GUI-VP Kit: A RISC-V VP Meets Linux Graphics – Enabling Interactive Graphical Application Development

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

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

ABSTRACT

Today, *Virtual Prototypes* (VPs) are heavily used to enable early software development and to accelerate the design process. The aim of this work is twofold: (i) enable the early development of interactive graphical applications running on Linux, and (ii) provide an easy-to-use and configurable solution for RISC-V.

In this paper, we present *GUI-VP Kit*. *GUI-VP Kit* includes *GUI-VP*, a greatly extended and improved RISC-V VP, as well as configurations to build a runnable Linux environment, and input/output drivers that form the interface between peripherals and Linux applications. In our experiments employing *GUI-VP Kit*, we show that well-known X-applications can be executed in *GUI-VP* using a VNC network connection. Moreover, we demonstrate reasonable speed for a Linux port of a classic first-person 3D-game.

CCS CONCEPTS

• **Hardware** → **Simulation and emulation**; • **Software and its engineering** → **Development frameworks and environments**; • **Computer systems organization** → *Embedded software*; • **Information systems** → *Open source software.*

KEYWORDS

Virtual Prototype, RISC-V, SystemC, TLM, Simulation, Linux, Graphics, Software development, Network, X Window System, VNC

ACM Reference Format:
Manfred Schlägl and Daniel Große. 2023. *GUI-VP Kit*: A RISC-V VP Meets Linux Graphics – Enabling Interactive Graphical Application Development. In *Proceedings of the Great Lakes Symposium on VLSI 2023 (GLSVLSI '23), June 5–7, 2023, Knoxville, TN, USA*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3583781.3590253

1 INTRODUCTION

A *Virtual Prototype* (VP) is a high-level, executable model of the entire *Hardware* (HW) platform which runs unmodified production *Software* (SW) [11, 18]. Since VPs allow to parallelize the HW and SW development and improve the communication through the supply chain they provide an efficient and effective means for the development of electronic systems. Significant cost reductions are achieved with VPs by exploring design alternatives and validating system functionality *before* the physical HW is built.

Today, VPs are predominantly created in SystemC. SystemC is a standardized class library for C++ (IEEE 1666, [17]) and provides *Transaction Level Modeling* (TLM) [20] to describe the communication and interactions between components of a system in an abstract way. As a consequence, simulation speed is orders of magnitude higher in comparison to *Register Transfer Level* (RTL) models [11]. For a broader overview on SystemC we refer the reader to [12–14].

A central element of a HW platform is the processor. In the recent past, the open and royalty-free *Instruction Set Architecture* (ISA) RISC-V [21, 22], which features an extremely modular design, gained enormous momentum in academia and industry. Many RISC-V processors, open-source and commercial ones, have been developed. This also holds for VP-based solutions supporting RISC-V, see for instance [3, 6, 7, 10, 19].

Contribution: In this paper, we consider the open-source SystemC *RISC-V VP* introduced in [1, 15, 16] that is available on GitHub [6]. While this VP has been used in several (research) projects, all these projects focused either on bare-metal SW, or on application SW on top of a "small" *Operating System* (OS) without virtual memory management. However, when it comes to graphical application SW development on top of Linux, the *RISC-V VP* has a number of limitations. To address these, we propose several extensions and improvements to *RISC-V VP*. The extensions and improvements are a consequence of a **careful analysis of the complex simulation stack** which has to be understood and managed to (i) realize a Linux Bring-Up on a VP, (ii) provide graphical output, and (iii) provide input for the to be developed graphical application.

Altogether, this results in ***GUI-VP Kit*** which includes the ***GUI-VP*** as well as configurations to build a Linux system and drivers that form the interface between peripherals and Linux applications. Hence, *GUI-VP Kit* enables the execution of complex Linux graphics frameworks and interactive GUI applications, as we will show later. In addition, the VP's networking capabilities allow users to apply the same well-known SW development workflows used for real embedded systems, such as rapid development, test, and debug iterations using *Network File System* (NFS) or *Secure Shell* (ssh) for SW deployment and the *GNU Debugger* (GDB) for remote debugging.

To demonstrate the potential of *GUI-VP Kit* for the development process of interactive graphical applications, we present two kinds of experiments. First, we show the use of X.Org [9] applications as an example for a non-trivial graphics framework. The full stack (Linux boot on *GUI-VP*, X.Org startup, application start) is available in less then 3 minutes and runs smoothly thereafter. Second, we

GLSVLSI '23, June 5–7, 2023, Knoxville, TN, USA

ACM ISBN 979-8-4007-0125-2/23/06.
https://doi.org/10.1145/3583781.3590253

investigate the performance of *PrBoom*, a Linux port of a classic first-person 3D-game [5]. We analyze different configurations and receive reasonable frame rates considering the complexity of the 3D game scenes.

Finally, we open source[1] *GUI-VP Kit* to facilitate further advance of the RISC-V open-source ecosystem in general.

Related Work: Above we have already mentioned some examples of VP-based solutions supporting RISC-V. In the open-source space there are solutions using QEMU [7]. However, their ISS and platform parts are not in the SystemC world, hence accuracy and granularity cannot be modeled following the SystemC standard. [10] presents an interesting combination of QEMU and SystemC. However, it is not available and it does neither target graphic applications nor respective interfaces for input. Both, [19] and [3] have no memory management unit and hence do not support Linux. To the best of knowledge, the RISC-VP [6], which we use as basis, is the only freely available SystemC-based VP that is capable of booting Linux. Finally, the commercial VP tools, e.g. Synopsys Virtualizer or Siemens EDA/Mentor Vista, are proprietary. In contrast, *GUI-VP Kit* is available as open-source on GitHub.

2 *GUI-VP KIT* SIMULATION STACK

Figure 1 shows the complex simulation stack of the *GUI-VP Kit* with all extensions and improvements that enables the development of interactive graphical Linux applications. The heart is the *GUI-VP* (Hardware Level), that runs on top of the Host System (Host Level). The *GUI-VP* creates a simulated environment for the Linux kernel (Operating System Level) as basis for the Linux Userland, where the graphic frameworks reside and the graphical SW applications are running (Application Level).

The green blocks in Figure 1 show the components that are provided by *RISC-V VP* [15, 16]. The VP comes with RISC-V 32 bit (RV32) and 64 bit (RV64) models capable of running Linux. This includes: (i) *Instruction Set Simulators* (ISSs) to execute RV32 or RV64 instructions, (ii) a *Memory Management Unit* (MMU) for virtual memory management, (iii) Memory to hold programs and data, (iv) *Core Local-* and *Platform-Level Interrupt Controllers* (CLINT and PLIC) that provide system timers and interrupt handling, (v) two UARTs, where one is used as console for user textual interaction and one to realize network functionality using the *Serial Line Internet Protocol* (SLIP) and the *Network Tunnel Driver* (TUN).

A runnable Linux system image is required as a basis for our *GUI-VP Kit*. We discuss the basic steps of this Linux bring-up in Section 3. This also covers the initial *Device Tree* (DT), a data structure passed to the Linux kernel, that describes the available HW components, so that the kernel knows how to use and manage them.

The orange and red blocks in Figure 1 show the components we replace (orange) and add (red) to support interactive graphical applications. To realize graphical output and mouse input, we adopt the *Virtual Network Computing* (VNC) graphical desktop-sharing system that provides an interface for generic VNC clients on the Host Level. We integrated VNC in *GUI-VP* using the powerful *libVNCServer* open-source C library [4] and the custom C++ wrapper *VNCServer*. Based on this, we implemented the SystemC TLM modules *VNCSimpleFB* for the graphics side and *VNCSimpleInput* for the mouse input side.

In order for Linux to use the new modules, we customize the DT and add drivers for them into the kernel. We add support for the *simple framebuffer* driver family as a counterpart to *VNCSimpleFB*, which supports either the *Linux Frame Buffer* (fbdev), or the *Direct Rendering Manager* (DRM) interface. As counterpart to *VNCSimpleInput*, we introduce the newly developed *simpleinput* driver that provides an *Event Device* (evdev) interface.

Details on our respective extensions and their interfaces are given in Section 4 for the graphics side, and in Section 5 for the mouse input side. As final modification, we introduce the alternative CLINT implementation *lwrt_clint*, which ensures that the clock rate of the simulated system wall clock matches that of the host.

Finally, bringing the whole *GUI-VP Kit* to live means to run applications at the Application Level. The respective components are covered in Section 6.

3 FOUNDATION FOR *GUI-VP KIT*: LINUX BRING-UP

In this section we lay the foundation for *GUI-VP Kit* and show all necessary steps to run Linux on it. As a starting point for *GUI-VP*, we first show basic changes to *RISC-V VP* in Section 3.1 that are made to facilitate all further extensions. After that, we illustrate how Linux images are created that can be booted on the *GUI-VP* in Section 3.2. Finally, with the *Device Tree* (DT), we present the form in which *GUI-VP* is described to the Linux kernel so that it can correctly handle the HW provided to it in Section 3.3.

3.1 *GUI-VP*: Generic & Configurable Top-Level

The *RISC-V VP* contains two SystemC top-levels, from which a model of an RV32 and a model of an RV64 multi-core system are created. To also get runnable single-core models and to simplify further extensions, we merge the two top-levels of *RISC-V VP* into a single, parameterizable one. As final result the *GUI-VP* build system can now create four models from this single SystemC top-level: (i) RV32 single-core, (ii) RV32 quad-core, (iii) RV64 single-core, (iv) RV64 quad-core.

3.2 Runnable Linux System Image

To get a runnable RISC-V Linux system image including bootloader, kernel and user space we use Buildroot in its most recent version 2022.11.1. Buildroot is a simple, efficient and easy-to-use tool to generate embedded Linux systems through cross-compilation [2].

Buildroot itself does not contain sources for Linux system components. Instead, it comes with an extensive collection of automated recipes for components, that describe what dependencies exist on other components, where to find the source code, how to build it, and how to assemble the resulting artifacts into a runnable system. This whole process is highly customizable by a configuration file.

To have a basis for our *GUI-VP* RV32 and RV64 models, we need two Buildroot configurations. We use the respective default configurations for RV32 and RV64 which are included in Buildroot as a starting point and customize them in the following way:

- Add support for generation of OpenSBI, which handles the initial board bring-up and Linux kernel start.

[1] https://github.com/ics-jku/GUI-VP_Kit

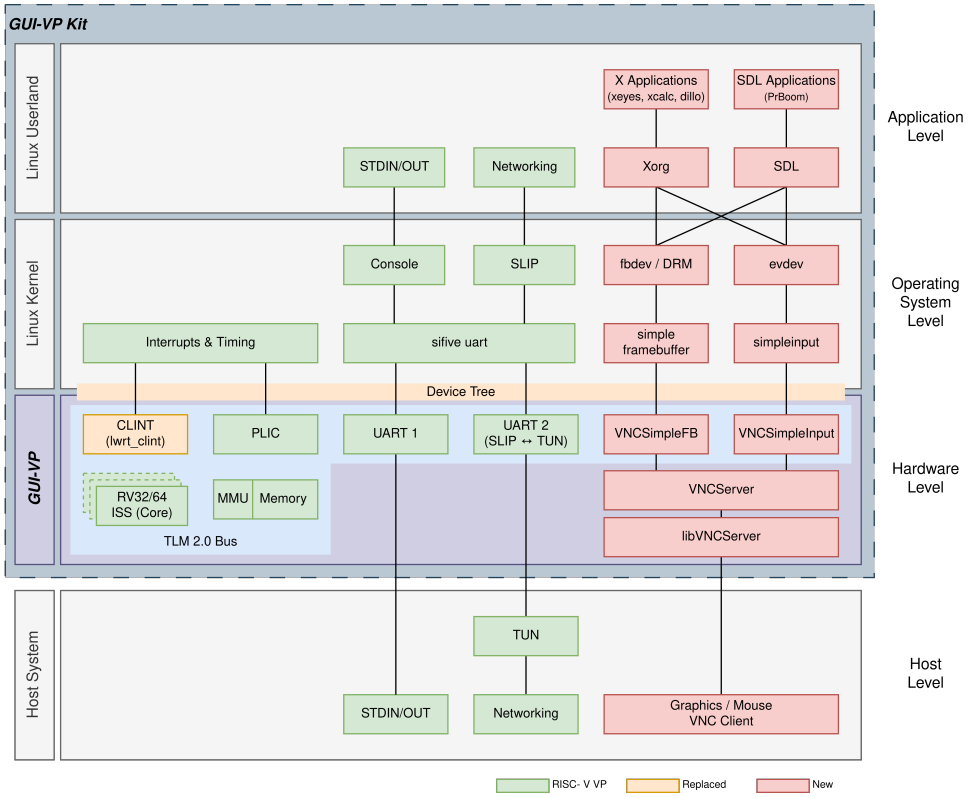


Figure 1: *GUI-VP Kit* Simulation Stack

- Add support for Linux kernel image generation. The Linux kernel configurations are also based on default configurations for RV32 and RV64 included in the kernel sources. However, they are optimized by removing unneeded functionality (e.g. unneeded drivers).
- Add support for SLIP (Linux kernel and user space tool) to enable networking.
- Add tools for debugging and testing
- Enable generation of a fully integrated, bootable single SW image that contains the OpenSBI Bootloader, the Linux Kernel and the root file system.

3.3 Device Tree

Having now the *GUI-VP* models and bootable Linux images, there is still one part missing to get running systems: The description of the simulated HW components of the models, so that the Linux kernel knows how to use and manage them, the DT.

The DT is typically written in a human readable form, the *Device Tree Source* (DTS) and then compiled to a compact binary representation, the *Device Tree Binary* (DTB). In our case the DT contains:

- The number and descriptions of the cores (RV32/RV64), including the available extensions and the address-translation scheme in use by the MMU.
- The size of the memory and its location in the memory map.
- The interrupt controllers (CLINT and PLIC) and their register locations and parameters.
- The peripherals, in our case two UARTs. Their location, interrupt sources, default baud rate and an indication which driver the Linux kernel should use to handle the peripheral, the *compatible string*.

Similar to the VP top-levels in Section 3.1, we also introduce a single parameterizable DTS. Based on this, the build system automatically creates the DTBs for the four VP models presented above

With the resulting build system that creates the executable *GUI-VP* models, the Linux system images, and the DTBs, we now have the foundation for our *GUI-VP Kit*. In the following sections we show how this initial version is extended to support GUI applications by adding support for graphics output (Section 4) and mouse input (Section 5).

4 GRAPHICS OUTPUT

In this section, we describe the realization of the graphical output from applications running within *GUI-VP* to the outside world.

The main concept used for this, is that of the *Frame Buffer* (FB). A FB is a specific memory area that contains a bitmap of an image, in which all pixels of the image are arranged one after the other in a specific format. A typical use case for FBs is the realization of efficient interfaces between graphics HW and SW: The SW updates part or all of the FB content according to its graphics output. The graphics HW periodically outputs the entire FB to a display device.

Our realization of the graphics output consists of three major parts: First, the interface, that provides the graphical output from

GUI-VP to the outside world, is presented in Section 4.1. Second, the interface provided by *GUI-VP* to the Linux system, is described in Section 4.2. Third, and finally, the interface between the HW and the Linux applications provided by a Linux kernel driver, is covered in Section 4.3.

4.1 Interface: Hardware/Host Level

First, we focus on the interface from *GUI-VP* to the outside world. To provide a generic and easy way to access the output, we adopt the VNC graphical desktop-sharing system for this task.

VNC relies on the *Remote Frame Buffer Protocol* (RFB), which provides transport of graphical output from a server to a client over a TCP/IP based network, and also the transport of input events (e.g. mouse, keyboard) in the opposite direction. By using such a widely spread solution, users can utilize a large number of different clients according to their requirements and personal preferences.

For integration of VNC in the *GUI-VP* we use the powerful *libVNCServer* open-source library. This C library not only provides a fully functional VNC/RFB server implementation, but can also be configured to perform protocol handling in dedicated threads. This is particularly useful in our case, as it means that protocol handling does not add to the load on the SystemC thread and thus does not reduce simulation performance.

For the basic integration of *libVNCServer*, we implement the C++ class *VNCServer*, which not only serves as the basis for graphics output, but also facilitates the integration of mouse event propagation as presented later in Section 5.

After configuration and initialization of *VNCServer*, its run-time interface for graphics consists of two parts. First, a pointer to the *RFB Frame Buffer* (RFB-FB) with configured size (width and height) and format (e.g. color order, bits per pixel). Second, a function to indicate a modified area of the RFB-FB and to trigger an update on RFB (RFB Trigger).

4.2 Interface: Hardware/Operating System Level

The second major part is the hardware interface provided by *GUI-VP* to the Linux system. This is realized by the SystemC module *VNCSimpleFB*. Using SystemC TLM, it provides a virtual, memory-mapped peripheral that translates accesses to its memory region to changes in the RFB-FB provided by *VNCServer*.

Conceptually, we could directly map TLM transactions to the RFB-FB and call the RFB trigger on any change. However, TLM transactions are generated by load/store instructions and are therefore very fine granular (8 to 64 bit). This would result in RFB triggers being called very frequently, which would have a negative impact on simulation performance. To prevent this, we decouple FB updates from RFB triggers in the following way: Each TLM transaction is applied directly to the RFB-FB. When a change is made (TLM write), the RFB trigger is not called, but instead a flag is set to indicate the change. The flag is checked periodically by another SystemC thread. If the flag indicates a change, the RFB trigger is called and the flag is reset.

To finally enable *VNCSimpleFB* in *GUI-VP*, we integrate it in our SystemC top-level and map it at address *0x11000000*.

4.3 Operating System Level

The third and last major part is the Linux driver that handles the HW and provides a standardized interface for Linux applications. For this we utilize the already existing drivers in the category of *simple-framebuffer*. These drivers are normally used to access FBs that were already configured and initialized by other SW that ran before the Linux kernel was started (e.g. firmware, bootloader, ...). They don't include any functionality to configure FB peripherals, but rather provide a simple representation of a memory mapped FB in the Linux kernel.

In a first step, we have to make the FB known to the Linux kernel. We therefore add a new node to our DT, that contains:

- The address range, the FB is mapped: *0x11000000* to *0x11ffffff*
- The pixel format: *rgb565*, which corresponds to 2 bytes per pixel, where the lowest 5 bits represent blue, the next 6 bits represent green and the highest 5 bits represent red
- The resolution (width, height): 800x480 (WVGA)
- The number of bytes in a line: $800 * 2$ bytes per pixel = 1600
- The compatible string *simple-framebuffer*, from which the Linux kernel determines the driver to handle the peripheral

The second step is to enable support for the driver in the Linux kernel configuration. Here we can choose between two alternative implementations. One is enabled by *CONFIG_FB_SIMPLE* and provides support for the relatively compact and simple, but deprecated fbdev interface. The other is enabled by *CONFIG_DRM_SIMPLEDRM* and provides support for the more powerful, but also more complex DRM. Since DRM is more modern and flexible and comes with an emulation of fbdev anyway, we choose this approach as default.

After these last steps, *GUI-VP Kit* is now able to run Linux applications with graphics output. In Section 5 we now discuss how *GUI-VP Kit* is extended to support mouse input.

5 MOUSE INPUT

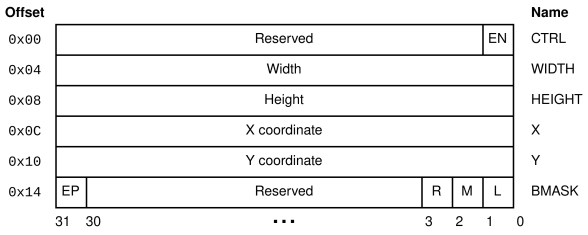
This section covers the realization of the propagation of mouse events from the outside world to SW running within *GUI-VP*.

Similar to the graphics output described before, our implementation of mouse input again consists of three main parts: First, the interface that provides the mouse input from the outside world to *GUI-VP*, is presented in Section 5.1. Second, the interface provided by *GUI-VP* to the Linux system, is described in Section 5.2. Third, and finally, the interface between the HW and the Linux applications provided by a Linux kernel driver, is covered in Section 5.3.

5.1 Interface: Hardware/Host Level

We will first focus on how the events are propagated to *GUI-VP*. Events are provided by *libVNCServer* by using C callbacks that can be installed in the initialization phase. However, these mechanism is not directly applicable to call member methods of C++ Classes as used in *GUI-VP*. To solve this, we first take a closer look at the interface provided by *libVNCServer*.

The arguments of the C callbacks contain not only event related data, but also a pointer to a data structure *rfbClient* corresponding to the client from which the event originated. Each *rfbClient* has a pointer to the *rfbScreen* data structure corresponding to its parent server, which is created during the initialization phase. Finally, *rfbScreen* contains a member *screenData* implemented as a void



EN: ENables event propagation; EP: Event Pending
R/M/L: Right/Middle/Left mouse button

Figure 2: Simpleinput Register Interface

pointer that can be used to provide custom, application-specific data. This is now used in the following way: In the initialization phase, we set the *screenData* such that it points to our *VNCServer* instance. We then install static C functions as callbacks for connection-, mouse-, and keyboard-related events. This functions now serve as wrappers for C++ methods. They extract the *VNCServer* instance via *rfbClient* as described above and call event handling member methods of this instance corresponding to the event type. With this we now have *VNCServer* method calls for all events provided by *libVNCServer*.

To propagate the input events to other C++ objects, we introduce the new interface *VNCInput_if*. This interface is realized as abstract C++ class and declares event handling member functions for mouse and keyboard events. *VNCServer* provides a method to register objects of classes that implement this interface. If such an object is registered, the implemented interface methods are called by the event handling methods of *VNCServer*.

5.2 Interface: Hardware/Operating System Level

The next major component is the HW module, that handles the propagation of the input events to the SW that is running on *GUI-VP*. This is realized by the SystemC module *VNCSimpleInput*. The module implements the *VNCInput_if* interface and is registered in the *VNCServer* instance by the top-level of *GUI-VP*.

In contrast to the FB handling presented in Section 4, the Linux kernel does not come with a generic driver for this task. We therefore have to specify our own register-based interface for the module and we have to implement a corresponding Linux driver for it.

The register interface consists of the six registers show in Figure 2. If the *EN* bit is set by the SW in the *CTRL* register, all mouse events that are propagated by the *VNCServer* callbacks are put at the end of a queue that is restricted to hold 10 events. A SystemC thread monitors the queue and triggers an interrupt when events are present. A beginning TLM read of *BMASK* removes the first event from the queue and loads it in the registers *X*, *Y* and *BMASK*. If there are still queued events, the *EP* bit in *BMASK* is set to indicate, that further events are ready to be read.

To finally enable *VNCSimpleInput* in *GUI-VP*, we integrate it in our SystemC top-level and map it at address *0x12000000*.

5.3 Operating System Level

As counterpart to the *VNCSimpleInput* module we implement a new Linux driver *simpleinput*. The driver code is provided by *GUI-VP Kit* as a patch for the Linux kernel and is automatically applied when the kernel is built.

The driver registers itself as so called *platform driver* with a specific device id and a probe function. On Linux kernel boot, the DT is checked for a node with a *compatible string* matching the device id. If this is the case, the driver's probe function is called, which requests all required resources and registers with the Linux kernel's input subsystem. The resources are the area containing the memory-mapped registers and the interrupt, for which a handler function is also installed. Both resources are described in a node in the DT, as we will see below.

When registering with the input subsystem, the driver must also define its capabilities, i.e. the type and code of events it provides (defined in the Linux kernel sources). For the mouse buttons, it defines the mouse buttons codes *BTN_LEFT*, *BTN_MIDDLE* and *BTN_RIGHT* of type *EV_KEY*. The values for this type of events are implicitly interpreted as boolean, reflecting either pressed (true) or released (false). The mouse coordinates are provided by VNC as absolute values, so they are defined as codes *ABS_X* and *ABS_Y* of type *EV_ABS*. For these types of events, the driver must explicitly specify a range that their values can take. This ranges are determined by reading the values from the registers *WIDTH* and *HEIGHT*.

Also part of the registration is the installation of handlers that are called by the input subsystem, when the according input device is opened and closed. In the open handler the driver enables event propagation and thus also interrupts by setting *EN* in the *CTRL* register. Correspondingly, event propagation is disabled in the close handler by resetting the *EN* bit. In this way, unnecessary load in the VP and kernel is avoided when the device is not in use.

The event propagation itself is finally done in the interrupt handler function. This function first reads the current event from the *BMASK*, *X* and *Y* registers. As described above, it is important that *BMASK* is read first, as this ensures that valid data from the HW module is loaded into all other registers. The coordinate values read from *X* and *Y* are reported directly to the input subsystem as *ABS_X* and *ABS_Y*. For the mouse buttons, the value read by *BMASK* is compared to the last value read to avoid redundant reporting of events. Only if there is a difference, the key bits *L*, *M* and *R* are decoded and reported to the input subsystem accordingly as *BTN_LEFT*, *BTN_MIDDLE* and *BTN_RIGHT*. The completion of reporting the current event to the input system is indicated by a call of the function *input_sync*. To improve event forwarding throughput, this entire sequence is repeated as long as the *EP* bit of *BMASK* indicates that more events are pending in the HW queue.

The final steps are now to enable support for the driver in the Linux kernel configuration and to make the device known to the Linux kernel in the DT.

The former is done by adding *CONFIG_INPUT_SIMPLEINPUT* to the configuration. For the latter, a new node is added to our DT, that contains the following descriptions:

- The addr. range the module is mapped: *0x12000000-0x12000fff*
- The interrupt parent and number: PLIC interrupt no. 10
- The compatible string *simpleinput*, from which the Linux kernel determines our driver to handle the peripheral

After these last steps, *GUI-VP Kit* is now able to run Linux applications with graphics output and mouse input. This is now demonstrated in Section 6.

6 DEMONSTRATION AND EVALUATION

In this section, we present the experiments conducted with *GUI-VP Kit*. All experiments were performed on a standard notebook with an Intel Core i7-8565U and 16 GiB RAM under Debian 11.

Due to the generic nature of the simulation stack provided by *GUI-VP Kit*, nearly any Linux graphics framework and application can be run on *GUI-VP*. As an example for this, we present the use of the very common graphical *X Window System* X.Org, in Section 6.1. We show two examples of interactive X applications running on the Linux system on *GUI-VP*, including a lightweight web browser.

In Section 6.2 we show that the performance is sufficient for more complex applications. Here, we use PrBoom, a Linux port of a classic first-person 3D-game, which is based on the *Simple DirectMedia Layer* (SDL) library [8]. We evaluate the performance based on the average *Frames Per Second* (FPS) that PrBoom achieves on different *GUI-VP* models and in different configurations.

6.1 X Window System and Applications

On Unix-like operating systems the *X Window System* is a very common framework to realize GUI applications. It is designed as a client-server architecture. The X server only manages the devices (graphics card, mouse, …) and provides an abstract interface for basic operations like graphics output and keyboard and mouse input. The applications themselves are implemented as X clients. They use the interface provided by the X server for their graphics output and event input.

For the following demonstrations we use the wide-spread, free and open-source implementation X.Org, which is provided by the *X.Org Foundation* [9] and also available in Buildroot.

For graphics output, X.Org uses the Linux kernel's DRM interface. As presented in Section 4, DRM transports graphics output via the *simple-framebuffer* Linux driver to our TLM module *VNCSimpleFB* in *GUI-VP*. From there, we transport the image data via our *VNCServer* wrapper, the *libVNCServer* library and RFB to the VNC client. For mouse input, X.Org uses the Linux evdev input device, that was presented in Section 5. Mouse input from the VNC client is passed to *GUI-VP* via RFB, *libVNCServer* and *VNCServer*. From there, we deliver the events to our *VNCSimpleInput* TLM module. This module is then handled by our Linux driver *vncsimpleinput*, which provides the evdev interface. The full stack can be seen in the architecture diagram in Figure 1.

To integrate X.Org in our setup, we simply enable support for it in the Buildroot configuration included in *GUI-VP Kit*. The applications used in the following demonstrations are also selected from the wide range of X applications available in Buildroot.

X.Org Standard Applications: A screenshot of the VNC client connected via network to *GUI-VP* is depicted in Figure 3. It shows a running X.Org server with the lightweight *fluxbox* window manager on top. We have started well-known X applications, from top to bottom and left to right: *xlogo*, *xclock*, *xcalc*, *xeyes* and *xev*. On our machine it takes about 170 seconds from startup of the *GUI-VP* to a fully functional state. Until this point, the RV64 ISS of *GUI-VP* executed approximately 4.1 billion RISC-V instructions. The interaction with the running X applications is smooth and responsive.

X.Org Web Browser Application: The respective screenshot is shown in Figure 4. Again it shows a running X.Org server. However, in this case the lightweight web browser *dillo* is started directly without a window manager. The web browser shows a website hosted locally on the simulated system. On our machine it takes about 140 seconds from startup of the VP to a fully functional state. Until this point, the RV64 ISS of *GUI-VP* executed approximately 3.3 billion RISC-V instructions.

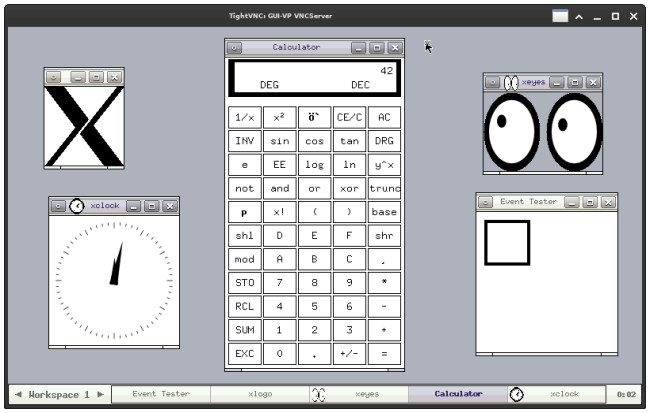


Figure 3: X.Org with Window Manager and Applications

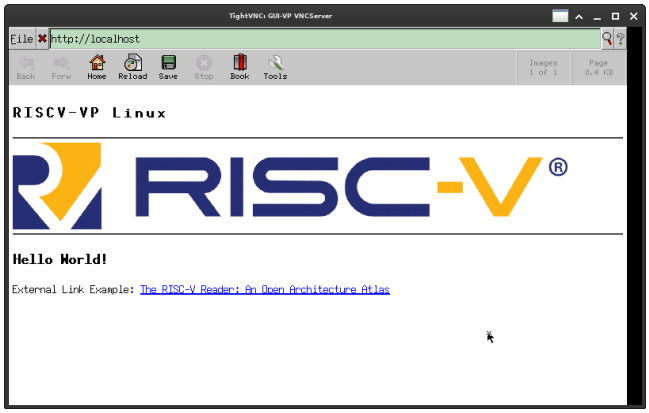


Figure 4: X.Org with Web Browser

6.2 *GUI-VP Kit* Performance: PrBoom

In this section, we investigate the performance of *GUI-VP Kit* using the SDL-based classic first-person 3D-game *PrBoom*. A screenshot of the VNC client connected to *GUI-VP* via network, showing the graphical output of *PrBoom* is given in Figure 5.

In our experiments, we examine the average frame rate, measured in FPS, achieved by *PrBoom* in different setups described below. For the measurements, we modify *PrBoom* so that it calculates the average FPS. To make our experiments reproducible and thus obtain comparable data from multiple runs, we use the non-interactive demo mode of *PrBoom*. This mode is started automatically when no user interacts with the game and runs several demos one after another. As final result for each experiment, we take the average FPS after the first demo is finished.

The results obtained from the experiments on our machine are summarized in Figure 6. The Y-axis shows the average FPS achieved

Figure 5: PrBoom on RV32 Single-Core (640x480)

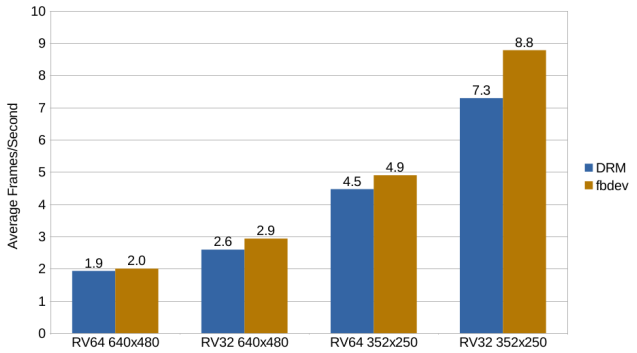


Figure 6: Average FPS achieved with PrBoom on *GUI-VP*

in a setup. The X-axis shows the different setups, as combinations of the *GUI-VP* model (RV32 or RV64) and the game resolution (640x480 or 352x250). The orange bars show the frame rate when using DRM. The blue bars shows the frame rate when using fbdev.

The achieved average FPS range between 1.9 and 8.8. Quite intuitively, a lower resolution results in higher FPS. Furthermore, the compact and simple fbdev interface offers higher performance than the more modern, but also more complex DRM interface. The reason for the difference in performance between RV32 and RV64 is not so obvious and will be investigated further in future work.

Finally, to interpret the collected FPS values in terms of performance, we have to take into account that the performance of 3D games does not only depend on the graphics output. Since 3D scenes change a lot between frames, a high degree of recalculation is necessary for each frame. In addition, the game logic, including all actors, must be executed in sync with the scene. To put this in perspective: On our *GUI-VP* ISS, we can observe up to 13.7 million RISC-V instructions executed *per frame*. Considering this, the achieved FPS on *GUI-VP* are in a reasonable range.

7 CONCLUSIONS

In this paper we presented *GUI-VP Kit*, a full simulation environment that enables the development of interactive graphical Linux applications early in a system design process. *GUI-VP Kit* comes with (i) *GUI-VP*, a RISC-V VP capable of running Linux that was greatly extended to support graphics output and mouse input, which is made available to the user via VNC, and (ii) all necessary configurations to generate a fully functional Linux environment, including all necessary drivers to handle the new peripherals, and the example frameworks and applications.

The capability of the *GUI-VP Kit* simulation stack to run generic Linux graphics frameworks was demonstrated by two examples based on the X.Org Window System. We have shown that *GUI-VP* is able to boot a Linux system with an X server and X applications in less than 3 minutes. The performance of *GUI-VP Kit* was evaluated using PrBoom, a Linux port of a classic 3D-game. In terms of concrete results, we showed that with up to 8.8 FPS the performance is reasonable considering the complex calculations (up to 13.7 million RISC-V instructions) required for each frame.

GUI-VP Kit and all experiments are available as open-source on GitHub.

In the future, we plan to further improve *GUI-VP*'s simulation performance and extend *GUI-VP Kit* with additional features useful for developing graphical applications, such as keyboard input, persistent storage capabilities, and 3D graphics.

ACKNOWLEDGMENTS

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

REFERENCES

[1] Accessed: 2023-02-19. http://systemc-verification.org.
[2] Accessed: 2023-02-19. Buildroot. https://www.buildroot.org.
[3] Accessed: 2023-02-19. DBT-RISE. https://github.com/Minres/DBT-RISE-Core.
[4] Accessed: 2023-02-19. libVNCServer. https://libvnc.github.io.
[5] Accessed: 2023-02-19. PrBoom. https://prboom.sourceforge.net/.
[6] Accessed: 2023-02-19. RISC-V Virtual Prototype. https://github.com/agra-uni-bremen/riscv-vp.
[7] Accessed: 2023-02-19. RISCV-QEMU. https://github.com/riscv/riscv-qemu.
[8] Accessed: 2023-02-19. SDL. https://www.libsdl.org/.
[9] Accessed: 2023-02-19. X.Org. https://www.x.org.
[10] Amir Charif, Gabriel Busnot, Rania Mameesh, Tanguy Sassolas, and Nicolas Ventroux. 2019. Fast Virtual Prototyping for Embedded Computing Systems Design and Exploration. In *Proceedings of the Rapid Simulation and Performance Evaluation: Methods and Tools*. 3:1–3:8.
[11] Tom De Schutter. 2014. *Better Software. Faster!: Best Practices in Virtual Prototyping*. Synopsys Press.
[12] Daniel Große and Rolf Drechsler. 2010. *Quality-Driven SystemC Design*. Springer.
[13] Muhammad Hassan, Daniel Große, and Rolf Drechsler. 2022. *Enhanced Virtual Prototyping for Heterogeneous Systems*. Springer.
[14] Vladimir Herdt, Daniel Große, and Rolf Drechsler. 2020. *Enhanced Virtual Prototyping: Featuring RISC-V Case Studies*. Springer.
[15] Vladimir Herdt, Daniel Große, Hoang M. Le, and Rolf Drechsler. 2018. Extensible and Configurable RISC-V based Virtual Prototype. In *Forum on Specification and Design Languages*. 5–16.
[16] Vladimir Herdt, Daniel Große, Pascal Pieper, and Rolf Drechsler. 2020. RISC-V based Virtual Prototype: An Extensible and Configurable Platform for the System-level. *Journal of Systems Architecture - Embedded Software Design* 109 (2020), 101756.
[17] IEEE Std. 1666 2011. *IEEE Standard SystemC Language Reference Manual*. IEEE Std. 1666.
[18] Rainer Leupers, Grant Martin, Roman Plyaskin, Andreas Herkersdorf, Frank Schirrmeister, Tim Kogel, and Martin Vaupel. 2012. Virtual platforms: Breaking new grounds. In *Design, Automation and Test in Europe*. 685–690.
[19] Màrius Montón. 2020. A RISC-V SystemC-TLM simulator. arXiv:2010.10119
[20] OSCI 2009. *OSCI TLM-2.0 Language Reference Manual*. OSCI.
[21] Andrew Waterman and Krste Asanović. 2019. *The RISC-V Instruction Set Manual; Volume I: Unprivileged ISA*. SiFive Inc. and UC Berkeley.
[22] Andrew Waterman and Krste Asanović. 2019. *The RISC-V Instruction Set Manual; Volume II: Privileged Architecture*. SiFive Inc. and UC Berkeley.