ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-MOPHA156

THE LINUX DEVICE DRIVER FRAMEWORK FOR HIGH-THROUGHPUT LOSSLESS DATA STREAMING APPLICATIONS*

K. Vodopivec†, E. Breeding, J. Sinclair, Oak Ridge National Laboratory, Oak Ridge, USA

Abstract

Many applications in experimental physics facilities require custom hardware solutions to control process parameters or to acquire data at high rates with high integrity. These hardware solutions typically require custom software implementations. The neutron scattering detectors at the Spallation Neutron Source at Oak Ridge National Laboratory transfer custom protocols over optical fiber connected to a PCI Express (PCIe) read-out board. A dedicated kernel-mode device driver interfaces the PCIe read-out board to the software application. The device driver must be able to sustain data bursts from a pulsed source while acquiring data for long periods of time. The same optical channel is also used as bi-directional low-latency communication link to detector electronics for configuration, real time health monitoring and fault detection. This article presents a Linux device driver design, implementation challenges in a low-latency high-throughput setup, kernel driver optimization techniques, real use case benchmarks and discusses the importance of clean application programming interface for seamless integration in control systems. This generic framework has been extended beyond neutron data acquisition, thus, making it suitable for new and diverse applications as well as rapid development of field programmable gate array (FPGA) firmware.

INTRODUCTION

In this article we present a low-level software framework that facilitates rapid PCIe-based FPGA firmware development and fast integration into control systems such as Experimental Physics and Industrical Control System (EPICS). At the Spallation Neutron Source (SNS), custom hardware and electronics are commonly developed to implement functionality that is not commercially available. Throughout the design and implementation, FPGA firmware interfaces and functionality are subject to change, and thus, software is needed to drive test cases and verify functionality. When firmware development is completed, new equipment needs to be integrated into the control system for long term operations. Software must provide a means to control firmware parameters, monitor functionality and read out data at very high sustained data rates. Our software framework, as shown in Figure 1, consists of a Linux kernel device driver for tight integration with PCIe devices, the user space library providing powerful but elegant application programming interfaces and several generic tools for swift prototyping and verification.

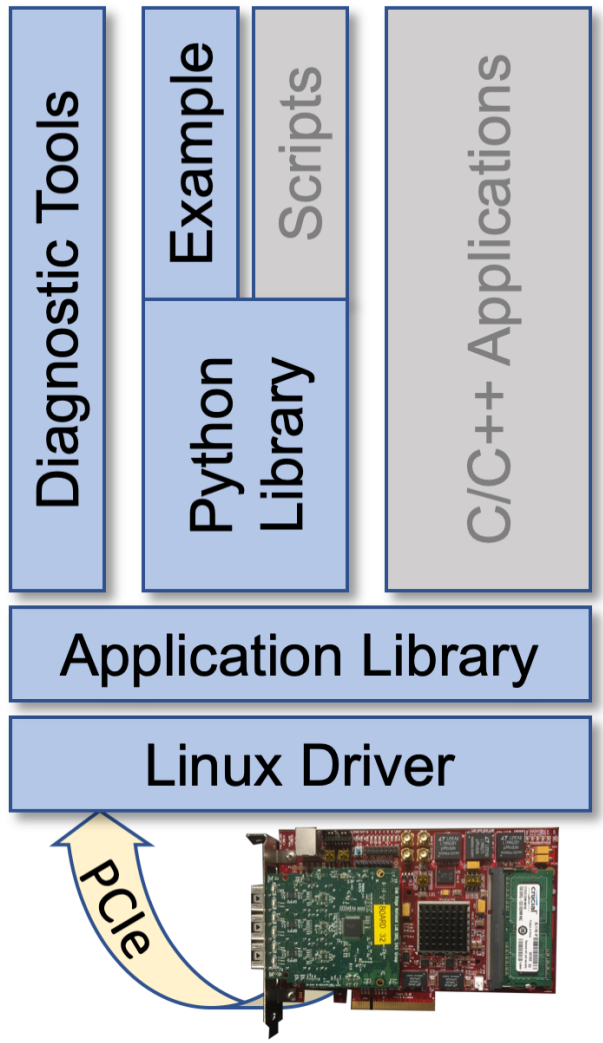

Figure 1: Device driver framework architecture.

LINUX DEVICE DRIVER

Detector data from SNS neutron scattering instruments are sent as packets over optical fiber to multiple PCIe read-out boards hosted in a Linux server. The read-out board uses a Xilinx FPGA to handle both the PCIe interface and the packet transfers with the detector electronics. The theoretical

* This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan(http://energy.gov/downloads/doe-public-access-plan).
† vodopiveck@ornl.gov

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-MOPHA156

maximum data rate on the optical channel is 212.5 MB/s; this rate is established by the 2.5 Gbps line rate and 8b/10b encoding of the physical layer device used by the detector electronics. Due to limited memory resources on the detector electronics, packet sizes are capped by design to be ≤ 10kB. Small packet sizes coupled with a sustained rate of 200MB/s results in an extremely high interrupt rate. In the Linux environment, this means implementing a kernel device driver with direct access to firmware resources.

FPGA firmware verifies and transfers incoming data packets to the CPU using direct memory access (DMA) to contiguous system memory. Physical memory allocation is performed by the driver, and the driver communicates both the starting address and size of this memory to the firmware. This contiguous memory is managed as a circular buffer. Producer and consumer indices protect from overwriting data yet to be processed by software. It is thus evident that allocated memory must be big enough to accommodate processing scheduling latencies and sustain preemption fluctuations.

Data Transfers

Depending on the selected memory allocator, the Linux kernel generally doesn't allow the allocation of large blocks of physical memory. The upper limit is a function of both memory page size – driven by CPU architecture – and the Linux kernel compile-time option called MAX_ORDER. On Intel x86 platforms, this upper limit is 4MB; however, this is the upper limit which degrades over time as memory allocations of various sizes can fragment the available memory pool. Furthermore, the 4MB buffer, even when allocated early in the boot process, is not adequate for the rates required by some of the instruments at SNS. To achieve larger buffer sizes, our driver uses a technique commonly known as memory reservation. This method reserves physical memory before the Linux kernel boots, thus making the memory invisible to the Linux kernel. Knowing the reserved memory physical address range, the device driver can use and manage reserved memory for its private operations. Memory reservation can be requested through the *memmap* kernel boot-time parameter and allows the reservation of arbitrary sizes contingent on installed memory. While this technique works well with Linux kernels 2.6 and above, it is considered fragile by the Linux community. Starting with Linux 3.6 (back ported to 3.4), the new integrated functionality called Contiguous Memory Allocator (CMA) is available. Like *memmap* memory reservation, CMA blocks off parts of memory at boot time, but it later works with the Linux memory allocator to allow the use of reserved memory for arbitrary allocations, until there's a request for a large memory block. This allows for better memory utilization in dynamic environments but has no significant impact in highly specialized applications where large memory blocks are in constant use.

Interrupt Handling

While it is common practice for DMA engines to generate an interrupt per received packet, this is not sustainable in this scenario given the required throughput and small packet sizes. Local measurements on a high-end server show that the Linux operating system can handle tens of thousands of interrupts per second; however, with growing numbers, the interrupt latency increases and becomes non-deterministic. Each interrupt involves pre-empting currently running tasks, switching the CPU context, and potentially elevating to privileged mode. All these operations cost CPU cycles and consequently have a negative impact on overall system performance. Most high throughput devices (like gigabit Network Interface Cards or RAID controllers) combine interrupt requests into groups and assert one interrupt per group. This technique is commonly referred to as interrupt coalescing. For our application, interrupt coalescing involves transferring multiple incoming packets via DMA before sending them and then issuing a single interrupt. In our framework, the driver sets the number of packets per interrupt to coalesce. Using this approach, we can govern the number of interrupts delivered to the operating system. The number of packets to coalesce can be set as a static group size based on expected throughput, at which size the interrupts fluctuate depending on the current throughput. Alternately, the group size can change dynamically with fluctuating throughput to keep the interrupt rate constant.

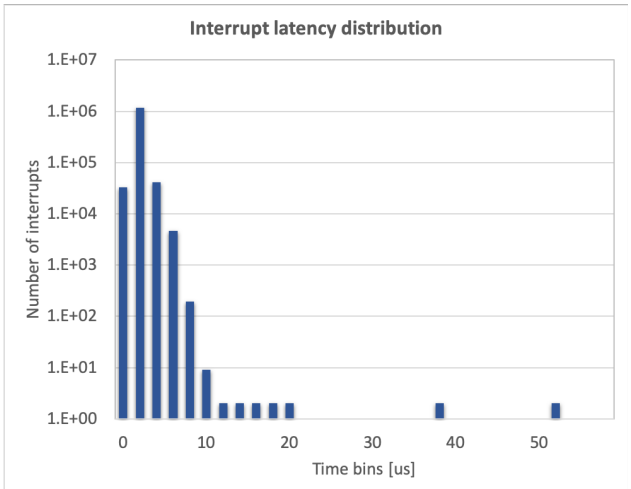

Figure 2: Expected interrupt latency distribution as measured on Linux 3.10 kernel.

A high number of interrupts can have a significant impact on overall performance as it adversely affects both kernel tasks and application processes. A high interrupt rate also increases the standard deviation of interrupt latency. We define interrupt latency as the time between the assertion of the interrupt by the firmware and the start of the corresponding interrupt handler routine in the operating system. Due to the limited buffering resources in our hardware, long interrupt latencies could result in data loss. While limiting the number of interrupts greatly reduces latencies, it doesn't

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-MOPHA156

eliminate the potential for long latencies. Our measurements show that Message Signaled Interrupts (MSI) are delivered to the kernel with more deterministic delays than legacy PCI interrupts. On multi-core systems, further improvements to interrupt latency times can be made by isolating one CPU core and dedicating it to the interrupt handler routine. Care must be taken to set CPU affinity for both kernel tasks and user space processes. CPU isolation is the most intense modification of the operating system configuration. It may also not fully utilize a single CPU core, which reduces overall system performance capability. We find it necessary only when many high throughput devices are installed in the same system. Figure 2 shows the distribution of interrupt latency measured on regular Linux kernel without any particular optimizations applied. Actual interrupt delay numbers vary based on selected CPU model and can be heavily impacted by the other system tasks.

APPLICATION LIBRARY

The Linux kernel environment is not well-suited for general purpose application development due to its limited programming interfaces and very little control or management of the actual code behavior. Programming mistakes will often result in system crashes, potentially changing system behavior in unpredictable ways, and in extreme cases even damaging hardware. Thus, most software applications execute in user space to take advantage of the rich set of programming interfaces provided by standard libraries, which include standardized interfaces for common hardware devices; however, user space applications must access hardware through a device driver. The application must assume additional complexity in order to interact correctly with the driver. To eliminate this complexity from the application, we have created a high-level application interface library that focuses on performance, general-purpose usability, and ease of use.

Communication with custom Linux device drivers can be done in several ways, including exposing the device through the */proc* file system, and more recently using the sysfs alternative. Most commonly the device file approach is used, where each device is explicitly represented with a distinct device file in the */dev/* folder. Device files can be opened as regular files, but such access is subject to device driver implementation as to what file functions are supported. The *read()* and *write()* system functions allow the transfer of data from/to the device driver, and bi-directional *ioctl()* is a general way for communicating with the device driver. It is exactly this universal ability to transfer arbitrary data to and from a kernel driver that makes the use of *ioctl()* unpopular with Linux developer community; however, due to its wide usage it is unlikely to be removed in future Linux kernel versions. Our device driver implements exclusively *pread()* and *pwrite()* hooks, and it uses an offset parameter as an index to select particular functionality within the device driver.

The application library provides C/C++ interface functions to work with devices through the Linux device driver. Functions to open and close the device are thin wrappers around the device file *open()* and *close()*, but add extra checks. When a device is opened, the device driver and application library exchange protocol versions to ensure compatibility. When configured for exclusive access, the device driver will reject any second or subsequent attempts to open the device. One of the commonly used functions is to query for device status and information from the device driver, such as the amount of DMA memory allocated. Functions that allow reading and writing of device registers, and function to transfer data to and from DMA memory, are described next.

The majority of device functionality is exposed through memory-mapped I/O register access, which allows the software application to use the register map based on the firmware implementation, without a reloading of the device driver. This capability is especially useful during FPGA firmware design and development.

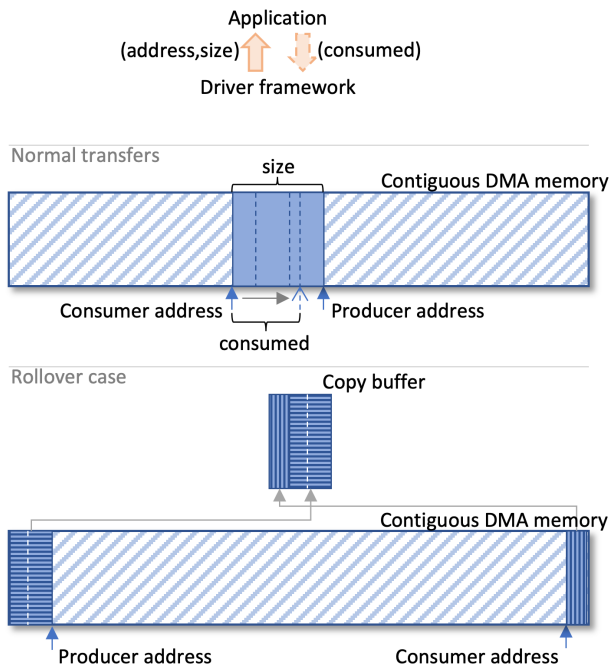

Figure 3: Implementation of zero-copy approach with contiguous DMA memory.

Functions for transferring data to and from the device are designed using a "zero-copy" approach. With zero-copy, data packets written to system memory by PCIe DMA engine are directly manageable by the software application. When an application requests data from DMA memory, the device driver and library exchange their producer and consumer indices to determine the valid data range. Then the library returns a pointer to the DMA memory where valid data begins, along with its size. The application is responsible for interpreting the data as packets. The byte count of the processed data must be acknowledged back to the library, which then moves the consumer index in DMA memory

ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-MOPHA156

accordingly, thereby enabling the FPGA firmware to push more data. Figure 3 visualizes the implemented zero-copy concept. This generic approach of passing memory blocks to the application allows for arbitrary data formats as well as DMA memory sizes. It also supports a DMA scatter-gather mechanism if implemented by the firmware in the future. One disadvantage of this approach is in the handling of the circular nature of the DMA buffer. This is especially true for the case where the application is decoding packets in-situ. As the producer index reaches the end of the circular buffer, the last packet gets split, with its beginning at the end of the circular buffer and the remaining part rolling over to the beginning of the circular buffer. In such cases the application cannot process the data and reports back to the library a processed data count of zero bytes. The library detects this rollover case and will use a small internally allocated buffer to make a copy of the contiguous data. The next time the application asks for data, it receives a pointer to the copied buffer. This is the only scenario in which data is actually copied, and the impact of this copy operation on the overall library performance overhead is negligible.

DIAGNOSTIC TOOLS

The diagnostic tools complete the framework and they serve three roles. As complete and executable programs, they serve as working code examples that are easy to understand and can be used as templates for application-specific modifications. Secondly, with generic diagnostic tools already available out-of-the-box, new projects can immediately focus on the development of the FPGA firmware and the software application, in parallel, and FPGA development can be tested almost completely before the customized software application is complete. The last but very important role of the diagnostic tools is to serve as an unbiased mediator between the FPGA firmware and the software application. Because they are used in many applications, are well tested, and less likely to misbehave, these tools can help identify and narrow down the source of any problems. The set of diagnostic utilities includes a tool for reading or writing arbitrary device registers; a tool for programming flash memory and rapid firmware updates; a data verification tool that interprets test packets and verifies test pattern integrity; a proxy tool that enables sending and receiving of packets through a POSIX pipe; and finally a Python extension to the application library with an analogous interface. While not as performant as C code, this last element is especially useful for quick prototyping and testing through Python scripting.

DEPLOYMENT STATUS

This Linux device driver framework has been extensively used during the past 5 years to collect event-based neutron scattering data from 17 neutron instruments at the Spallation Neutron Source as further explained in [1], and is being commissioned on 4 additional neutron instruments at the High Flux Isotope Reactor as of late 2019. It has been demonstrated to be capable of handling 3 distinct devices running at the maximum throughput of 200MB/s per device on a multi-core CPU system. This is more than adequate for all current SNS and HFIR neutron instruments, and provides adequate capacity for several even higher flux neutron instruments being designed for the Second Target Station project. The framework has also been entrusted with handling the raw amplitude and phase data of the newly developed LLRF system for the APS Upgrade project at Argonne National Laboratory, and has enabled APS to perform rapid prototyping of the MicroTCA platform for digital LLRF control. Moreover, the framework has supported accelerator developments at the Spallation Neutron Source for the MicroTCA-based FPGA firmware for Ring LLRF systems, the Injection Kicker Waveform monitor, and the MPS trigger controller.

SUMMARY

In this paper we presented the generic Linux device driver framework, as used in custom FPGA applications at the Oak Ridge National Laboratory and Argonne National Laboratory. Specific areas addressed in this article include: kernel driver optimization techniques, real use case benchmarks, and discuss the importance of a clean application programming interface (API) for seamless integration into control systems.

REFERENCES

[1] K. Vodopivec, B. Vacaliuc, "High Throughput Data Acquisition with EPICS", in *Proc. ICALEPCS2017*, https://doi.org/10.18429/JACoW-ICALEPCS2017-TUBPA05, 2018.