Accelerating Bare Metal Driver Development with Linux Drivers and System Verilog DPI-C.

Suchir Gupta	Amit Sharma	Suneetha Suryadevara	Vishnuvardhan Reddy
Synopsys Inc	Synopsys Inc	Synopsys Inc	Synopsys Inc
suchiir@synopsys.com	amits@synopsys.com	suryas@synopsys.com	vishnuvr@synopsys.com
Sunnyvale, CA, USA	Sunnyvale, CA, USA	Sunnyvale, CA, USA	Sunnyvale, CA, USA

***Abstract* – Developing bare metal device drivers—software running directly on hardware without an operating system—is critical yet time-consuming for IP verification, system-level validation, and embedded development. Although mature Linux drivers exist for virtually all peripheral hardware, their tight integration with the operating system prevents their direct use in bare metal environments. This paper presents a systematic framework that leverages System Verilog DPI-C to convert Linux drivers into functionally equivalent bare metal drivers. Our approach introduces three key innovations: (1) hardware abstraction wrappers with SV DPI-C interfaces replacing kernel services, (2) expert-guided driver transformation preserving core device logic, and (3) comprehensive validation in simulation and emulation environments ensuring zero functional regressions. We validated this framework through implementation of a USB4 bare metal driver (connection manager), tested through Host and Device router enumeration and NVM operations in both VCS simulation and Zebu emulation environments, demonstrating an 83% reduction in development time (from 6-8 months to 1 month) while reusing 39,000 lines of verified Linux code. While this approach streamlines driver conversion, adaptation may be required for drivers with complex kernel dependencies or real-time constraints. This methodology can be applied to convert any Linux device driver to a bare metal driver, significantly accelerating development across various hardware platforms including hardware/software co-verification, pre-silicon debug, SoC bring-up, emulation, and virtual prototyping environments.**

I. Introduction

Bare metal drivers are essential software components that configure, control, generate stimulus, and respond to requests from hardware IP without operating system support. These drivers play a crucial role in IP verification environments, but their development cycles tend to be long, with limited opportunities for code reuse and extensive testing required to ensure IP specification compliance.

In contrast, the Linux kernel provides mature, thoroughly tested drivers for virtually every hardware peripheral—reflecting years of development, optimization, and bug fixes. However, these drivers are tightly integrated with the Linux infrastructure and cannot be used directly in bare metal environments because of their reliance on kernel services.

Direct hardware control is essential in bare metal IP verification environments, where the overhead of the Linux kernel is prohibitive. Our innovative framework systematically separates device-specific logic from OS dependencies, making it possible to reuse proven Linux driver logic in bare metal environments. By leveraging the robustness of the Linux driver ecosystem, our framework ensures functional equivalence and maintains the performance characteristics required for verification environments.

Using this framework, we developed a bare metal driver (connection manager) for USB4 transactor software, enabling configuration and control of USB4 transactor hardware. The USB4 transactor operates successfully in both simulation and emulation domains. This approach can be applied to convert any Linux-based IP driver into a bare metal IP-based driver.

II. Background And Related Work

A. *Linux Driver Architecture:*

Understanding Linux driver interactions with peripheral hardware is crucial for appreciating the complexity of converting these drivers to bare metal environments [2]. In the Linux ecosystem, driver-device interaction involves multiple system components:

1. **Device Discovery:** The Linux kernel scans PCIe bus during boot and discovers devices by their vendor and device identifiers through configuration space reads.
2. **Driver Binding:** The Linux kernel matches discovered devices with registered drivers through the device model subsystem, invoking the `probe()` function.
3. **Resource Management:** Device drivers call `pci_enable_device()`, `pci_request_regions`() to allocate hardware resources.

4. **Memory Mapping:** The `ioremap()` function creates virtual memory mappings to physical device registers, enabling register access via standardized `ioread32()` and `iowrite32()` interfaces.
5. **Interrupt Handling:** The `request_irq()` function registers interrupt handlers with the kernel's IRQ subsystem, establishing the connection between hardware interrupts and software handlers.

Figure 1 showcases how Linux framework connects device driver with devices. For any memory write or read operations, the device driver invokes `iowrite()/ioread()` calls transmitting the data from kernel to device via CPU, PCIE Root Complex (RC), and PCIE Endpoint (EP) components. For interrupt handling, the device driver receives the interrupt from device through the reverse path: PCIe EP, PCIE RC, CPU, and eventually kernel framework. The components in orange boxes are on emulator, while the components in blue boxes are on Host PC.

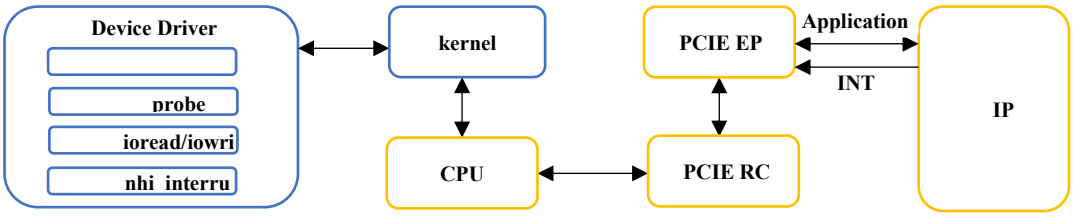

Figure 1. Device driver interaction with devices in Linux eco system.

B. Challenges in Bare Metal Driver Development:

The tight coupling between Linux driver and kernel infrastructure makes direct bare metal reuse challenging. The following critical components are missing in bare metal environments (simulation, emulation, embedded systems):

- **No Linux Kernel:** No device enumeration, memory management, or interrupt routing frameworks.
- **No PCIe Infrastructure:** No PCIe root complex, endpoint discovery, or configuration space access
- **No System Services:** No virtual memory management, interruption handling, or resource allocation

This infrastructure gap explains why Linux kernel framework cannot be used directly in bare metal verification environment, and underscores significance of our SV DPI-C bridge approach.

C. Related Work

SV DPI-C has been used in various verification contexts to bridge hardware and software domains. Prior work has demonstrated its effectiveness for testbench automation and verification component development. However, to our knowledge, no prior work has systematically leveraged SV DPI-C to enable direct reuse of Linux kernel drivers in bare metal verification environments.

Driver portability has been addressed through hardware abstraction layers in embedded systems, but these solutions typically target different operating systems rather than bare metal environments. Our work is distinct in enabling direct Linux drivers to reuse without requiring a full OS implementation.

While Universal Verification Methodology (UVM)-based verification environments often employ Register Abstraction Layer (RAL) models for automated driver development, our framework targets bare metal environments where UVM infrastructure is unavailable—including emulation platforms, virtual prototyping, and embedded systems. Our SV DPI-C approach provides a lightweight alternative that does not depend on UVM or RAL, making it broadly applicable across diverse verification scenarios.

D. Transactor Architecture

The bare metal driver developed is an integral part of the USB4 transactor solution. This section offers an overview of a typical transactor's architecture, which is crucial for understanding the methodology behind bare metal driver development.

A transactor has two main components: hardware and software. The hardware portion is implemented in Synthesizable System Verilog (SV), while the software is written in C/C++. Within the hardware wrapper, you'll find the IP, the logic necessary to interact with the IP, and SV DPI-C [1], used to communicate with the corresponding software. The software side includes the device driver that follows the software IP specification to control the IP, user APIs for managing the transactor, and extern "C" functions for interacting with the hardware wrapper. Utilizing SV DPI-C allows effective communication between software and hardware, ensuring the transactor operates smoothly in both simulation and emulation environments.

Figure 2 presents the transactor's topology. The components in orange boxes are on emulator, while the components in blue and green boxes are on Host PC. The testbench calls user-level APIs to configure the

transactor and register user-level C callback functions. These user APIs connect with the IP driver to manage and configure the IP. When the IP driver handles memory transactions, it uses extern "C" functions to communicate with the IP via the SV-DPI-C interface. If the IP generates events, it triggers an interrupt that reaches the IP driver through the SV-DPI-C interface. Once received, the IP driver either processes the interrupt on its own or invokes user callbacks to gather responses from the test bench.

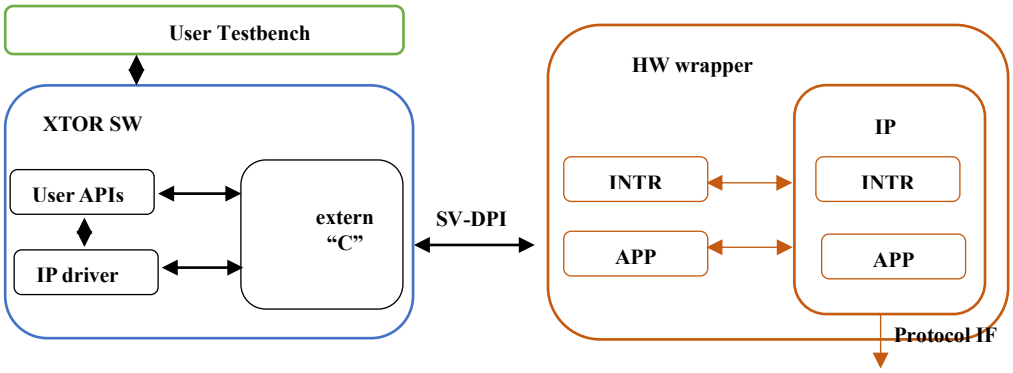

Figure 2. Transactor topology.

III. Framework and Methodology for Linux to Bare Metal Driver Conversion

This section explains the systematic process for converting Linux device drivers to bare metal drivers. The framework consists of four interconnected components and a structured transformation methodology.

A. *System Architecture:*

Our framework replaces the missing Linux infrastructure with four interconnected components that recreate the Linux driver environment in bare metal environment:

1. **Kernel Proxy:** A C++ class that emulates Linux kernel services, providing the same interfaces (memory allocation, device mapping, interrupt handling) but implemented for bare metal environments. This component acts as a translation layer between the driver's expectations and bare metal reality.
2. **SV DPI-C Interface:** This bridge connects software calls to hardware registers and routes hardware interrupts back to software. It provides the bidirectional communication channel essential for driver-hardware interaction without OS mediation.
3. **Hardware Wrapper:** A System Verilog module that translates SV DPI-C calls into protocol-specific transactions (AXI/APB) and delivers them to the IP. This module also handles interrupt generation and delivery to the software layer.
4. **Modified Device Driver:** The Linux device driver with necessary modifications to remove OS dependencies while preserving all device interaction logic. The core driver algorithms remain unchanged, ensuring functional equivalence.

B. *Device driver transformation methodology:*

1. **Updated device driver:**
    a. Download the linux code from www.linux.org. Create a dedicated workspace and copy the relevant driver directory to this workspace.
    b. Investigate the device driver to identify all kernel dependencies including memory management calls, PCI subsystem interactions, interrupt handling, and other kernel services. Since PCIe infrastructure and Linux kernel code are not required in bare metal environments, comment out these calls or create stub functions within the kernel proxy class. This includes removing all Linux library header file dependencies.
    c. Add comprehensive debug messages throughout the driver to enable troubleshooting and validation during the conversion process.
2. **Kernel Proxy:**
The kernel proxy is a C++ class that provides alternate definitions for Linux kernel services, achieving equivalent functionality without OS support:
    a. **Device mapping:** During initial boot, the Linux kernel binds the device driver to hardware by calling the probe function. The probe function enables the PCI device, establishes memory maps for register access, and registers interrupt handlers with the kernel. Since the Linux kernel is unavailable, create a user-level initialization API that invokes the probe function. This probe function creates all

necessary data structures for driver operation. To locate the probe function, examine the `pci_driver` structure definition in the driver source code.

b. **Memory writes/reads:** Linux device drivers invoke `iowrite()/ioread()` functions for memory operations with devices. The kernel infrastructure routes data to the device through this standardized interface, traversing the CPU, PCIe RC, PCIe EP, and protocol translation layers. Since Linux infrastructure is unavailable, create alternate definitions of `iowrite()/ioread()` in the kernel proxy. Driver calls to these functions are redirected to the proxy implementations, which invoke extern "C" functions that transport data to the IP via SV DPI-C.

c. **Interrupt handling:** In Linux, devices assert interrupt lines that are delivered to drivers through the kernel framework. Interrupts propagate from the device through PCIe EP, PCIe RC, and CPU to reach the kernel, which then dispatches them to registered driver handlers. For bare metal operation, create alternate interrupt handling mechanisms. The IP passes interrupt information to software via SV DPI-C. The software delivers interrupt information to the kernel proxy, which invokes the appropriate interrupt handler in the Linux device driver. Interrupt handler functions can typically be identified by examining functions with the `_msi` suffix in the driver source.

d. **Alternate macro definition:** Linux drivers use numerous macros and functions defined in kernel headers. Since these libraries are unavailable in bare metal environments, provide alternate definitions for all required macros and functions in the kernel proxy.

3. **Hardware Wrapper design:** Since bare metal software requires direct hardware register access and interrupt handling without Linux kernel and PCIe framework support, develop a System Verilog hardware wrapper module. This wrapper encapsulates the IP, implements IP-specific protocol logic, and provides software-accessible interfaces, ensuring seamless interaction between hardware and bare metal software. The wrapper replaces the CPU, PCIe RC, PCIe EP, and protocol translator components present in the Linux framework.
   A. Develop protocol-specific interfaces (AXI/APB/etc.) that interact with the IP's register interface.
   B. Develop interrupt handler logic in the hardware wrapper that detects when the IP triggers interrupts.
   C. Implement SV DPI-C methods in the hardware wrapper that transport protocol transactions and interrupt data bidirectionally between hardware and software domains.

This systematic approach ensures drivers are efficiently adapted for bare metal operation, eliminating Linux kernel and PCIe infrastructure dependencies while providing direct, efficient hardware resource access.

IV. USB4 BARE METAL DRIVER DEVELOPMENT CASE STUDY

This section explains how we used the conversion methodology to create a USB4 bare metal device driver for the USB4 router in the transactor solution, outlining the changes from the Thunderbolt Linux driver to the USB4 bare metal version [4].

A. Driver Modification process:

We downloaded the Linux kernel source code from `www.kernel.org` and created a dedicated workspace. We copied the Thunderbolt driver from `<linux_source_directory>/drivers/thunderbolt` to the workspace. After investigating the driver structure, we identified all memory management, PCI subsystem, interrupt handling, device management, and synchronization calls. Since PCIe infrastructure and Linux kernel services are not required for bare metal environments, we systematically commented on calls in the workspace. We added debug messages throughout the driver to facilitate debugging and validation.

Figure 3 illustrates modifications to the `nhi.c` file to create the updated USB4 device driver. Key changes include removal of PCI initialization code, replacement of kernel memory allocation with bare metal equivalents, and modification of interrupt registration logic.

```
#ifdef INCLUDE_PCIE_COMMENTED_CODE
 #include <linux/slab.h>
 #include <linux/errno.h>
#endif

int nhi_init(void) {
 ...
 dev_dbg(2, "Entered ... ");
#ifdef INCLUDE_PCIE_COMMENTED_CODE
      ret = pci_register_driver(&nhi_driver);
#endif
 ...
}
```

Figure 3. Modification of nhi.c file to created updated USB4 device driver.

B. Kernel Proxy Implementation:

1. **Device mapping:** We investigated the `pci_driver` structure definition in the Thunderbolt driver workspace. For USB4, when the Linux kernel calls `probe()`, the `nhi_probe()` function is invoked in the device driver. Figure 4 illustrates the overall flow for device mapping. The user calls the transactor-defined `init()` function from the testbench. The `init()` function in the kernel proxy calls `nhi_probe()` in the device driver, creating all necessary data structures to interact with the USB4 device.

```
//User testbench calling init()
nhi_initialize((void*)this);

//struct pci_driver explaining which functions in device driver would be called when Linux kernel calls probe functions.
static struct pci_driver nhi_driver = {
 .name       = "thunderbolt",
 .id_table   = nhi_ids,
 .probe      = nhi_probe,
 .remove     = nhi_remove,
 .shutdown   = nhi_remove,
 .driver.pm = &nhi_pm_ops,
};

//Proxy kernel calling device driver probe function.
void proxy_kernel::init( void *params ) {
  ...
  nhi_probe(x, dev,1); //Call device driver probe function.
  ...
}
```

Figure 4. User testbench calling device driver probe() function rather than Linux kernel.

2. **Memory writes/reads:**
For USB4, we created alternate definitions for `iowrite()/ioread()` in the kernel proxy. The proxy kernel invokes extern "C" functions that transport data to the USB4 IP via SV DPI-C. Figure 5 demonstrates how the USB4 Thunderbolt device driver's memory write through `iowrite32()` reaches the IP through SV DPI-C, bypassing the Linux framework entirely. Similar flow exists for `ioread32()` calls.

```
//Linux device driver performing write and read operations.
static void nhi_mask_interrupt(struct tb_nhi *nhi, int mask, int ring) {
 if (nhi->quirks & QUIRK_AUTO_CLEAR_INT)
   iowrite32(nhi,val & ~mask, nhi->iobase + REG_RING_INTERRUPT_BASE + ring);
 else
   iowrite32(nhi,mask, nhi->iobase + REG_RING_INTERRUPT_MASK_BASE + ring);
}

//DPI-C taking the information further to IP.
extern "C" void mstr_w_data(int w , svBitVecVal * addr);

//proxy kernel redefining and passing the information to DPI-C
void iowrite32(void * lx, uint32_t value, uint32_t addr) {
    mstr_w_data(value, addr);
}
```

Figure 5. Device driver performing memory write through proxy kernel avoiding Linux kernel framework.

3. **Interrupt handling:**
After investigating the Thunderbolt device driver, we identified `nhi_msi()` as the interrupt handler function. Figure 6 illustrates the overall flow for interrupt handling in the bare metal verification environment. The USB4 IP generates an interrupt which reaches the kernel proxy through SV DPI-C. The proxy kernel's interrupt function calls `nhi_msi()`, which triggers the interrupt processing logic in the Thunderbolt driver.

```
//extern C function getting the notification from IP about the interrupt, and passing to proxy_kernel.
extern "C" void interrupt()
  proxy_kernel->interrupt();

//Proxy kernel calling interrupt in Linux device driver.
void proxy_kernel::interrupt()
  nhi_msi(1, );
```

Figure 6. IP passing interrupt information to proxy kernel which calls interrupt function in Linux device driver.

4. **Alternate macro definition:** We replaced several Linux library definitions that were used by the Thunderbolt device driver. Figure 7 provides examples of alternate definitions for functions and macros that were used in the device driver but whose definitions were present in Linux libraries.

```
//Kernel macro redefinition -- devm_kcalloc
#define devm_kcalloc(x,n, size, type) calloc(n,size)

//Linux kernel function cpu_to_be32_array redefined.
void cpu_to_be32_array(uint32_t *dst, uint32_t * src, size_t size) {
    for (size_t i = 0; i < size; ++i)
      dst[i] = htobe32(src[i]);
}

//Linux kernel function ida_free redefined.
void ida_free(struct ida * ida, unsigned int id) {
  free(ida);
}
```

Figure 7. Alternate definitions for functions and macros which were used in device driver, but definitions were present in Linux library.

C. *H/W Wrapper Implementation:*

For the USB4 router, we implemented the following components in the hardware wrapper:

a. **Protocol Interface:** We developed AXI/APB interface implementations in the hardware wrapper to communicate with the USB4 router's register interface.
b. **Interrupt Handler:** We developed interrupt detection and propagation logic that is invoked when the IP triggers an interrupt.
c. **SV DPI-C Interface:** We developed SV DPI-C methods that transport AXI/APB transaction data and interrupt information bidirectionally between hardware and software domains.

```
module hw_wrapper (
);
  // DPI-C functions determined by dependency analysis
import "DPI-C" task interrupt;
export "DPI-C" task mstr_w_data(addr, value);

//DPI function will be invoked by H/W. This function will translate the //data to AXI transaction, and present to IP.
function void mstr_w_data(reg[31:0] addr, reg [1:0]data);
…
endfunction

//This logic will interpret the interrupt, and then notify the S/W through SV-DPI import call.
always @(posedge clk) begin
  if (|intr)
    __interrupt();
end
xtor_usb4_svs usb (
  .intr (intr)
  .axi* (axi*)
  .usb4* (usb*)
  );
endmodule
```

Figure 8. Code snippets of USB4 hardware wrapper.

D. Implementation Considerations:

The driver conversion process revealed several practical insights. Identifying the minimal kernel API set and commenting out Linux and PCIe-related code required approximately 2 weeks of analysis, covering around 150 kernel functions across memory management, device I/O, interrupt handling, and synchronization primitives.

Regarding simulation performance, we lacked a direct comparison benchmark, though we observed that pure UVM-based setups were slower due to System Verilog and UVM framework overhead. Our DPI-C approach provided better performance in this context. Memory transactions via DPI calls were frequent during normal operation, representing the primary performance consideration. However, since data was transferred as complete transactions rather than individual signals, we did not encounter issues with 4-state value handling across the DPI boundary.

We established coding guidelines that include explicit parameter validation and error handling in all DPI-C functions, timeout mechanisms to prevent simulation hangs, atomic register access to avoid race conditions, and a prohibition on global variables across the DPI boundary to ensure reentrancy. Our wrapper-based DPI approach provided sufficient abstraction and debugging capability without requiring pure DPI calls.

Figure 9 illustrates overall topology for Linux driver conversion to bare metal driver with SV DPI-C interface. The components in orange boxes are on emulator, while the components in blue boxes are on Host PC.

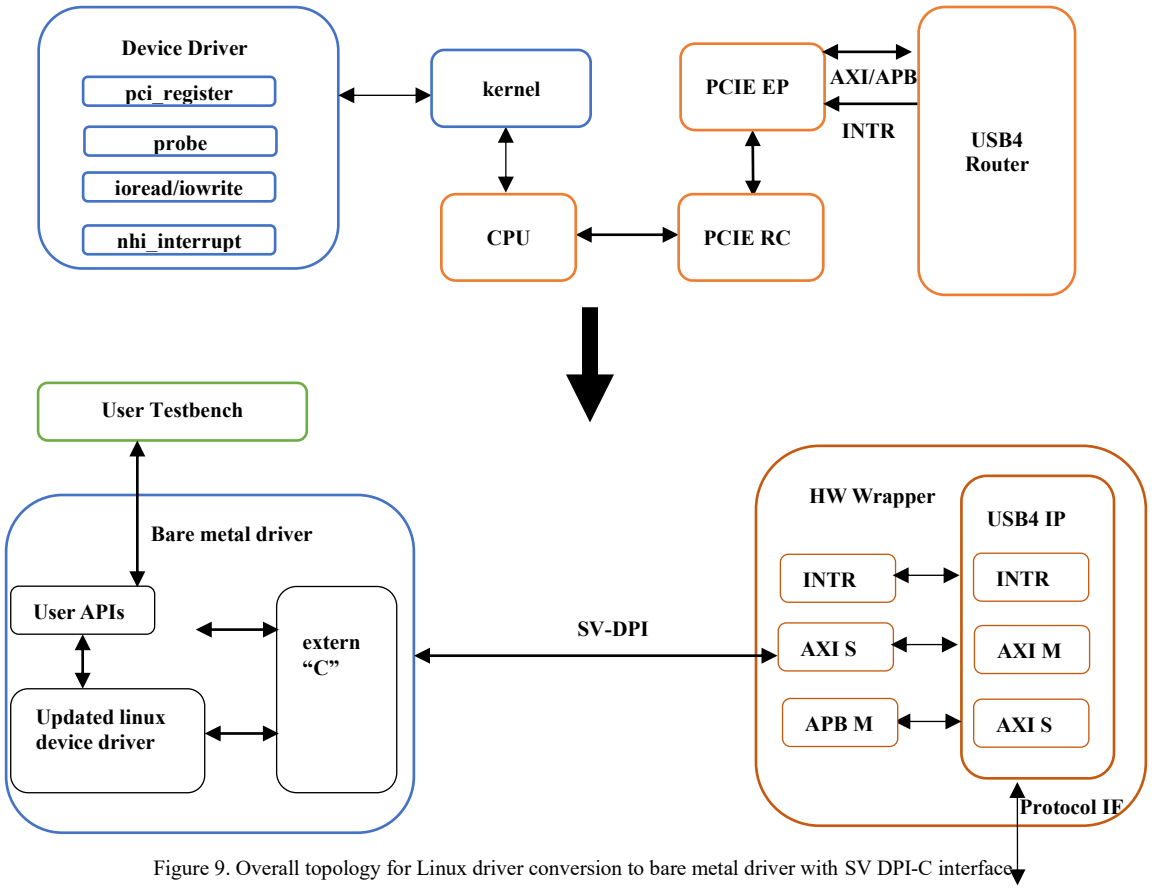

Figure 9. Overall topology for Linux driver conversion to bare metal driver with SV DPI-C interface

IV. Validation Methodology and Results

A. *Validation Approach:*

To ensure functional equivalence between the Linux driver and converted bare metal driver, we implemented a comprehensive validation methodology. We developed test cases covering initialization, data transfer, error handling, and power management. Specific validation scenarios included enumerating Host and Device routers through the connection manager and performing NVM writes and reads at 4Gb/s in both simulation and emulation domains. All functional tests passed successfully with zero defects detected, confirming functional equivalence with the reference Linux driver implementation.

Integration Testing: We verified end-to-end data paths including initialization sequences, data transfer operations, and error handling scenarios.

B. *Quantitative Results:*

Our framework achieved measurable benefits in the USB4 driver conversion case study. Table 1 summarizes the quantitative results.

Metric	Traditional Approach	Our Framework
Development Time	6-8 months	1 month
Time Reduction	Baseline	83%
Code Reused	0 lines	~39,000 lines
Defects Found	N/A	0

Table 1: Quantitative comparison of traditional bare metal driver development versus our framework

The results demonstrate an 83% reduction in development time, from the typical 6-8 months required for traditional bare metal driver development to approximately 1 month using our framework. This dramatic improvement stems from reusing approximately 39,000 lines of verified Linux driver code. All functional tests

passed successfully with zero defects detected, confirming functional equivalence between the Linux driver and converted bare metal driver.

V. Conclusion

This work presents a practical framework that bridges the gap between Linux driver infrastructure and bare metal verification environments through systematic use of System Verilog DPI-C interfaces. The USB4 case study validates our approach, demonstrating successful conversion of a complex device driver with extensive kernel dependencies. The framework achieved an 83% reduction in development time while maintaining complete functional equivalence between the original Linux driver and the bare metal version, confirming the robustness of the conversion process.

The framework's efficiency stems from reusing approximately 39,000 lines of verified Linux driver code. During validation, all functional tests passed successfully with zero defects identified, ensuring that the converted bare metal driver maintained both efficiency and reliability. This systematic methodology is highly versatile and adaptable to various hardware platforms, allowing it to be applied to the conversion of any Linux device driver to a bare metal driver across diverse development environments.

The bare metal drivers produced using this framework support multiple critical use cases. They can be utilized for configuring and managing IP in both simulation and emulation environments, providing robust support for pre-silicon validation and system-level verification, thereby enhancing the reliability of hardware designs before they reach production. The framework extends to virtual prototyping environments, enabling early software development and testing, allowing teams to address potential issues and optimize performance in advance of hardware availability. This capability accelerates hardware verification workflows by bridging the gap between mature Linux drivers and bare metal verification requirements, ensuring functional correctness through effective code reuse.

The broad applicability of the framework highlights its significant impact in expediting the development of bare metal drivers across various platforms and use cases. It offers a strategic advantage for hardware verification and early software development, making it a key enabler in modern hardware design cycles.

References

[1] IEEE Standard Association, “IEEE Standard for SystemVerilog - Unified Hardware Design, Specification, and Verification Language,” IEEE Standard 1800-2017, Dec. 2017 Available: https://standards.ieee.org

[2] Corbet, J., Rubini, A., & Kroah-Hartman, G. (2005). Linux Device Drivers, Third Edition. O'Reilly Media.

[3] USB Implementers Forum, "USB4 Specification Version 2.0," Sept. 2022. Available: https://www.usb.org/document-library/usb4r-specification-v20

[4] Intel Corporation, "Thunderbolt Technology Brief," 2020. Available: https://www.intel.com/content/www/us/en/io/thunderbolt/thunderbolt-technology-brief.html/[7