Prototyping and Performance Evaluation of a Dynamically Adaptable Block Device Driver for PCIe-based SSDs

Eleni Bougioukou, Athina Ntalla, Aspa Palli, Maria Varsamou and Theodore Antonakopoulos
University of Patras
Department of Electrical and Computer Engineering
Patras 26504, Greece
e-mails: *<bougioukou, athinantalla, apalli, mtvars, antonako>@upatras.gr*

***Abstract*—Solid-state drives use non-volatile memories for storing and retrieving information in the form of sectors and/or pages and demonstrate better performance than hard disks. In many cases, the maximum IO performance of the used memory technology is not achieved due to limitations imposed by the software device driver that interfaces the storage card with the hosts's operating system. Today's computing machines with conventional operating systems have been developed based on the performance characteristics of hard disk drives. In this work, we present the prototype of a new block device driver with a flexible host-device interface suitable for PCIe-based solid-state drives. The block device driver is compatible with the standard software dataflow of a Linux-based OS, and at the same time exploits the operational features of such devices to provide improved performance. Experimental results that demonstrate how the system performance is affected by decisions on the device driver's functionality are presented along with the used testing methodology.**

I. INTRODUCTION

Solid state drives (SSDs) that utilize non-volatile memories (NVM), like NAND Flash and Phase-Change Media (PCM), is the most well-established technology for replacing magnetic hard disk drives (HDD), both in enterprize 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]. Multiple NVM channels operating in parallel are used in SSDs, thus increasing the total I/O rate that can be achieved by an SSD to nearly a million IOs per second (IOPS), as opposed to thousands of IOs observed on traditional magnetic hard drives [2].

Today the most well known NVM technology is NAND Flash, which is used in almost all commercial SSDs. PCM is a new emerging NVM technology that demonstrates DRAM-like read performance, comparable write performance and much higher endurance than Flash, but still much lower storage density. Although the various NVM technologies have different characteristics in terms of minimum and maximum data block size, rewritability, need for erase before write, endurance, aging and raw bit error rate, they require similar functionality by the SSD's storage controller, which is responsible for transferring the data between the host system and the actual NVM chips at

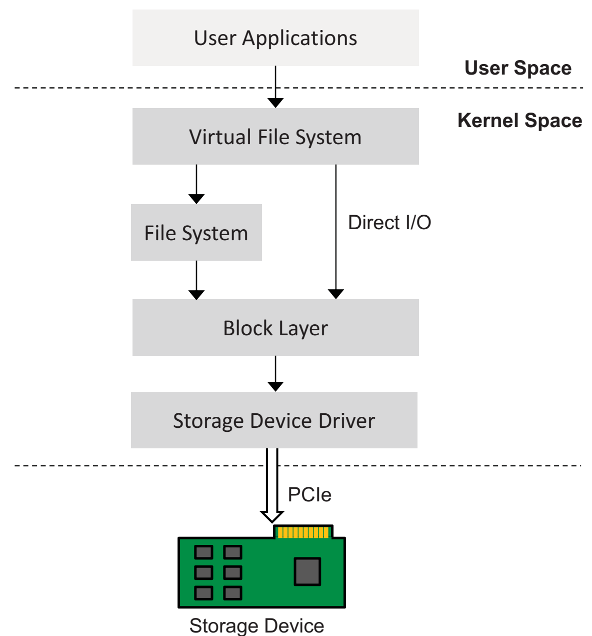

Fig. 1. Linux Storage Device I/O stack.

very high rates and with extremely high reliability. To ensure backward compatibility, first generation SSDs were connected to a host as external storage devices using the SATA/AHCI interface. But SATA was designed for replacing mechanical hard disk drives, and has become increasingly inadequate as SSDs achieve higher IO performance. High performance SSDs are connected directly to the host's internal I/O architecture through the PCI Express bus [3], a serial interface that can utilize multiple lanes in parallel and can achieve data rates higher than 1GB/s, whereas SATA rev. 3 can offer data speeds of approximately 600 MB/s.

Regarding the interface with the user applications at the host system and to ensure transition transparency, meaning no changes at the user-level, the OS software layers and I/O stack used for the HDDs remain the same and the differences are encapsulated by an emulation software layer, called the

Flash Translation Layer (FTL), which is added in the SSD's storage controller. A major drawback of this approach is that, since most contemporary OS storage layers and I/O stacks are developed and optimized based on functional assumptions that are valid for mechanical disks only, the SSDs are prevented from reaching their full potential performance. Fig. 1 presents a commonly used OS stack, the Linux kernel I/O stack [4]. The applications at the user-level access storage devices through standard system calls to the filesystem. The kernel forwards these requests to the virtual filesystem (VFS) layer, which interfaces generic filesystem calls to 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 (usually 4K bytes, named pages) 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 mass storage devices without using a filesystem to manage 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 [5].

The performance of an SSD, both in terms of I/O latency and I/O throughput (in kIOPS), depends on the characteristics of various components, i.e. the used NVM technology, the NVM channel, the internal architecture of the storage controller, the host I/O interface and the upper layer software components, especially the device driver. As aforementioned, the traditional OS storage layers are tailored on HDDs characteristics and impose a performance bottleneck on SSDs. Due to that, some applications and device drivers are choosing to bypass the traditional stack, thus improving the resulting performance [6]. This choice increases complexity and removes generic features that are provided by a common OS storage layer. In another work, a modification of the block layer was proposed that adds multiple I/O submission/completion queues in both software/hardware levels to exploit the multicore architecture [7].

In this work, we present the architecture of a dynamic block device driver that is compatible with the conventional Linux I/O stack, but at the same time it takes into account the special characteristics of a PCIe NVM storage device, thus providing a high-performance interface between the host computer and the storage device. The proposed device driver is characterized as dynamically adaptable, since it changes in real time its functional parameters according to the applied user workloads. As a result, minimum latency is observed when light workloads are applied, while the I/O throughput is maximized when heavy workloads have to be serviced. Although the device driver has been developed in the Linux environment, with some necessary modifications, it can be used in any OS system. The adaptability of the device driver is supported by a customized high-performance PCIe host controller interface, based on dynamically-changing queues. To evaluate the performance of both the proposed device driver and the PCIe host controller interface and investigate various workload adaptation techniques, a complete PCIe storage device prototype was built, which incorporates both dedicated reconfigurable hardware components and optimized microprocessor software modules.

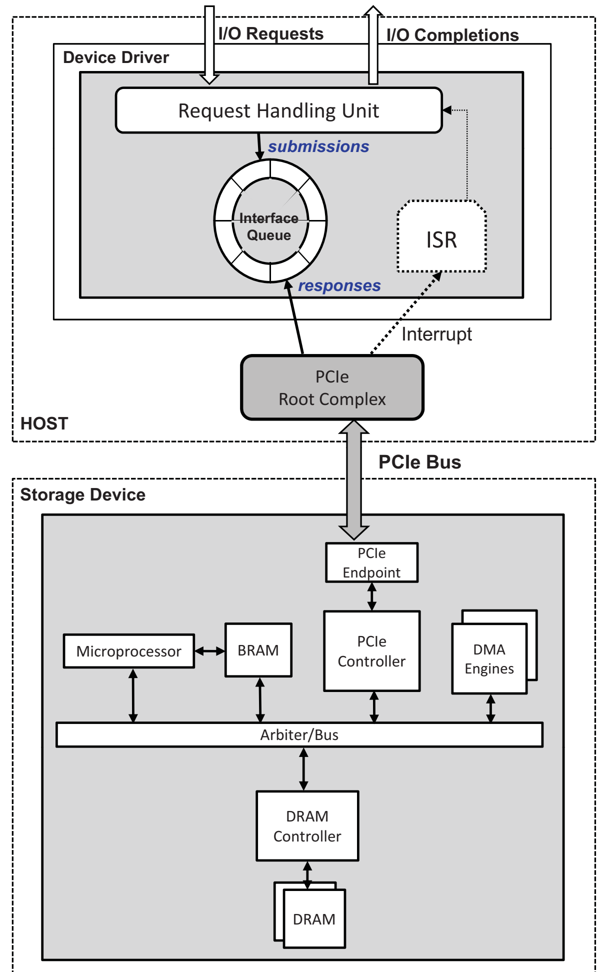

Fig. 2. PCIe NVM storage device architecture.

Section II describes the general architecture of the PCIe storage device, with emphasis on the block device driver and the PCIe host controller interface. The dynamically adaptable device driver is further analyzed in Section III and performance measurements for various workloads are presented and analyzed in Section IV.

II. PCIe STORAGE DEVICE SW/HW ARCHITECTURE

Fig. 2 shows the structure of the dynamic device driver for PCIe NVM storage devices proposed in this work, along with the general architecture of such a device, which supports the proposed PCIe host controller interface. The Linux-compatible block device driver accepts I/O requests from the user applications through the Linux block layer and responds with the relevant I/O completions. The operating systems provide advanced algorithms to order the incoming requests, which are tailored to the inherent characteristics of hard disks. For example, in hard disks, due to the rotational delay and head seek time, random accesses that require frequent disk head movement are slow, while sequential accesses that only require rotation of the disk platter are fast. To achieve

higher performance, various scheduling functions have been incorporated in the block layer that queue multiple commands to minimize head movement or re-order commands taking rotational positioning into account. However, since SSDs often exhibit negligible latency difference between sequential and random I/Os, these queue management functions are of no use and add unnecessary CPU overhead. Because of that, the interface of the proposed device driver with the upper Linux layers was designed to substitute the scheduling algorithms in the block layer with a straightforward approach, where all I/O requests enter an unordered queue and they are processed by the device in a First-Come First-Served (FCFS) order.

The NVM storage device is connected as a PCIe endpoint on a slot of the PCIe root complex of the host motherboard and the device driver is responsible for forwarding the read/write requests to the device according to the PCIe host controller interface. In most contemporary SATA, SAS and PCI-E based SSDs, information and data exchange is achieved through a submission queue and a completion queue, stored in a memory space on the host's main memory, which can be accessed by both the PCIe device and the host processor, supported by a set of registers in the PCIe address space and an interrupt mechanism, where a single interrupt is used to signal completions. This interface has been inherited by the HDDs that always service commands in a serial way and does not exploit a crucial feature of SSDs, meaning the ability to execute simultaneously many different read/write requests, due to the fact that the NV memories are used in sets of chips that operate in parallel. To exploit the increased parallelism of SSDs, recently a new PCIe interface was proposed, namely NVM Express [8], which supports a flexible number of submission and completion queues. With NVMe each user-space process can have its own submission and completion queue to the SSD accompanied by a dedicated interrupt. However, its implementation involves complicated interrupt handling and thread synchronization routines and usually requires modifications at the user-application level. To ensure user data consistency and OS compatibility, we propose a low-complexity flexible PCIe interface, which still uses a single submission/completion queue, but the requests are sent to the storage device in variable-size blocks and the maximum block size is determined by the parallelism capability of the storage device.

I/O commands/responses are exchanged through a list of descriptors stored in a circular buffer, which represents the submission/completion queue. A block of descriptors corresponds to a subset of the total number of descriptors in the list. All descriptors in a block are continuous in the descriptors' memory space and the blocks are dispatched in a sequential way. Each descriptor contains information regarding the location of data in the host's main memory and the respective storage area in the NVM memory space, combined with necessary control flags, such as descriptor ownership (host or device). Certain fields are valid only when the descriptor corresponds to a response and indicate successful execution or not and type of error, if any. Fig. 3 explains the interface synchronization between the host device driver and the PCIe device using a timing execution diagram. 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 in its local memory, where the descriptors are processed and the respective read/write data transactions are initiated. To ensure data integrity, the host processor is not allowed to access the memory space allocated to the block of descriptors that is currently being processed by the storage device. 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. Alternative modes where each descriptor is returned to the host immediately as the respective command has been completed or the block is returned in subsets are also supported. 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 a future enhancement of this interface. For optimization purposes, when the device returns a serviced block of descriptors, it already has the first commands of the next block in its local memory and starts processing them locally, before getting the next block of commands. This enhancement is currently under development, but the first simulation and experimental results show that this approach can further improve the interface's efficiency, especially when Gen-3 PCIe technology is used.

Fig. 3. Exchange of blocks of descriptors between the host and the PCIe storage device.

Apart from the NVM storage channels and the associated controllers, the general architecture of a PCIe storage device that supports the proposed interface also includes a PCIe controller, local BRAM memory and a microprocessor that acts as the system's supervisor and coordinator. One or more local busses are used for transferring the data between the memory, the PCIe interface and the NVM storage controllers

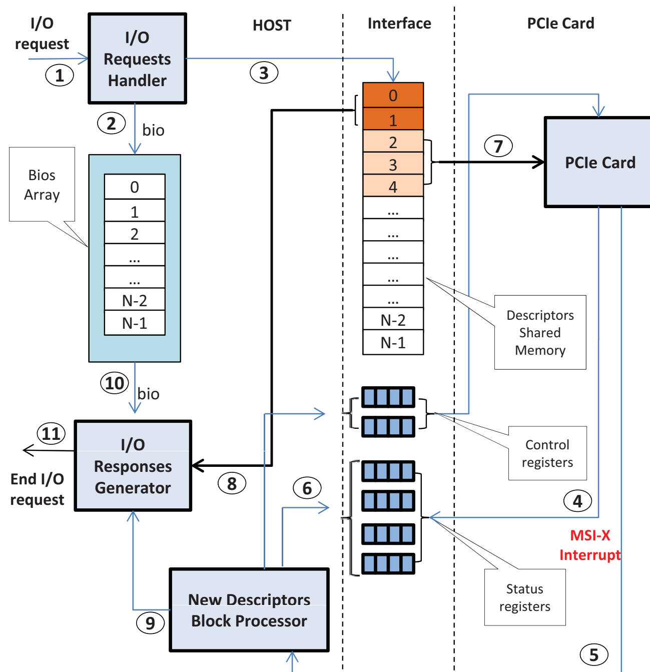

Fig. 4. Device driver architecture.

using dedicated DMA engines. When the microprocessor is notified that a new block of descriptors is available in the host memory, it initiates a PCIe DMA transaction to transfer the descriptors in its local memory. Then the descriptors are processed one-by-one or in parallel and depending on the type of I/O commands, it initiates PCIe DMA data transactions from/to the host memory and writes/reads data to/from the NVM chips accordingly. When a command completes, the microprocessor checks for errors and updates the respective descriptor. The updated block of descriptors is returned to the host via a PCIe DMA transaction followed by a PCIe interrupt. A single DMA engine may be used for both descriptors and data transfers or multiple separate ones.

III. Dynamically Adaptable Block Device Driver

Fig. 4 provides a detailed description of the functionalities of the dynamic 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* function handles all requests received from the higher layers. Linux defines a special structure to represent the I/O request between the block layer and the device driver, which is called 'bio' and corresponds to 4KBytes Linux page data transfer. The bios are passed directly from the block layer to the driver space without any prior manipulation and then they are added to a dedicated request queue. The device driver is responsible for transforming each bio into a suitable descriptor, which is then placed into the interface queue and eventually is passed to the storage device for processing. Each descriptor is filled with information about the owner of the descriptor (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. 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 pending requests. To be able to adapt to different workload conditions, 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. Registers in the PCIe address space hold the offset of the new block in the interface queue along with its size. Additionally, the *New Descriptors Block Processor* function wakes up the *I/O Responses Generators* function, which processes the block with the responses that have just been returned by the PCIe device. This function checks one by one all the updated descriptors of the returned block, generates the respective I/O completions for the original I/O requests and sends them to the user applications through the block layer, preserving the original order. In the case that there were errors during data transfers, the I/O completions inform the user applications accordingly. Finally, it resets the appropriate descriptors in the interface queue, making them available for new transactions. At the same time, the driver continues processing requests arriving from the block layer.

Necessary structures that ensure synchronization between the bios' queue and the interface queue are defined. More specifically, a special *'Bios Array'* is defined, which keeps the memory pointers of the pending bio structures, along with two pointer variables, one which points to the first free position in the interface queue for the future descriptors and another one which points to the first descriptor of the next block that will be sent to the PCIe device. To avoid time-consuming search software routines, the position of each bio pointer in the *'Bios Array'* corresponds to the position of the respective descriptor in the interface queue. So, the size of the *'Bios Array'* depends on the total number of descriptors that can be stored in the interface queue at the host main memory. The interface queue is implemented as a cyclic buffer, whose size is fixed and is determined by the OS during initialization, based on parameters retrieved by the storage device. These parameters provide information regarding its internal architecture (storage capacity, NVM technology, number of channels, read/write times etc.). For implementation complexity reasons, the size of the interface queue is always a power of two. Every time a block of descriptors is returned from the PCIe device, the *I/O Responses Generator* function finds the respective bio pointers in the *'Bios Array'*, generates I/O completions and removes the pointers from the array.

A major advantage of the proposed device driver structure is that it supports variable-size blocks of descriptors. The maximum supported descriptors' block size, determined by the internal capabilities and functional characteristics of the PCIe storage device (e.g. number of NVM channels, different queues to serve even/odd pages), could be used as a fixed block size of this interface. However, in the case of light workloads, if the driver had to wait for the maximum number of descriptors to be collected, the resulting latency would increase significantly. 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 host of its availability, the *New Descriptors Block Processor* function

decides 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 the I/O latency.

IV. Performance 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 All Programmable SoC architecture. Zynq-7000 integrates a feature-rich dual-core ARM Cortex-A9 MPCore based 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 dynamic device driver and PCIe host controller interface for PCIe NVM 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 NVM 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.

Initially, the efficiency of the proposed PCIe host controller interface was studied independently to the device driver, because the performance of a storage device, as it is observed by the user, both in terms of KIOPS and I/O latency, is significantly affected by the performance of the interface between the host and the storage device. In the case of a PCIe-based interface, its performance depends on the efficiency of the PCIe controller both in the host root complex side and the device endpoint side, the number of PCIe transactions needed for each data transfer, which is determined by the communication protocol between the host and device, and the type of command (PCIe read commands require more PCIe packets than the PCIe write commands for the same ammount of data). Regarding the proposed interface, the number of distinct PCIe transactions needed depend on the size of the block of descriptors, whether it is fixed or changes dynamically, the number of DMA engines used in parallel and how the responses are generated (i.e. all responses in a single block, each response is generated when the respective command has been completed, or in sets of completed commands).

In order to study the effect of the size of the block of descriptors, a static version of the device driver was used, where the descriptors are exchanged in blocks of predefined

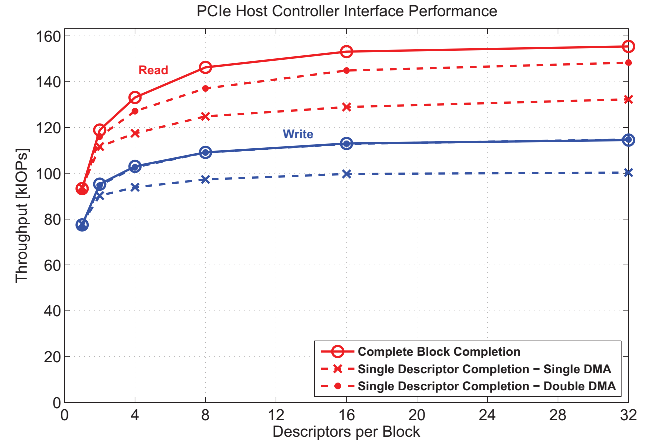

Fig. 5. Maximum performance achieved when fixed size blocks are used.

size, less than or equal to the maximum number of requests that the storage device can support in parallel. The lowest I/O latency is achieved when a single new request is sent to the PCIe card. On the contrary, the highest throughput is achieved when the full parallelism of the storage device is exploited, usually expressed as the maximum block size. To measure the maximum achievable performance at the interface, the device driver was constantly fed with I/O requests, so that there are always pending full blocks of descriptors. Fig. 5 highlights the performance of the system, in terms of throughput, versus the size of the block of descriptors when only read or write requests are serviced, for two different policies that the device may support, complete block and one-by-one descriptor response. Especially for the case of the one-by-one descriptor response, the use of an additional DMA engine dedicated solely to descriptors transfers was also investigated.

It is obvious that as the number of PCIe transactions needed to be performed for the same amount of descriptors and data transfers increases, the total exchange rate decreases. However, to better analyze this case, the block processing procedure in the storage device must be examined. 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. To optimize the performance, while waiting for the DMA transaction to be completed, the microprocessor starts servicing the next command. So, the inter-command execution time is determined by either the DMA completion time or the command processing time, whichever is the maximum. Although the descriptor processing time is independent to the type of its requests, read or write, the inter-command execution time is significantly higher in the case of a write request. 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. As a result, the maximum I/O throughput can be achieved when the user performs only read operations. Since data are stored in DRAM and not

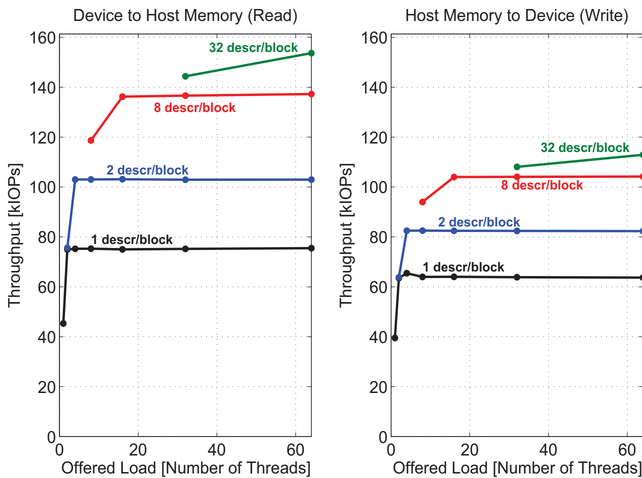

Fig. 6. Performance of the static version of the block device driver.

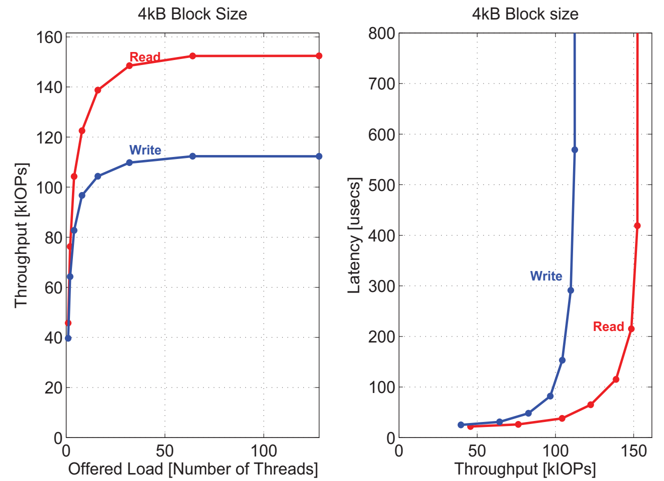

Fig. 7. Performance of the dynamic version of the block device driver.

in actual NVM technology, which is characterized by much higher read/write times, this is the maximum performance that can be achieved in such a storage device. Mixed workloads will achieve a performance between the all-read and the all-write performances.

Finally, in the case when each descriptor is answered to the host individually, 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. Since these are two independent processes, this situation can be avoided by using a separate DMA engine for the transfer of descriptors. It should be noted that the I/O latency is affected by the descriptors' block size and the serviced descriptors' return strategy. The lowest latency can be achieved with the smallest block size and by having each descriptor returned to host immediately as it is served, while the maximum I/O rate is achieved by maximizing the block size and by answering all commands as a single block.

For estimating the performance of the complete architecture, device driver and PCIe interface, the user-space tool named *Fio* was used [9]. *Fio* is a common benchmarking tool for mass storage devices that support both synchronous and asynchronous, direct and filesystem I/Os using one or multiple threads. A constant flow of requests is generated, random or sequential, with a specified percentage of read/write requests. It reports statistics such as IOPS, throughput, and average latency. We used the tool to generate different workloads, where a number of threads that run in parallel send continuously direct synchronous I/O requests to the block device. Fig. 6 shows the performance of the system with the static version of the device driver, in terms of throughput in KIOPS, versus the number of threads and different descriptors' block sizes, for read-only and write-only workloads. To be able to estimate the maximum performance that can be achieved by the device driver for the various configurations, only full descriptors' blocks are allowed to be sent to the PCIe device. That, along with the fact that synchronous user-space requests are blocked until they are completed, the testing methodology explains why there are no measurements for number of threads less than the number of descriptors in a block.

The system reaches its maximum performance when the offered load is such that there always exists a full descriptors' block with pending requests when the device driver gets an interrupt that signals the availability of the storage device. For any descriptors' block size, this can only be achieved if the number of threads that issue parallel requests is greater than that particular size. Since the driver grants access to the storage device for only one block with a fixed number of descriptors at a time, any increase in the number of threads, and consequently the offered load, does not further improve throughput. The additional requests are just stored in the device driver host memory waiting for another block to be sent to the storage device. Although the throughput is improving significantly for larger descriptors' block sizes, the latency also increases. So, it would be preferable if the descriptors' block size was updated dynamically according to the applied workload.

A. The Dynamically Adaptable Device Driver

Fig. 7 demonstrates the performance achieved using the dynamically adaptable version of the device driver, for read-only and write-only scenarios. This figure also presents the relation between throughput and latency observed in the system. In this case, the number of descriptors that can be sent to the storage device varies according to the offered workload. As the offered load increases, the descriptors' block size also increases, and higher I/O rate is achieved, while when the workload decreases, the descriptors' block size decreases as well, ensuring better latency. Fig. 7 verifies that the driver adapts to the offered load and the maximum achievable throughput is achieved for all cases. As the offered load exceeds the system's storage capability, the latency, as expected, increases exponentially.

In all experiments, the *Fio* tool was configured to send synchronous requests of 4kB data blocks. Each request enters the device driver as one bio structure that represents one Linux page transfer and the user application cannot send another request before the previous request has been completed. However, a user application may issue requests of variable sizes. Since Linux manages data in 4kB pages, requests of less

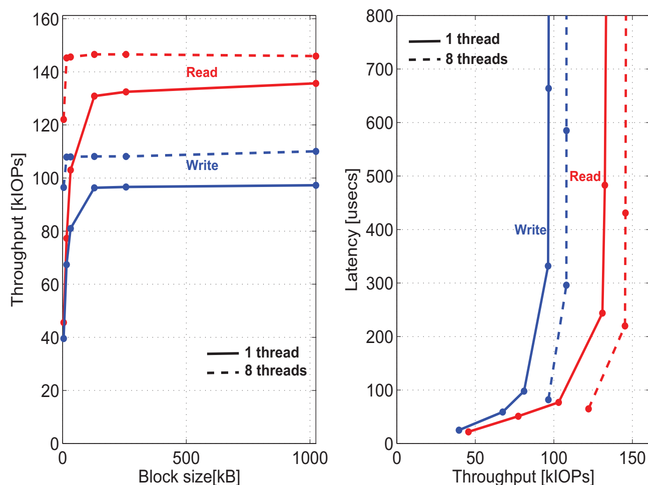

Fig. 8. The effect of single and multi-threaded applications with variable block sizes.

that 4kB are executed with one 4kB transaction with padding bytes, while requests of more than 4kB are partitioned and enter the device driver level as multiple bios of 4kB data blocks. For a request to be completed, the total number of corresponding bios has to be serviced. Fig. 8 shows that the size of the data requests generated by the user applications has a significant effect on the performance of the storage device. It can be seen that when a single-threaded user-space application uses data requests of large block sizes, a better throughput can be achieved, but at the same time the latency also increases.

V. Conclusions

We presented a dynamically adaptable block device driver for PCIe-based SSDs along with a suitable PCIe host controller interface. The device driver adapts its parameters to the user applied workloads and to the current status of the storage device, and that results to minimum latency and high I/O performance when light and heavy workloads are applied respectively. A fully-functional PCIe storage device prototype was built and various configurations and system parameters were studied. Experimental results have shown that the presented approach can fully exploit the capabilities of the used storage device when proper loading conditions are applied.

Acknowledgment

This work was supported by the IBM Zurich Research Laboratory in the framework of a Joint Research Program.

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] Feng Chen, Rubao Lee and Xiaodong Zhang, "Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing," in *The 17th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, San Antonio, Texas, USA, February 12-16 2011.

[3] "PCI Express Base Specification, Revision 2.1," PCI SIG, Tech. Rep., March 4, 2009.

[4] D. P. Bovet and M. Cesati, *Understanding The Linux Kernel.*, 3rd ed. Oreilly & Associates Inc., 2005.

[5] J. C. A. Rubini and G. Kroah-Hartman, *Linux Device Drivers.*, 3rd ed. Oreilly & Associates Inc., 2005.

[6] A. M. Caulfield, T. I. Mollov, L. A. Eisner, A. De, J. Coburn, and S. Swanson, "Providing safe, user space access to fast, solid state disks," *SIGARCH Comput. Archit. News*, vol. 40, pp. 387–400, 2012.

[7] Matias Bjoerling, Jens Axboe, David Nellans and Philippe Bonnet, "Linux block io: Introducing multi-queue ssd access on multi-core systems," in *The 6th ACM International Systems and Storage Conference - SYSTOR13*, Haifa, Israel, June 30July 2 2013.

[8] "NVM Express Specification, Revision 1.1a," NVMHCI Workgroup, Tech. Rep., September 23, 2013.

[9] Fio: Flexible i/o tester. [Online]. Available: http://freshmeat.net/projects/fio/