A Hybrid Device Driver for Next-Generation Solid-State Drives

Eleni Bougioukou and Theodore Antonakopoulos.

Department Of Electrical and Computer Engineering, Patras – 26504, Greece.
e-mails: bougioukou@upatras.gr and atonako@upatras.gr

Abstract

Solid state drives (SSDs) are based on non-volatile memories, like NAND Flash, and usually support fixed size data blocks. The typical data granularity is sector (512 B) or page (4 KB). Next generation SSDs will use new memory technologies, like Phase Change Materials (PCM) or 3D NAND Flash, with more advanced characteristics, ie. direct access of large data blocks and simultaneous support of small data chunks. For being able to support new types of applications, the SSDs must also be able to support various data sizes at their native data structures. In order to fully explore the capabilities of the underlying storage technology, this functionality requires the use of more advanced device drivers at the operating system. In this work, we present the architecture of such a hybrid Linux device driver, which supports concurrently the functionality of block and character device driver.

1. Introduction

SSDs are used extensively in today's embedded computing systems, either for storage purposes or as low-cost caching modules. SSDs are the most well-established technology for replacing magnetic hard disk drives, both in enterprise and consumer storage systems. This is mainly due to the low I/O latency that SSDs demonstrate which is in the order of tens of microseconds as opposed to tens of milliseconds for HDDS [1]. High performance SSDs are connected directly to the computing system's internal I/O architecture using the PCI Express (PCIe) bus, an interface that utilizes multiple serial Gbps lanes in parallel and achieves data rates of a few GB/s, depending on the used PCIe technology [2]. The PCIe functionality is exploited by using shared memory structures at the host memory for information exchange between the SSD's internal controller and the device driver internal logic.

Fig.1 presents a commonly used OS stack, the Linux kernel I/O stack [3]. The applications at the user-level access the storage devices through various system calls. The kernel forwards large data block requests to the virtual filesystem layer, which uses generic filesystem calls based on filesystem-specific functions. The filesystem is aware of the logical layout of data and metadata on the storage medium and sends read and write requests of fixed-size data blocks (typical size is 4K, named pages, and minimum size is 512 bytes) to the block layer on behalf of the user applications. The block layer provides an abstract interface which conceals the differences between storage devices of different technologies. User-level applications are also allowed to directly access the mass storage devices without the intervention of the filesystem for managing their data. This path is called “Direct” or “Raw” I/O. Block requests enter a request queue and finally arrive at the device driver, which is responsible for exchanging data with the storage device according to a specific host controller interface [4].

Next generation SSDs will use new memory technologies, like Phase Change Materials (PCM) or 3D NAND Flash, with more advanced characteristics, i.e. direct access of large data blocks and small data chunks. For being able to support new types of applications, these SSDs must also be able to support various data sizes at their native data structures. In order to fully explore the capabilities of the underlying storage technology, this functionality requires the use of more advanced device drivers at the operating system.

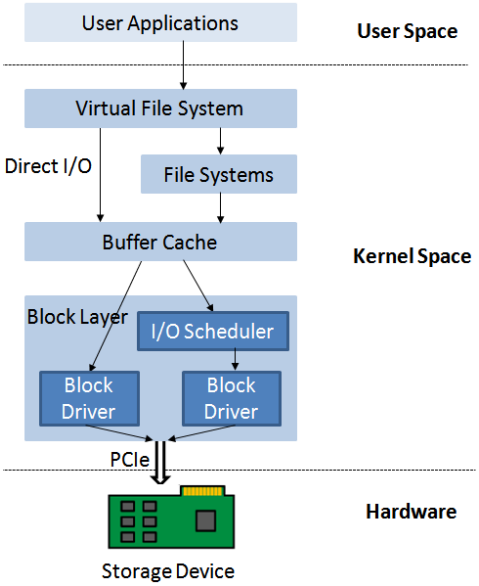

Fig. 1 Linux storage device I/O stack

In this work, we present the architecture of such a hybrid Linux device driver. The driver interacts with various kernel modules (block device drivers, character device drivers etc). Unlike traditional device drivers, it can accept requests of various data size i.e. large data blocks and/or small data chunks. The device driver is responsible for dispatching these requests to the actual storage device according to their interface. Information and data exchange is achieved by using a shared memory space on the main memory of the host, a set of registers in the PCIe address space and an interrupt mechanism.

2. Hybrid Device Driver

Fig.2 presents the Linux kernel I/O stack for the proposed hybrid device driver. User applications interact with any of the known Linux devices (block, character) sending them I/O requests of various data sizes. These modules make use of a custom interface exported by the hybrid device driver and transfer the requests to it. The hybrid device driver can accept requests from multiple modules at the same time. These requests are queued and dispatched to the storage device by using a flexible PCIe based interface. Finally, the hybrid device driver responds with the relevant I/O completions to the corresponding modules. I/O requests/completions are exchanged through a list of descriptors stored in a circular buffer, which represents the submission/response queue, as shown in Fig.3.

The driver is responsible for transforming each request to a suitable descriptor that will be passed to the storage device for processing. Each descriptor, stored in the circular buffer (host main memory), can be accessed by the PCIe storage device and the host processor. The descriptors are passed to the PCIe device in blocks of variable size. The blocks' size is dynamically adjusted in order to achieve high data rates, especially when the offered load is high. A block of descriptors corresponds to a subset of the total number of descriptors in the list. The offset of the next block to be processed along with its size are notified to the PCIe device through registers in the PCIe address space. Then, with a DMA transaction, the PCIe device transfers the block in its local memory, where the microprocessor processes the descriptors and initiates the respective read/write data transactions. In the mean time, driver continuous processes requests arriving from the various kernel modules. To ensure data integrity, access is forbidden to the memory space allocated to the block of descriptors that is currently being processed by the storage device. When a block has been fully processed and the content of its descriptors has been updated, the PCIe device returns the block to the host memory and informs the host processor by generating an interrupt.

Fig.4 provides a detailed description of the functionality of the hybrid device driver, which consists of three main functions, namely I/O Requests Handler, New Descriptors Block Processor and I/O Responses Generator. The I/O Requests Handler handles all requests received from the higher layers. The device driver transforms each request into a suitable descriptor, which is then placed into a shared memory and eventually is passed to the storage device for processing. Each descriptor is filled with information about its owner (host or PCIe device), the activity (command or response), the type of request (read or write), the physical address at the host main memory and the data offset in the PCIe device address space. Each request is placed into a waiting queue until it is completed.

When the driver receives an interrupt from the PCIe device that signals the completion of the previous block, the New Descriptors Block Processor function is activated for dispatching a new block with descriptors to the PCIe device, if there are any pending requests. To be able to adapt to different workloads, the descriptors are passed to the PCIe device in variable-size blocks. In the special case where a new descriptor is created when the PCIe device is idle, a block with a single descriptor is dispatched directly by the I/O Requests Handler function.

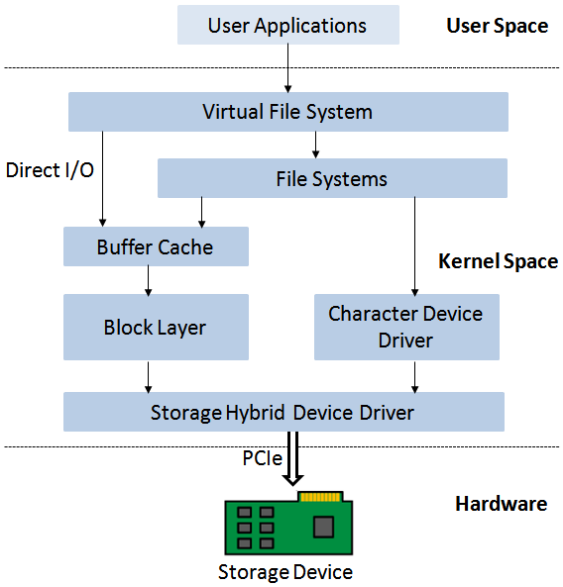

Fig. 2 Linux storage device I/O stack with hybrid device driver

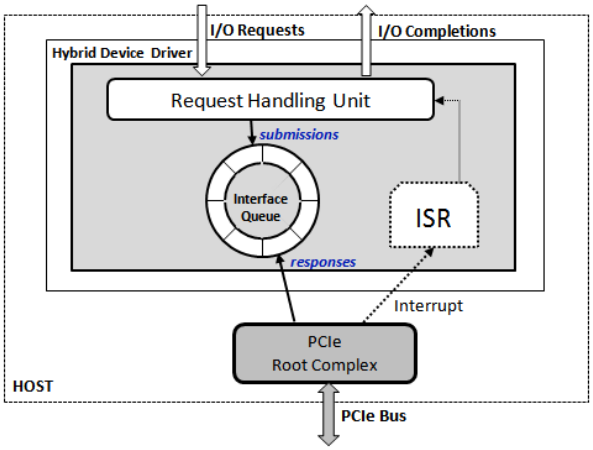

Fig. 3 Hybrid device driver

In this case, the system experiences the minimum possible latency. Registers at the PCIe address space hold the offset of the new block in the interface queue along with its size.

The New Descriptors Block Processor function wakes up the I/O Responses Generator function, which processes the block with the responses that have just been received by the PCIe device. This function checks sequentially all descriptors of the returned block and resets them in the shared memory, making them available for new transactions. It also wakes up the original I/O requests and generates I/O completions to the corresponding kernel devices, preserving their order.

Fig. 5 presents the synchronization between the host device driver and the PCIe device. When the device driver has a new block of descriptors to send to the storage device, it notifies the device and provides the offset of the next block to be processed along with its size through registers in the PCIe address space. Then the device transfers the block to its local memory, where the descriptors are processed and the respective read/write data transactions are initiated. When the block has been fully serviced and the content of the descriptors has been updated with the responses, the PCIe device returns the block to the host memory and informs the host processor by generating an interrupt.

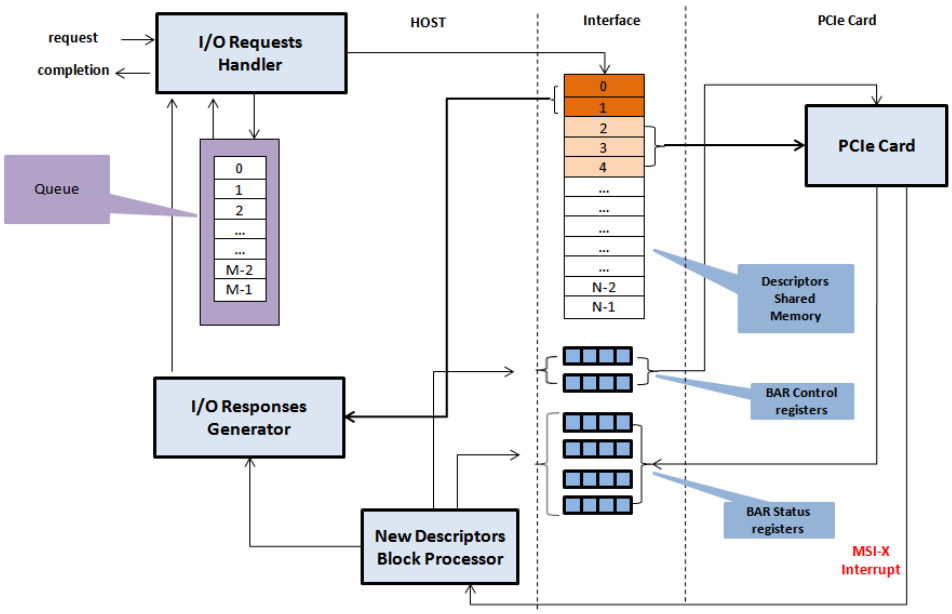

Fig. 4 The hybrid device driver architecture

Other modes of operation are also supported, ie. a descriptor is returned to the host immediately as the respective command has been completed or a block is partially answered using subsets. Then, the device driver processes the returned block, generates the respective I/O completions of the original I/O requests and at the same time grants access to the PCIe device for the next block of descriptors.

The diagram also presents an enhancement of this interface. In a typical system, sequential block processing results to loss of processing power, since idle time occurs between issuing an interrupt and information update of the next block to be processed. Delays in the interrupt dispatching by the host, along with OS delays due to context switching, can lead to exceptionally high idle execution times despite the fact that the device driver may have already prepared the subsequent blocks of descriptors in host memory space. This can lead to performance degradation. For confronting this, we propose an enhancement to this interface with a smart forward command mechanism, where the driver not only sends the current block of descriptors but also informs the device regarding a few pending descriptors, independent to the size of the next block. This way, when the device returns a serviced block of descriptors, it already has a number of the next descriptors in its local memory and starts processing them. As a result, the device remains constantly active and the maximum performance is achieved.

A major advantage of the proposed device driver structure is that it supports dynamically adaptable and variable-size blocks of descriptors. The maximum supported descriptors' block size is determined by the internal capabilities and functional characteristics of the PCIe storage device. For that reason, the proposed device driver specifies dynamically the size of the next block, according to the currently applied workload. That means that whenever the PCIe device sends an interrupt to inform the host of its availability, the New Descriptors Block Processor function determines the size of the next block of descriptors, taking into account the number of pending requests and the status of the PCIe device. It then dispatches either a maximum size block, leading to maximum possible I/O throughput, or a block with all pending requests, minimizing I/O latency.

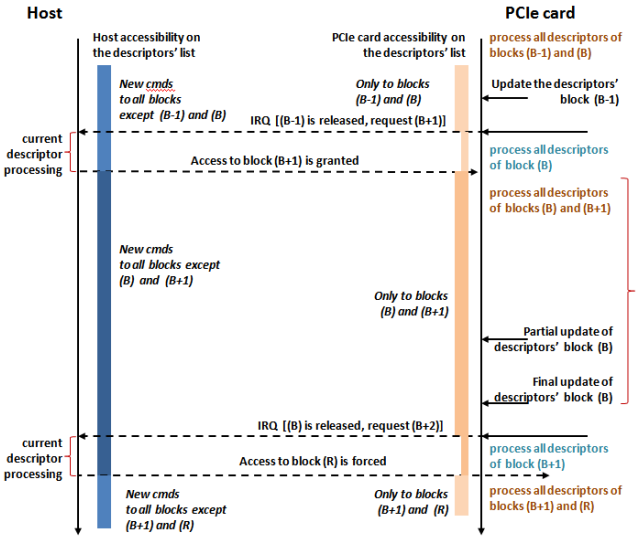

Fig. 5 Exchange of blocks of descriptors between the host and the PCIe storage device.

3. Experimental Results

For this work, a complete PCIe storage device prototype was built using the powerful Xilinx ZC706 development platform, which is based on the Zynq-7000 SoC architecture. Zynq-7000 integrates a feature-rich dual-core ARM Cortex-A9 MPCore processing system and Xilinx programmable logic in a single device. The ARM CPU is the heart of the processing system which also includes on-chip memory, external memory interfaces, and a rich set of I/O peripherals. The various hardware controllers as well as the processing system are I/O interconnected via high-bandwidth AMBA AXI interfaces. The ZC706 board provides a hardware environment for developing and evaluating designs targeting the Zynq-7000 Programmable SoC and includes features, such as DDR3 SODIMM memory component and four-lane PCIe Gen. 2 interface, that enable building high-performance embedded systems. Based on this platform, we were able to build the prototype of a highly reconfigurable PCIe storage device, which was used to validate the efficiency of the proposed hybrid device driver and PCIe

host controller interface for PCIe storage devices, to investigate alternative configurations and procedures and to evaluate the performance of the various components individually and the whole system as well. Since in this work we are focusing on the performance of the device driver and the PCIe interface, instead of the actual chips, the DDR3 memory of the ZC706 platform was used. This way the only hardware limitation is the read/write rates of the DDR3 SODIMM controller.

Each block of descriptors is initially transferred in the local memory with a DMA PCIe transaction, and then the microprocessor starts analyzing the descriptors' contents and executes the commands. Depending on the type of command (read or write), for each descriptor a different type of DMA PCIe (Device-to-Host or Host-to-Device) transaction is initiated. There are two different scenarios. Each descriptor is answered to the host individually or the whole block with descriptors is processed and finally answered to the host.

According to the experimental results of Fig. 6 higher performance is achieved during a read operation. This is due to the different PCIe procedures regarding the read and write transactions, as well as variations in the implementation efficiency of the hardwired PCIe controller of the ZC706 development platform. In the case when descriptor is answered to the host individually (4K, RPR), the DMA engine is blocked and cannot be used for the data transfer of the next command until the previous descriptor has been successfully transferred to the host memory. So the performance decreases as the block size is increased. In the case of answering all the commands as a single block (4K, RPB) we can see that the maximum I/O rate is achieved when the block size has the maximum value. On the contrary, minimum latency is achieved by returning each descriptor to the host immediately as it is served.

As already described, the hybrid dynamic driver can accept requests of variable data sizes. Fig. 6 presents that the maximum performance is achieved for read and write requests of large data blocks. In all experiments, a custom tool was configured to send synchronous requests of variable size data blocks (4K, 512B, 64B etc).

In order to be able to evaluate the latency of each component of the storage system, we measure the latency of three different paths (Fig.7). The first path includes the total system i.e. hybrid device driver, PCIe, controller and DRAM. At the second path the controller and the DRAM have been replaced by a loopback. Finally, the third path includes only the hybrid device driver and a loopback. The measured latency for each path for read and write transactions is illustrated in Table 1. According to these measurements, the presented device driver exhibits very low latency, comparable with the performance required by high-end system using the latest PCIe technology.

Paths	Read Latency (usecs)	Write Latency (usecs)
1	22.15	24.82
2	7.14	7.14
3	0.99	0.99

Tab. 1. Latency of individual paths of the whole system

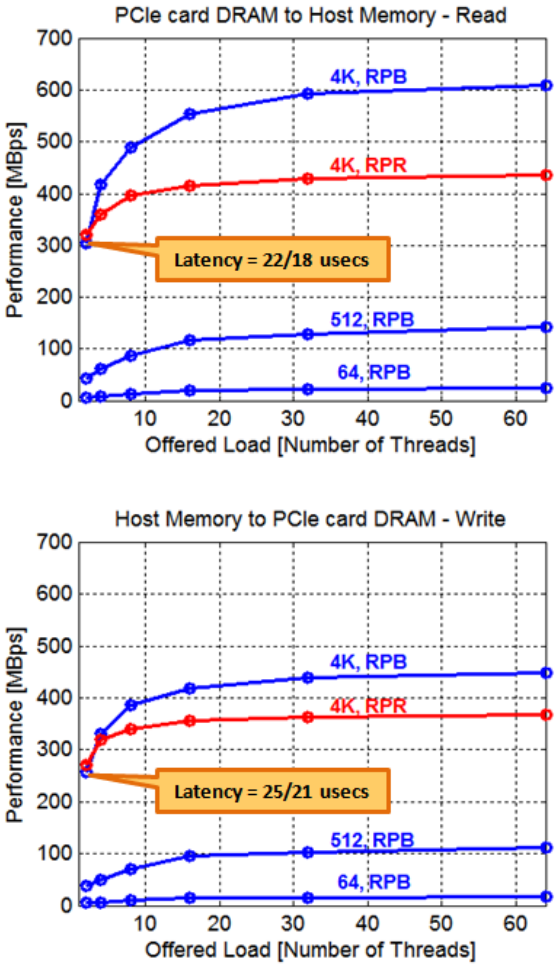

Fig. 6 Experimental results for read and write transactions

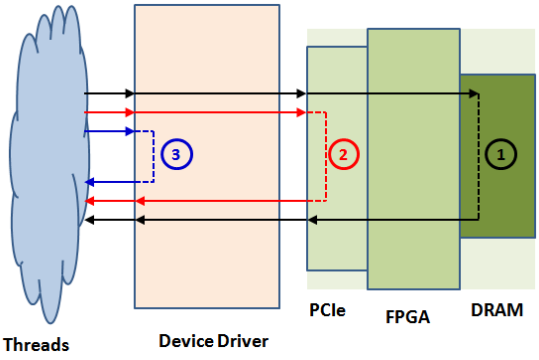

Fig. 7 Individual paths of a request

4. Conclusions

In this work, we presented the architecture and functionality of a hybrid Linux device driver, which supports concurrently variable size data. The device driver demonstrates improved performance and can be used in high-end systems.

References

1. J. Brewer and M.Gill, "Nonvolatile memory technologies with emphasis on flash: A comprehensive guide to understanding and using flash memory devices", Wiley-IEEE Press, 2008.
2. "PCI Express Base Specification, Revision 2.1", PCI SIG, Tech. Rep., March 4, 2009
3. D. P. Bovet and M. Cesati, "Understanding the Linux kernel" 3rd ed. Oreilly & Associates Inc., 2005
4. J. C. A. Rubini and G. Kroah-Hartman, "Linux Device Drivers" 3rd ed. Oreilly & Associates Inc., 2005.