred (payload); and (iv) the data access length. Modules typically act as either initiator (e.g. processor modules), or as target (e.g. peripheral module). However, modules can also have multiple sockets configured in different roles. For example, a TLM bus module may have multiple target sockets to connect multiple processor core modules, and multiple initiator sockets to connect multiple peripheral modules. In a bus module, transaction routing is typically done based on the transaction's start address and a memory map that defines the start and end addresses of each peripheral in the address space. Within a SystemC simulation, transactions are executed as function calls with a reference to a transaction object.

Fig. 1 shows an example of a simple VP with one processor core, a memory and a peripheral module, where the processor intends to load one byte from address *0x20010010*. To achieve this, the processor module first constructs a read transaction object with the start address *0x20010010*, a data access length of one byte, and a pointer to the location where the loaded byte should be stored. It then calls the transport method (e.g. `b_transport(...)` in SystemC TLM-2.0) of its initiator socket, which is connected to the target socket of the TLM bus module – thereby invoking the bus module's transport

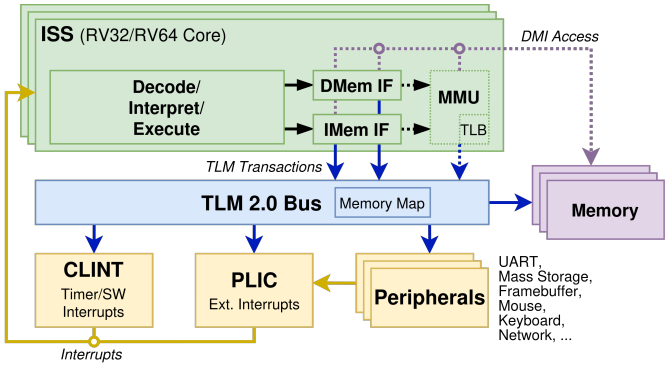

Fig. 2: RISC-V VP++ Architecture

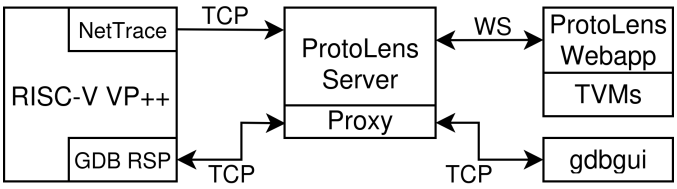

Fig. 3: Architecture of ProtoLens consisting of three components

method. The bus module uses the transaction's start address and the memory map to determine the addressed peripheral module. The bus adjusts the transaction's start address to a target module-relative address (*0x10*), then calls the transport method of the initiator socket connected to the peripheral, thereby invoking the peripheral module's transport method. The peripheral receives the transaction and interprets it as a request to read one byte from its internal memory at address *0x10*. It writes the retrieved value to the destination pointer provided in the transaction, then returns control to the bus, which in turn returns control to the processor. At this point, the transaction is complete, and the processor has the requested value available at the specified location.

B. RISC-V VP++

In this paper, we consider the versatile open-source SystemC/TLM-based RISC-V VP++ introduced in [9]. This VP was selected for its extensive capabilities and flexibility, which are briefly outlined below.

A key component of any hardware platform is the processor. The RISC-V *Instruction Set Architecture* (ISA) [29], [30] has gained significant traction in both academia and industry, thanks to its open standard and highly modular design. RISC-V VP++ supports the RISC-V ISA in 32-bit (RV32) and 64-bit (RV64) configurations. The architecture of the VP is outlined in Fig. 2. The VP includes fast interpreter-based *Instruction Set Simulator*s (ISSs) [31] for RISC-V, capable of simulating multiple cores, as indicated by the stacked ISS components in Fig. 2. The ISSs also include optional support for an *Memory Management Unit* (MMU) to realize *Virtual Memory Management* (VMM). A TLM-based bus links the ISSs, memory, and peripherals. A *CLINT* and *PLIC* provide timer and interrupt functionality. The VP comes with support for the *RISC-V "V" Vector Extension* (RVV) [32], and is used for advanced verification approaches [33]–[35]. RISC-V VP++ provides a variety of pre-built platform models which range from from basic bare-metal microcontroller systems to complex application processor platforms with an MMU, capable of running Linux with interactive graphical applications [36].

Two particularly relevant features of the RISC-V VP++ for this paper are its debugging and graphics output capabilities. For debugging, the VP includes a built-in GDB server that enables source-level debugging of SW executed within the VP. When activated, this server exposes a TCP-based interface conforming to the widely adopted GDB *Remote Serial Protocol* (RSP), which serves as a de facto standard for remote debugging. This compatibility allows the use of a broad range of debugger front-ends, including nearly all modern *Integrated Development Environment*s (IDEs).

The second feature, graphics output, is provided by a peripheral module called *VNCSimpleFB*. On the TLM side, this module exposes a memory-mapped region that functions as a framebuffer. SW running within the VP can generate graphical output by writing pixel data directly to this framebuffer. On the host system side, the VP offers a TCP-based interface compatible with the *Virtual Network Computing* (VNC) protocol. This allows any standard VNC client to display the graphics rendered by the SW running within the VP. With GUI-VP Kit [36], a fast-to-deploy and easy-to-use SW experimentation environment for Linux and interactive graphical applications is also available for RISC-V VP++.

IV. ProtoLens

In this section, we introduce our tool ProtoLens. First in Section IV-A, we present the architecture of ProtoLens, which consists of three components: the VP (in our case RISC-V VP++), the server and the web application. Thereafter, Section IV-B provides an in-depth look into the generation and content of the TLM transaction trace visualized by ProtoLens. Finally, Section IV-C outlines how ProtoLens enhances the classical debugging workflow by seamlessly integrating with traditional source-level debuggers.

A. Architecture

Fig. 3 depicts the architecture of ProtoLens. The extended RISC-V VP++ transmits TLM transaction data to the *ProtoLens Server* (PLS) via the NetTrace module, as illustrated in Fig. 3 (left and center, respectively). Additional processing of the transaction data is done by the PLS and the result is provided to the *ProtoLens Webapp* (PLW) via Websockets (abbreviated as WS in Fig. 3). The PLW provides users with application configuration, VP control capabilities, and with visualization of transaction data.

ProtoLens can run either in trace or in debug mode which is configurable through the PLW. In trace mode, the user can passively observe the transaction traces visualized in the PLW. During this mode, only the trace path (upper half of Fig. 3), including the VP NetTrace interface, the WS and the PLW is

active. In debug mode, the user can actively control execution both through the PLW and via a traditional debugger front-end. During this mode, both the trace path and the debug path (lower half of Fig. 3) are active, including the VP GDB RSP interface, the PLS Proxy, and gdbgui [37]. While ProtoLens supports any GDB front-end, it defaults to using the GDB front-end gdbgui, selected for its browser-based interface.

We will now give further insights into each of the three major components of ProtoLens:

a) RISC-V VP++: As mentioned above, transaction data is passed to ProtoLens via the NetTrace module, which extends RISC-V VP++ by providing a lightweight interface for trace capture. The NetTrace module uses a CSV-based data format (more details in Section IV-B) carried over TCP, to enable tool-agnostic data processing. At first, during the initialization of the VP, the memory map is sent to the PLS to enable dynamic visualization in the PLW. During SystemC simulation, the TLM bus, responsible for routing transaction-level communications between components, extracts the necessary data from a TLM transaction object and passes it to the NetTrace module. This passing of transaction data can be activated or deactivated via a command-line parameter given to the VP. If deactivated (i.e. the VP is used stand-alone / without ProtoLens), only a single conditional branch is introduced, resulting in virtually no performance degradation. An additional command-line flag halts the transaction initiator, triggering behavior equivalent to hitting a breakpoint in SW. This will discussed in more detail in Section IV-C.

b) ProtoLens Server*:* The Rust-based PLS functions as a central hub for data and control. The PLS connects to the NetTrace module and the GDB interface of the VP via TCP. The PLS further manages the launch and termination of the VP, executed as a sub-process of the operating system. Moreover, the PLS handles most of the computationally intensive tasks, to keep the user interface as lightweight as possible. For example, the PLS converts transaction payload to pixel data for transactions targeting the VP's graphics output. We will elaborate this functionality in more detail in the case study in Section V-B.

c) ProtoLens Webapp*:* The PLW receives data from the PLS via a Websocket connection. To visualize the transaction data obtained from the VP, the user interface generates a dynamic architecture graph. An example architecture graph, generated by ProtoLens connected to the basic microcontroller VP platform model included in RISC-V VP++, is shown in Fig. 4. The graph includes a generic core complex at the top (denoted *Core* in Fig. 4) and all memory mapped peripherals. Each peripheral is annotated with its start and end addresses. As the memory map of the VP is dumped on every startup, this graph can visualize changes of the memory map without any additional manual configuration. Transactions are visualized by green arrows drawn from the core complex to the target peripheral. The arrow direction indicates a read or write operation. The data access length and payload of the transaction are visualized along the arrow. This is shown in the center of Fig. 4, where a write transaction is currently being executed from the *Core* to the *SimpleTerminal*, with a length of *1* and carrying the hexadecimal byte *7A*. Transactions are also visible in the generic *Transaction Log* TVM (lower right corner in Fig. 4). The previously discussed transaction appears as the first entry in the *Transaction Log* table.

```
1  R;Core0;1;2004008;100;2;0000
2  W;Core0;8;8000000;120;2;FA09
```

Listing 1: Example transaction traces

TVMs are also used to provide peripheral-specific data visualization. One such TVM, the *Terminal* TVM can be seen in the bottom left corner of Fig. 4. The *Terminal* TVM filters all transactions directed to the *SimpleTerminal* peripheral, which handles character output on the VP console. The TVM interprets the transaction payload as printable characters and displays them accordingly – mirroring the output one would see on the VP console. In the example shown in Fig. 4, the character `z` is displayed, as it is the ASCII representation of the byte *7A*. Another example is the *Framebuffer* TVM which displays transaction payload directed to the VP's graphics output as pixels on a canvas, therefore reconstructing the displayed image. The *Framebuffer* TVM is discussed in detail in our case studies in Section V-B.

B. Transaction Tracing

Transaction tracing serves as the foundation for both the trace mode and debug mode described earlier. As outlined in Section III-A, during routing, the TLM bus first identifies the appropriate target and then invokes the corresponding method of that target peripheral. The TLM bus routing mechanism is extended to also forward transaction data to the NetTrace module (top left of Fig. 3), which subsequently transmits the data to the PLS. To enable a complete visualization, we extract most of the information carried by a TLM transaction. Listing 1 shows the transaction data format transmitted to the PLS. Each line in Listing 1 contains the following data elements: (i) the operation type (read or write), (ii) the initiator, (iii) the target peripheral id, (iv) the target memory address, (v) the simulation time [ns], (vi) the number of payload bytes, and the (vii) the transaction payload in hex.

All data elements except the initiator are part of the standard TLM-2.0 transaction object. However, to visualize the transaction from start to end, we also need the initiator of a transaction. To obtain the initiator, we utilize the TLM extension mechanism defined in the SystemC standard. We modified the transaction initiators – such as processor cores and DMA controllers – to append pointers to themselves in each generated transaction. This enables the TLM bus to identify the initiator. Since transaction objects are typically reused after creation, this mechanism introduces no significant overhead. As noted in Section IV-A, the data format presented in Listing 1 enables tool-agnostic processing. For instance, saving the received data to a file produces a TLM trace that can be utilized by other tools for further analysis of the VP simulation.

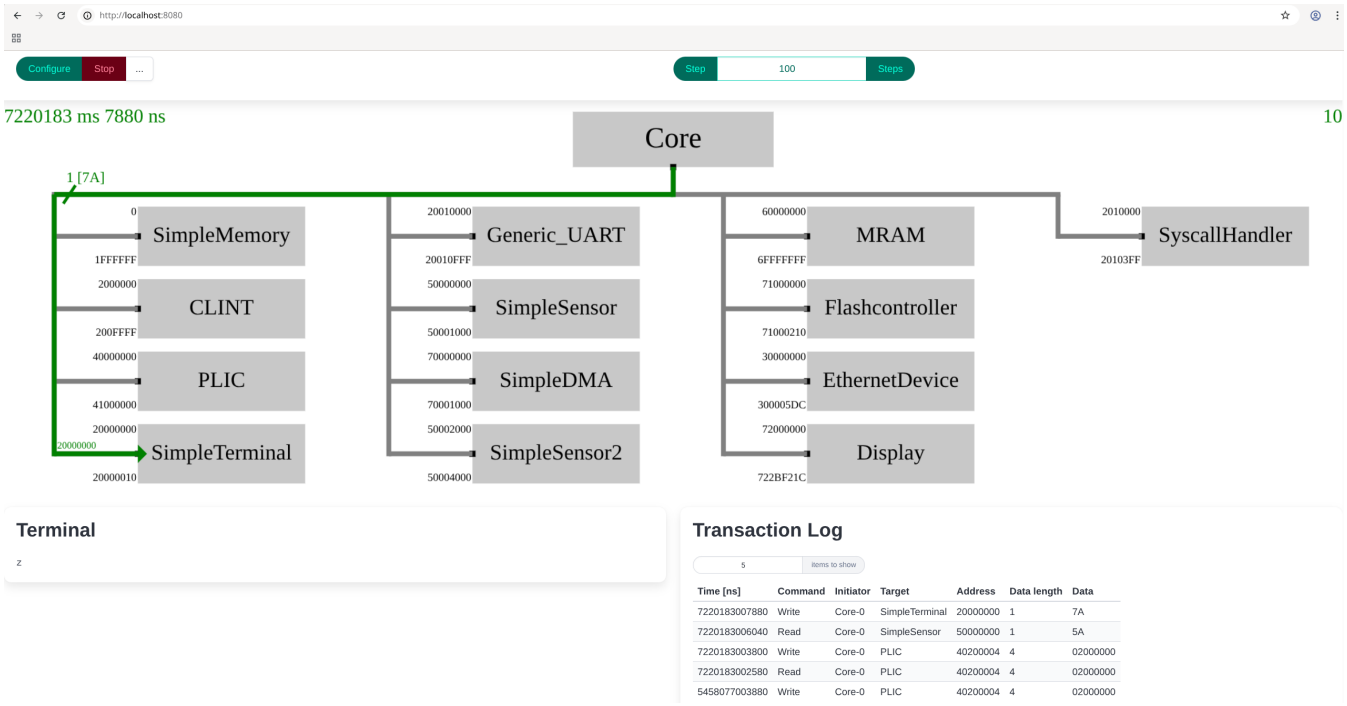

Fig. 4: *ProtoLens Webapp* user-interface. RISC-V VP++ running a Bare-metal SW accessing a sensor and terminal peripheral in debug mode

C. Debug Mode

In debug mode, ProtoLens provides interactive control over the VP by integrating transaction tracing with traditional source-level debugging. This mode supports a unified debugging workflow, allowing users to issue debugger commands through a traditional debugger front-end, while observing the system's behavior through transaction trace data visualized in the PLW. It builds on existing GDB RSP capabilities of the VP and enhances them with additional functionality for improved integration and control.

The VP already supports GDB-based debugging via its built-in GDB RSP interface, which can be enabled through a dedicated command-line argument. To further support fine-grained interaction, we extended the VP to provide an additional command-line option that halts the TLM transaction initiators after each routed transaction. From the debugger's perspective, this emulates hitting a breakpoint, enabling inspection of the system state after each transaction.

To bridge the gap between transaction visualization and traditional debugging, we introduced the PLS proxy (middle of Fig. 3). This component intercepts the GDB RSP communication between the debugger front-end and the VP, enabling ProtoLens to inject or interpret debugging commands. For example, the *continue* command can be dispatched directly from the PLW via the dedicated *step* and *steps* buttons (top of Fig. 4), reducing the need to switch between user interfaces.

While ProtoLens is compatible with any GDB RSP-compatible debugger front-end, it defaults to gdbgui [37], chosen for its browser-based design that enables seamless integration with the PLW. During debug mode, both the trace path and debug path are active, combining transaction monitoring with full-featured interactive debugging in a single environment.

V. Case Studies

To show the capabilities and advantages of ProtoLens we present two case studies. The first case study in Section V-A demonstrates how ProtoLens's debug mode complements classical debugging techniques by analyzing the behavior of a basic bare-metal SW example. The second case study in Section V-B demonstrates that ProtoLens can handle data-intensive applications, such as the real-time reconstruction of graphics output from a VP running Linux and a 3D game, using transaction traces.

A. Debugging Sensor Peripheral

In this case study, we demonstrate how ProtoLens's debug mode complements the classical debugging workflow by analyzing the behavior of a basic bare-metal SW example. For this, we utilize the basic microcontroller VP platform model included in RISC-V VP++, whose architecture graph is shown in Fig. 4.

Our example SW reads data from a sensor and prints the data via a terminal peripheral. The sensor peripheral is mapped onto the memory area from *0x50000000* to *0x50001000*. The sensor peripheral can be seen in the middle of Fig. 4, with the corresponding memory addresses displayed to the left of the module. The peripheral has two configuration registers named *scaler* and *filter*. These registers are located at *0x50000080* and *0x50000084* respectively. A 64 byte data frame is located at *0x50000000*. The sensor module periodically fills the data

```
1  // memory mapped register/inputs
2  static volatile char* const TERMINAL_ADDR =
       (char* const)0x20000000;
3  static volatile char* const SENSOR_INPUT_ADDR
       = (char* const)0x50000000;
4  static volatile uint32_t* const
       SENSOR_SCALER_REG_ADDR = (uint32_t*
       const)0x50000080;
5  static volatile uint32_t* const
       SENSOR_FILTER_REG_ADDR = (uint32_t*
       const)0x50000084;
6
7  // access/process sensor data
8  volatile _Bool has_sensor_data = 0;
9  void sensor_irq_handler() {
10   has_sensor_data = 1;
11 }
12 void dump_sensor_data() {
13   while (!has_sensor_data) {
14     asm volatile("wfi");
15   }
16   has_sensor_data = 0;
17   for (int i = 0; i < 64; ++i) {
18     *TERMINAL_ADDR = *(SENSOR_INPUT_ADDR + i)
           % 92 + 32;
19   }
20   *TERMINAL_ADDR = '\n';
21 }
22
23 int main() {
24   register_interrupt_handler(2,
         sensor_irq_handler);
25
26   *SENSOR_SCALER_REG_ADDR = 5;
27   *SENSOR_FILTER_REG_ADDR = 2;
28
29   dump_sensor_data();
30
31   return 0;
32 }
```

Fig. 5: Example: Bare-metal SW accessing a sensor and terminal peripheral

frame with new values and signals completion by triggering an interrupt. The update frequency is configurable through the *scaler* register, while the specific content written to the data frame is influenced by the *filter* register.

Fig. 5 shows the SW example. First an interrupt handler (Line 24) is installed, which sets a flag for the main SW (Line 10). After the interrupt handler was registered, the SW configures the *scaler* (Line 26) and *filter* (Line 27) registers. The SW then enters a *Wait for Interrupt* (WFI) loop (Line 14), and waits until an interrupt is triggered by the sensor peripheral. After the interrupt is handled, the sensor's data frame is read, converted into an ASCII character, and written to the terminal peripheral (Line 18) for console output.

To investigate the SW behavior, we configure the VP into the debug mode introduced in Section IV-C and start the VP from the PLW user-interface. When configured, gdbgui automatically starts and auto-connects to the VP via the PLS Proxy introduced in Section IV-C. We set a breakpoint in Line 19 of Fig. 5 and continue until the SW has hit the breakpoint. During execution the architecture graph shown in Fig. 4, automatically updates when new transactions are dispatched from the VP. When the breakpoint is hit, the VP has read the first element of the sensors data frame and wrote it to the terminal.

The Transaction Log TVM, located at the bottom right of Fig. 4, accurately displays this behavior. The second log entry records a read operation at the address of the sensor's data frame. The first entry indicates a write operation into the terminal's address space. This write operation is also visible in the graph visualization in the top half of Fig. 4. In the bottom left corner of Fig. 4, the Terminal TVM shows the character written to the terminal peripheral.

In summary, the integration of gdbgui and ProtoLens shown in this case study, provides a robust framework for step-by-step analysis, offering accurate and detailed visualizations that enhance the debugging process.

B. Tracing Linux Graphics Output

In this section we demonstrate the real-time reconstruction of the VP's graphics output using the transaction traces introduced in Section IV-B. We mirror the graphics output to a new TVM to display the image in the PLW.

In this case study, we utilize the application-processor *RV64* VP platform model capable of running Linux systems with interactive graphical applications, included in the RISC-V VP++. An architecture graph of the complex model is shown in Fig. 6. Compared to the simple model depicted in Fig. 4, the application-class model features a greater number of peripherals and, consequently, enhanced capabilities. One of these peripherals is the *VNCSimpleFB*, which is considered in this case study. The peripheral is shown as the last element in the third branch of the architecture graph in Fig. 6. The application-class VP runs a Linux operating system generated by GUI-VP Kit, presented in Section III-B. The executed OS image includes PrBoom, a port of a classic 3D game, which we use to demonstrate the capabilities of ProtoLens. All of our experiments are conducted on an Intel i7-8565U (4C/8T) with 16 GB of RAM.

To reconstruct the graphics output in the PLW, we introduce a new *Framebuffer* TVM. The *Framebuffer* TVM filters all transactions directed to the *VNCSimpleFB* peripheral, which handles the graphics output of the VP. The *VNCSimpleFB* is a memory-mapped peripheral that stores an image in the *RGB565* pixel format (2 bytes/pixel). The size of the image is *800x480* (WVGA). All pixel modifications of the framebuffer are done via TLM write transactions. This means that the transaction payload contains the pixel data written into the *VNCSimpleFB*. To determine the x and y coordinates where the pixels need to be drawn, we calculate the coordinate values from the transaction target address and the peripheral's start address. The coordinates are calculated by the PLS and combined with the pixel data into a custom data format which is then sent to the PLW. Doing the coordinate calculation in the PLS keeps the PLW as lightweight as possible. The *Framebuffer* TVM converts the *RGB565* pixel data to the *RGB888* format used by the HTML Canvas and draws the resulting pixels at the specified x and y coordinates.

Fig. 6 shows PrBoom [38] running on the VP in *640x480*. The bottom left of Fig. 6 shows the graphics output of the VP accessed via a VNC viewer. The bottom right of Fig. 6 shows the graphics output reconstructed by the *Framebuffer* TVM.

To reconstruct the *VNCSimpleFB* in real-time, ProtoLens has to handle a large volume of data every second. PrBoom redraws the whole game screen for every rendered frame which in our case contains $307,200$ pixels. In our experiments we observe average PrBoom framerates of 5 *Frames Per Second* (FPS). This corresponds to 1.536 million pixels per second which need to be processed. A transaction on *RV64* can carry at most 8 bytes. Since two bytes are used per pixel this leads to a capacity of four pixels per transaction. By dividing the

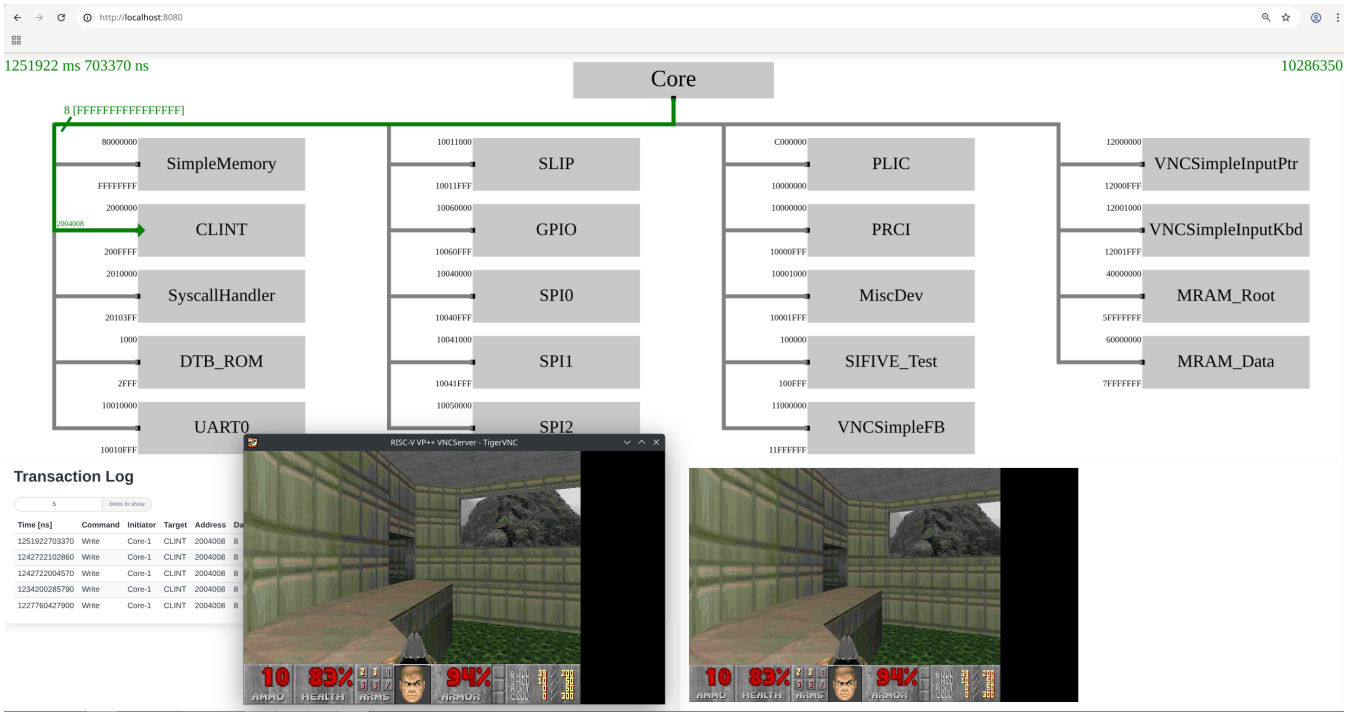

Fig. 6: PrBoom running on the RISC-V VP++. Bottom left: Graphics output of the VP, accessed via a VNC viewer. Bottom right: Graphics output reconstructed from transaction traces by the *Framebuffer* TVM.

pixels per second by the capacity of a transaction we receive 384k transactions per second. Therefore ProtoLens handles 384k transactions per second to refresh PrBoom with 5 FPS. This shows that ProtoLens provides useful real-time insight even in data-intensive applications.

VI. Conclusions

In this paper we introduced ProtoLens, a tool that provides a novel and robust approach of dynamic TLM transaction visualization. The integration with the highly-configurable, open-source RISC-V VP++, allows ProtoLens to be leveraged in a wide area of applications.

We have shown how ProtoLens can be used during debugging to support the developer with real-time transaction and architecture visualization. Additionally ProtoLens can be easily extended via the introduced TVMs, to provide peripheral-specific data visualization. With the real-time reconstruction of the VP's graphics output by a TVM we showed that ProtoLens is capable of handling data-intensive applications.

Although the initial design of ProtoLens and the case studies are based on RISC-V VP++, the modular architecture of ProtoLens allows it to be applied to other SystemC/TLM-based VPs that provide similar interfaces. ProtoLens, as well as the NetTrace extension for RISC-V VP++ are available as open-source on GitHub.

Future work will focus on integrating transaction-enhanced waveform viewers, further complemented by programmable waveform analysis techniques, as proposed in [39], but lifted to TLM.

Acknowledgments

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

References

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

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

[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*, 2012, pp. 685–690.

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

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

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

[7] *OSCI TLM-2.0 Language Reference Manual*, OSCI, 2009. [Online]. Available: https://www.accellera.org/images/downloads/standards/systemc/TLM_2_0_LRM.pdf

[8] N. Mook, E. de Kock, B. Arts, S. Chakraborty, and A. van Deursen, "Modeling and analysis technique for the formal verification of system-on-chip address maps," in *Design, Automation and Test in Europe Conference*, 2025.

[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] D. Große, R. Drechsler, L. Linhard, and G. Angst, "Efficient automatic visualization of SystemC designs," in *Forum on Specification and Design Languages*, 2003, pp. 646–657.

[11] C. Genz and R. Drechsler, "System exploration of SystemC designs," in *IEEE Computer Society Annual Symposium on VLSI*, 2006, pp. 335–342.

[12] R. Drechsler and M. Soeken, "Hardware-software co-visualization: Developing systems in the holodeck," in *IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems*, 2013, pp. 1–4.

[13] R. Drechsler and J. Stoppe, "Hardware/software co-visualization on the electronic system level using SystemC," in *International Conference on VLSI Design*, 2016, pp. 44–49.

[14] D. Becker, M. Moy, and J. Cornet, "SycView: Visualize and Profile SystemC Simulations," in *3rd Workshop on Design Automation for Understanding Hardware Designs*, 2016.

[15] *SystemC Verification Library 2.0.1*, Accellera Systems Initiative, 2017. [Online]. Available: https://www.accellera.org/images/downloads/standards/systemc/scv-2.0.1.tar.gz

[16] *Lightweight transaction recording for SystemC*, Minres, 2024. [Online]. Available: https://github.com/Minres/LWTR4SC

[17] *SCViewer*, Minres, 2024. [Online]. Available: https://github.com/Minres/SCViewer

[18] "Surfer," https://gitlab.com/surfer-project/surfer, 2024.

[19] F. Skarman, L. Klemmer, D. Große, O. Gustafsson, and K. Laeufer, "Surfer – an extensible waveform viewer," in *International Conference on Computer Aided Verification*, 2025, pp. 392–404.

[20] 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.

[21] W. Mueller, M. Becker, A. Elfeky, and A. DiPasquale, "Virtual prototyping of cyber-physical systems," in *Asia and South Pacific Design Automation Conference*, 2012, pp. 219–226.

[22] F. Pêcheux, C. Grimm, T. Maehne, M. Barnasconi, and K. Einwich, "SystemC AMS based frameworks for virtual prototyping of heterogeneous systems," in *IEEE International Symposium on Circuits and Systems*, 2018.

[23] E. Fraccaroli, M. Lora, S. Vinco, D. Quaglia, and F. Fummi, "Integration of mixed-signal components into virtual platforms for holistic simulation of smart systems," in *Design, Automation and Test in Europe Conference*, 2016, pp. 1586–1591.

[24] V. Herdt, H. M. Le, D. Große, and R. Drechsler, "Verifying SystemC using intermediate verification language and stateful symbolic simulation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 7, pp. 1359–1372, 2019.

[25] V. Herdt, D. Große, H. M. Le, and R. Drechsler, "Verifying instruction set simulators using coverage-guided fuzzing," in *Design, Automation and Test in Europe Conference*, 2019, pp. 360–365.

[26] ——, "Early concolic testing of embedded binaries with virtual prototypes: A RISC-V case study," in *Design Automation Conference*, 2019, pp. 188:1–188:6.

[27] P. Pieper, V. Herdt, D. Große, and R. Drechsler, "Dynamic information flow tracking for embedded binaries using SystemC-based virtual prototypes," in *Design Automation Conference*, 2020, pp. 1–6.

[28] ——, "Verifying SystemC TLM peripherals using modern C++ symbolic execution tools," in *Design Automation Conference*, 2022, pp. 1177–1182.

[29] A. Waterman and K. Asanović, *The RISC-V Instruction Set Manual; Volume I: Unprivileged ISA*, SiFive Inc. and UC Berkeley, 2019.

[30] ——, *The RISC-V Instruction Set Manual; Volume II: Privileged Architecture*, SiFive Inc. and UC Berkeley, 2019.

[31] 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.

[32] 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.

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

[34] 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.

[35] 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.

[36] 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.

[37] C. Smith. gdbgui - a browser-based frontend to gdb. [Online]. Available: https://www.gdbgui.com/

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

[39] L. Klemmer and D. Große, "WAVING goodbye to manual waveform analysis in HDL design with WAL," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 43, no. 10, pp. 3198–3211, 2024.