FastPath: Towards Wire-speed NVMe SSDs

Athanasios Stratikopoulos, Christos Kotselidis, John Goodacre and Mikel Lujan

July 8, 2018

FastPath: Towards Wire-speed NVMe SSDs

Athanasios Stratikopoulos, Christos Kotselidis, John Goodacre and Mikel Luján
School of Computer Science
The University of Manchester
Manchester, UK
Email: {Athanasios.Stratikopoulos, Christos.Kotselidis, John.Goodacre, Mikel.Lujan}@manchester.ac.uk

***Abstract*—The constant growth of data and its importance to drive Machine Learning and Big Data is pushing storage systems towards ever increasing I/O bandwidth and lower latency requirements. In recent years, the Non Volatile Memory Express (NVMe) standard has enabled SSD drives to deliver high I/O rates by allowing the storage to be connected directly via the fastest available interconnect to the processing chip. In parallel, the adoption of FPGAs in data centers is creating opportunities to accelerate various applications and/or Operating System (OS) operations. While, FPGAs in data centers have been connected via PCIe to mostly x86 servers, we have now also available heterogeneous SoCs with multi-cores and FPGAs integrated on the same die and connected by an on-chip interconnect.**

In this paper, we present how to rethink and accelerate NVMe performance on heterogeneous SoC with integrated FPGAs providing a first research insight on the performance benefits of such an approach. We provide an analysis of the Linux block I/O layer and showcase the relationship between the system's performance and its I/O implementation. Consequently, we introduce an FPGA-based fast path which accelerates the access to the NVMe drive. Our comparative evaluation demonstrates that our FPGA-based FastPath achieves up to 71% lower latency and up to 5x higher I/O performance against the baseline system on a Zynq development board.

***Keywords*-NVMe; SSDs; FPGA; Linux Block I/O; Heterogeneous Systems**

I. Introduction

Modern storage technologies, such as the NVMe standard, leverage high throughput interconnects (i.e. PCIe) providing significant I/O performance improvements for modern SSDs. Although NVMe SSDs have become ubiquitous in data centers, still exists performance bottlenecks to be tackled. The majority of the bottlenecks derive from the Operating System (OS) storage I/O software stack (e.g. file system & Direct I/O, Block I/O, NVMe driver), which has been designed and optimized following the traditional assumption that processing cores deliver higher performance than any peripheral I/O device [1].

Optimizing the I/O software stack inside the kernel is a specialized task, requiring OS expertise and understanding the different layers of this stack. For example, an I/O transaction from the application user space to a block device, includes multiple steps of delegating or copying data throughout stack (e.g. from user to kernel memory and from kernel memory to disk). In addition, following the reverse path of an I/O request, the hardware device should also communicate with the processing cores to mark successful completion of operations, or to notify about specific failures.

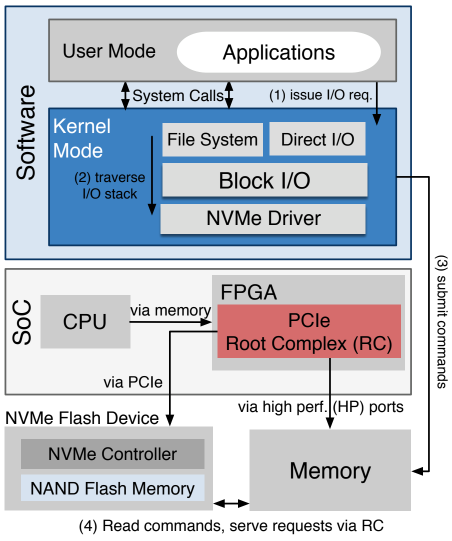
In parallel to the deployment of NVMe, the adoption of FPGAs in data centers is creating opportunities to accelerate various applications or OS functionality [2]. While FPGAs in data centers have been traditionally connected via PCIe to mostly x86 servers, there is also a trend of heterogeneous SoCs with multi-cores and FPGAs integrated on the same die and connected by an on-chip interconnect (e.g. Xilinx Zynq family and Intel/Altera Stratix 10 SoC, including future products prototyped by Intel HARP using Xeon class multi-cores and facilitated by EMIB).

Given these trends, recent work [3], [4] has started to investigate how to combine FPGA-based accelerators with Flash storage, to improve both energy efficiency and performance.

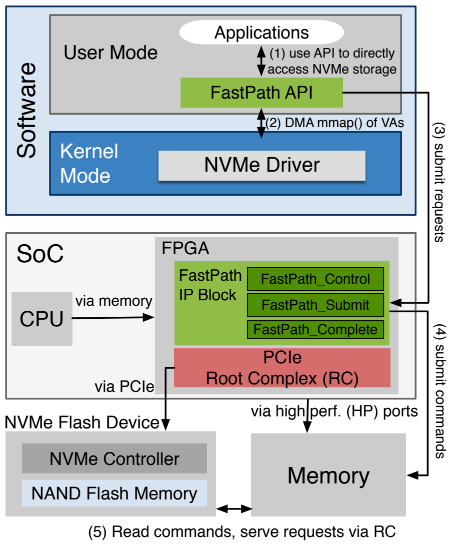
A natural question that arises regarding these heterogeneous SoCs is whether we can use the integrated FPGA not only for accelerating various applications, but also for accelerating OS functionality [5]. *And if so, how could we combine and delegate both acceleration and OS functionality onto the FPGA fabric bypassing the multi-core processing system?*

This paper addresses this research question by introducing FastPath. FastPath offloads the existing slow-performing operations of the I/O software stack of the Linux kernel onto the FPGA, pursuing a direct wire-speed link between applications and NVMe SSDs. Hence, we manage not only to achieve performance improvements, but to potentially enable "on-device-edge" computations where data can be streamed and processed from/to storage without any processing core intervention. This paper makes the following contributions:

- Analyzes the performance inefficiencies of the Linux software I/O stack on an ARM-FPGA SoC and describes how they affect NVMe performance.
- Describes FastPath, our low overhead solution for accelerating I/O operations by eliminating the slow I/O interactions.
- Details the prototype implementation of FastPath on an ARM-FPGA SoC.
- Evaluates FastPath against standard I/O benchmarks showcasing up to 5x performance increase and 71%



(a) The Baseline System.



(b) The FastPath System.

Figure 1: The NVMe architecture on SoCs with integrated FPGAs.

reduction in total latency.

The paper is organized as follows: Section II describes the I/O software stack for NVMe block devices analyzing each layer of the stack. Section III presents the FastPath architecture of the NVMe system running on an ARM-FPGA system. Section IV describes the evaluation methodology and experimental results. Finally, Sections V and VI present the related work and the concluding remarks of this paper, respectively.

II. The NVMe System Architecture

NVMe is a recent SSD standard [6], [7] designed to leverage the high throughput PCIe interconnect enabling scalable solutions in highly demanding systems such as data centres. Data centres, typically based on server aggregation and virtualization, aim for efficient resource utilization by multiplexing several virtual instances on the same physical servers. Unlike the conventional time sharing techniques, the NVMe standard follows a different approach allowing up to 64K pairs of hardware queues for accessing the block device (by sending I/O requests to the device disk controller). Furthermore, the submission/completion scheme of NVMe requests is configurable and allows multiple cores to share one completion queue, leading to more efficient memory utilization [8]. Although the default NVMe architecture includes up to 64K pairs of submission and completion queues, several research papers propose isolation schemes of write and read queues in order to improve performance [9].

The NVMe architecture is a two-layer architecture in which a software stack processes user requests and submits them to the storage device via the NVMe driver. Figure 1a depicts the baseline NVMe architecture highlighting the most important parts of each layer. At the software layer, user applications, through the File System or Direct I/O, submit requests (Step 1) to the Block I/O which end up at the NVMe driver (Step 2). The driver, in turn, retrieves those requests, creates corresponding NVMe commands and submits them to the controller in order to be served (Step 3). The controller, through memory via the Root Complex (RC) IP of the FPGA, reads the commands from predefined memory addresses and serves the requests (Step 4). The predefined memory locations are allocated during initialization, where the NVMe driver creates in-memory queues from which the controller can fetch commands. In addition to a pair of administrative queues, the driver creates N queue pairs, where N equals the number of cores of the system. Each individual pair consists of a submission and a completion queue where submission and completion commands are respectively enqueued. Finally, the design of this system aims high scalability as all cores are able to process I/O requests in parallel.

The I/O software stack executes on the processing cores of the system, and communicates with the controller through the physical PCIe root port (in our system the PCIe root port resides in the FPGA). Accordingly, the controller uses the PCIe root port to write completion responses to the main memory.

A. *Block I/O Overview*

The Linux Operating System (OS) manages the NVMe SSDs as any other block device in the system. Therefore, the associated Linux device driver is developed with respect to the I/O software stack. Any application (e.g. dump disk

-dd, or a user program) can read/write data from/to the disk by using either a File System or directly through a system call (e.g. ioctl, read, write).

Typically, applications run in user mode, which is a system mode with limited permissions compared to the OS. On the contrary, OS kernels are built in an administrative mode that enables them to access and administrate all system devices. Therefore, every OS, including Linux, offers various system call functions that allow them to perform device operations on behalf of the requesting applications. Moreover, several Application Programming Interfaces (APIs) have been used as abstraction layers to perform the user level requests to the OS kernel. The file system is such a layer in the storage system, allowing any application to access data within a block device without interacting directly with the physical sectors and logical block addresses.

The Block I/O (BIO) subsystem is a software layer in the Linux OS that performs I/O requests onto block devices [10]. In essence, it is responsible for communicating and manipulating I/O requests from applications to the storage devices providing a single entry point from the upper levels to the lower levels of the software stack. In particular, BIO provides error handling, profiling, and fair scheduling in order to improve performance. Finally, following the trend of modern multicore systems, the BIO subsystem has been also evolved to support multiple request queues [11]. This, in turn, increased the potential of emerging storage systems that leverage the PCIe interface; previously limited by the legacy kernel I/O stack.

B. NVMe Driver and Controller

The NVMe driver is the software layer between the BIO layer and the NVMe controller, which is responsible for synthesizing NVMe commands for each incoming I/O request from the BIO layer.

The NVMe driver is using a number of pages (Physical Region Pages - PRP) of the system memory to write or read data to and from the disk, respectively. Once the PRPs are successfully mapped, the NVMe command is ready to get synthesized. Besides the PRPs, an NVMe command is comprised of the starting logical block address (slba) which is the logical location within the block device, the length of the current transaction, and an opcode (divided into administrative and I/O).

After successfully locking the I/O queue, a pending command is copied into the I/O queue's memory, and the tail of the queue is written to the doorbell register, as a way to inform the NVMe controller. After *ringing the bell*, the block device starts processing the I/O request and the NVMe driver starts polling (unless it is interrupt driven) the corresponding completion queues, in a round robin way. Once a successful response arrives in the completion queue, the driver unmaps the allocated memory and returns a successful message to the BIO layer.

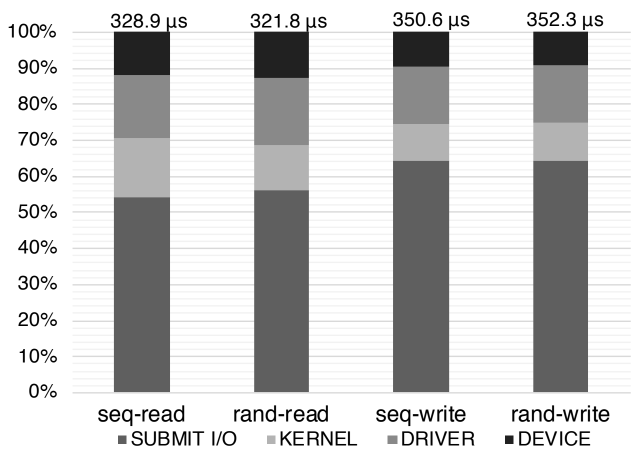


Figure 2: The average latency of the baseline system for 4KB block size in microseconds (μs).

The NVMe controller resides in the device and is responsible for the correct operation of the Flash drive. Operations performed by the controller include:

1) Queue structure configuration.
2) Execution of the submitted commands.
3) Translation of the logical block address to the physical block address in the Flash memory.
4) Error detection of the cells in the Flash drive.
5) Sending a response back to the driver for the successful or unsuccessful completion of the requested operation.

C. Performance limitations

The current Linux I/O software stack adds extra overhead when accessing block devices (e.g. NVMe drives) due to the numerous software layers that submission and completion commands must follow (including a number of expensive system calls). Figure 2 provides an insight of this overhead as measured on the Xilinx Zynq-7000 SoC, that consists of two ARM Cortex A9 processors and an FPGA, running the standard Flexible I/O (*fio*) benchmark [12] (typically used in assessing I/O performance).

As shown in Figure 2, for all configurations of read/write and sequential/random operations, the actual time of the device latency accounts only for 9-10% of the total latency. This means that up to 91% of the total execution time of a single I/O request is spent in submitting and completing the I/O request. Excluding the time spent in the device, the remaining time is distributed amongst the SYSCALL (SUBMIT I/O) to the BIO layer, the processing taking placing at the BIO layer in the kernel (KERNEL), and the device driver (DRIVER). Although, in our platform (32bit ARM Cortex A9 processors) the overhead is further exacerbated due to the low IPC, even in higher performing x86 systems this overhead can be up to 50% of the total execution time [13].

Table I: The FastPath API.

API	Description
`fastpath *fp= fastpath_alloc(int size);`	Allocates disk space, maps a DMA buffer into the application's virtual address space, and returns a *fastpath* object.
`fastpath_write(fastpath *fp, void *buffer, int size, FP_FLAGS);`	Sends an I/O write request.
`fastpath_read(fastpath *fp, void *buffer, int size, FP_FLAGS);`	Sends an I/O read request.
`fastpath_polling(fastpath *fp);`	Blocks until all requests are fulfilled.
`fastpath_free(fastpath *fp);`	Releases the allocated disk space.

III. FastPath: An NVMe FPGA-accelerated system

As demonstrated in Section II-C, the I/O software stack servicing NVMe requests adds a significant performance overhead on modern SSDs. Therefore, bypassing or accelerating this stack is of paramount importance in achieving wire-speed performance.

FastPath, our novel NVMe FPGA-accelerated system, attempts to solve this problem by: 1) bypassing the Linux software stack by designing and exposing an API to the user level that enables applications to place NVMe requests directly to the FPGA, and 2) implementing the `BIO` and driver logic into the FPGA for acceleration, achieving a near wire-speed performance from the time an NVMe request is received up until it is fulfilled.

FastPath has been designed with the following main objectives:

1) Avoid any data copying which can decrease the efficiency of the system.
2) Minimize the total latency of the baseline system by bypassing completely the Linux kernel.
3) Preserve compatibility, by allowing the baseline system to co-exist with the FPGA FastPath system.
4) Accelerate the submission and completion paths by offloading the functionality to the FPGA, bypassing completely both the Block I/O layer and the driver.
5) Support multi-threading enabling concurrent applications to access the NVMe drive securely.

The current version of FastPath has been designed with raw performance in mind. Therefore, it does not implement yet any filesystem underneath, but focused on a high-performance direct read/write path to the NVMe SSDs in a raw form. This approach is mostly suited for applications that require disk persistence such as in-memory databases. The following subsections describe in detail the various components that comprise FastPath.

Listing 1: The definition of the fastpath struct type

```
typedef struct {
    char *dma_address;
    int fp_id;
    int block_size;
} fastpath;
```

A. Disk Allocation and Release

Applications that require read/write access to the NVMe drive must explicitly request disk space. The `fastpath_alloc` function, shown in Table I, accepts as input parameter the *size* of the requested disk space, and returns a descriptor `fp_id` of type `fastpath` (shown in Listing 1).

The `fastpath` struct is composed of the following three fields:

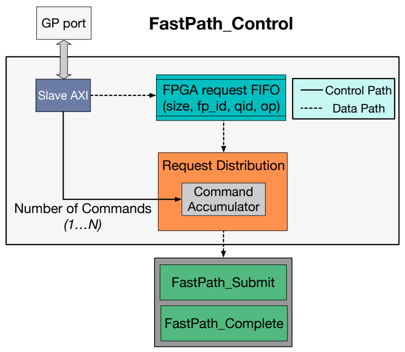
1) `dma_address`: A pointer to the DMA address for direct read/writing to the disk, avoiding any extra data copying.
2) `fp_id`: A unique incremental identifier of the current disk partition.
3) `block_size`: The block size of the requests. Initially this field is empty and is populated by the user prior to calling the FastPath read/write I/O functions. This way we can have a flexible block size per-request.

The `fastpath_alloc` function may fail either due to inability to allocate a logical partition or to communicate with the FastPath IP on the FPGA. In case of failure (denoted by a returned *NULL*), the application has to repeat the same call again. Finally, the release of an allocated partition is performed by the `fastpath_free` function.

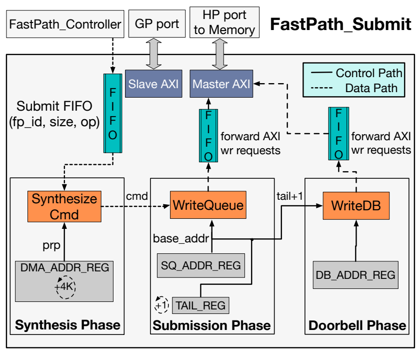
B. Memory Allocation

In contrast to the baseline system, FastPath enables zero copy of data during disk I/O, by mapping DMA buffers into the virtual address space of the applications. Every *FastPath_Submit* module (described later in Section III-D), that resides in the FPGA, has an assigned DMA buffer. For write operations, the data to be written on disk are placed into the DMA buffer, associated with the FastPath IP, and sent to the disk drive. Similarly for read operations, the data fetched from the disk are placed into the DMA buffers where applications can directly access it. In addition, the addresses of the DMA buffers are used as the Physical Region Pages (PRPs) during NVMe command creation.

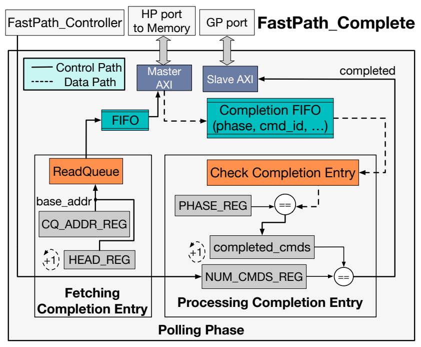
The DMA buffers are pre-allocated in the driver's memory address space during the initialization phase of the device. As shown in Figure 1b (Step 1), when the application calls the `fastpath_alloc` function, the FastPath library internally performs an *mmap* system call (Step 2) to the Linux kernel to map the buffers into the application virtual memory (this is the only system call performed in FastPath).



(a) Design of FastPath_Control module.



(b) Design of FastPath_Submit module.



(c) Design of FastPath_Complete module.

Figure 3: The FastPath IP architecture.

The invoked *mmap* function is a custom implementation inside the NVMe driver that allows us to expose the DMA buffers to the FastPath library. However, this function is never exposed to the user and is called transparently by the FastPath library upon initialization.

C. NVMe Request Submission

In FastPath, the submission of requests to the NVMe device is performed directly to the FPGA, bypassing completely the OS stack (Figure 1b, Step 3). As depicted in Figure 1b, applications can submit requests directly to the FastPath controller in the FPGA, through the `fastpath_read`/`fastpath_write` functions (Table I). These functions accept as arguments: 1) a pointer to the `fastpath` struct, 2) a pointer to the `buffer` where the user's data is read/written from/to, 3) the size of the request, and 4) the `FP_FLAGS`.

Regarding the `FP_FLAGS`, FastPath currently supports direct or indirect calls (`FP_DIRECT` flag) and synchronous or asynchronous requests (`FP_BLOCKING` flag). These two flags can be combined resulting in four different combinations of I/O requests. If the `FP_DIRECT` flag is set, the application can use the DMA buffers directly through the `dma_address` field of the *fastpath* struct. If the `FP_DIRECT` is not set, the FastPath library will internally perform a `memcpy` of the data pointed by `buffer` to the DMA buffers. The `FP_BLOCKING` flag indicates whether the application will block until all requests are completed. If set, the `fastpath_read`/`fastpath_write` functions will not return until all requests are served. Otherwise, they will return immediately leaving to the programmer the ability to inquire about the successful request completion, by using the `fastpath_polling` function.

D. FastPath Logic

As shown in Figure 1b, the FPGA hosts the Xilinx AXI-mapped PCIe Root port, alongside with the FastPath IP block. The FastPath IP block consists of three modules (`FastPath_Submit`, `FastPath_Complete`, and `FastPath_Control`) illustrated in Figure 3. Finally, all hardware blocks have been designed in the Bluespec hardware description language.

The `FastPath_Control` module receives I/O requests from the applications through the FastPath API (Figure 1b, Step 3) and forwards them to the rest of the FastPath modules. The `FastPath_Submit` module is responsible to form the NVMe commands and submit them onto the NVMe submission queue. The `FastPath_Complete`, on the other hand, polls the completion queue to signal the completion of a request as well as reports statistics to the FastPath API regarding the latency of the total request. The submission and completion queues are in memory data structures that hold NVMe commands and completion entries respectively. These queues are accessed by the NVMe controller of the SSD and their corresponding base addresses and doorbells are assigned during the initialization phase of the device driver.

The `FastPath_Control` module, as shown in Figure 3a, holds a request FIFO queue and serves two roles. The first is to accept from the FastPath API NVMe requests and forward them to the `FastPath_Submit` module. This allows us to have multiple FastPath IP blocks (depending on the size of the FPGA) and implement any scheduling algorithms at this stage. The second role is to bookkeep the number of NVMe commands that have to be completed in order to signal a completion message for a specific request (please note that the mapping between NVMe requests to NVMe commands is *1...N* based on the configuration). Every submitted request is forwarded by the `FastPath_Control` module in the *Submit FIFO* of the `FastPath_Submit` module, as shown in Figure 3b. For every forwarded request, the `Command Accumulator` is updated to submit the desired number of commands. Accordingly, the number of pending commands is passed to the *NUM_CMDS_REG* in the `FastPath_Complete` module, as shown in Figure 3c.

The `FastPath_Submit` module manipulates the memory for I/O requests in a 4KB[1] unit size, and performs the following actions:

- **Synthesizes the NVMe commands** based on the arguments of the `fastpath_read` and `fastpath_write` API functions (Synthesis phase in Figure 3b). An NVMe command is a 512-bit wide field that contains information about the submitted operation (e.g. read/write), the DMA address of the data to be processed, the starting logical block address in the volume, and a command identifier. The command identifier is used to indicate a data dependency between two commands. If the device support NUMA-optimized NVMe drivers [8], the command identifier will be combined with the submission queue identifier, to distinguish the commands submitted by different queues.
- **Submits NVMe commands onto a specific NVMe queue stored in memory** (Submission phase in Figure 3b). The tail of the depicted circular queue is stored in a register and is updated according to the actions of the NVMe driver. Whenever the tail reaches the depth size of the queue, it resets to the start.
- **Notifies the NVMe controller about the pending submission** (Doorbell phase in Figure 3b). This is achieved by storing the new tail in the "doorbell" register, which is a memory mapped PCI register (unique for each submission queue).

The `FastPath_Complete` module (Figure 3c) is responsible for:

- Polling the completion queue until the submitted commands of a request have been finished. Similarly to the submission queue, the completion queue is also circular. The head of the completion queue is stored in a register which is updated based on the new completion entries.
- Calculating and reporting time statistics.

Finally, contrary to the conventional system which is using an interrupt-driven mechanism to complete the I/O requests, the FastPath architecture implements a polling completion method for the two following reasons: a) it keeps the processor completely uninvolved in the I/O process, and b) avoids any traffic occurred by the interrupt controllers.

E. FastPath API and Programmability

FastPath has been designed with programmability and portability in mind. Therefore, the software component of the system has been developed as a thread-safe standard C library (*"libfnvme"*) that interfaces with user programs via the API shown in Table I, and has been validated against multithreaded workloads. The library abstracts away implementation details from the user and exposes only a lightweight API that allows easy integration with existing applications in a safe and secure manner (e.g. only 8 lines of code modified in order to run the *fio* benchmark with FastPath).

[1]Please note that the disk block size is device specific; ranging from 512B to 8KBs in our case. FastPath currently supports only 4KB with future plans to extend it to various block sizes.

Table II: System configuration.

Processor Core	Dual-core ARM Cortex-A9 MPCore @667 MHz
FPGA Device	Xilinx Zynq-7000 SoC (Device Name: Z-7035)
L1 Cache	32 KB Instruction, 32 KB Data per processor
L2 Cache	512 KB
On-Chip Memory	256 KB
External Memory	1 GB DDR3
DMA Channels	8 (4 dedicated to Programmable Logic)
PCI Express	Xilinx Root Complex IP operating at 125 MHz (PCIe-Gen2 speeds, up to 8 lanes)
NVMe Storage	Samsung PM953 SSD 480 GB NVMe 1.1 (attached via U.2 connector)
Operating System	Linux Kernel 4.4
fio	fio-2.99

Table III: NVMe I/O request latency on FastPath and baseline systems.

	Latency (μs)			
	seq-read	rand-read	seq-write	rand-write
Baseline	328.9	321.8	350.6	352.3
FastPath	105.4	105.7	100	100.67

IV. Performance Evaluation

We prototyped FastPath on an ARM SoC that consists of a dual-core Cortex A9 MPcore processor and an FPGA. Table II, contains the detailed characteristics of our testbed. The Flexible I/O tester benchmark (*fio*) [12] is used for both the validation and evaluation. In particular, the *libaio* engine is used to generate asynchronous I/O traffic (reads and writes) to the Flash drive. The FastPath architecture is operating at 125 MHz, which is the reference clock frequency generated by the Xilinx Root Complex IP. The Root Complex IP occupies 12.1% of the logic resources of the FPGA, while the FastPath IP block occupies 12.4%. In addition, the FastPath IP block employs 33.7% of the slices as memory.

Please note that we could potentially add multiple FastPath IP blocks. However, the lack of software parallelism (due to having a dual core processor) and the fact that the Root Complex IP has only one port to the memory, would not enable us to perform scalability studies.

Finally, both the baseline system and FastPath are evaluated on the same ARM SoC (Table II) in order to provide a fair comparison and the presented results are the geometric means of 15 iterations over 1 MB[2] of data per configuration.

A. Performance Analysis

1) Latency: The latency is measured as the time spent to submit an I/O request, forward it to the NVMe controller, process it, and complete. In essence, it is the time taken *after* issuing a request until it is completed.

[2]We also experimented with larger data sizes but we did not notice any performance differences.

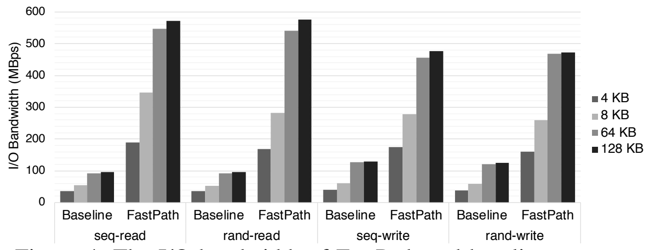


Figure 4: The I/O bandwidth of FastPath and baseline systems.

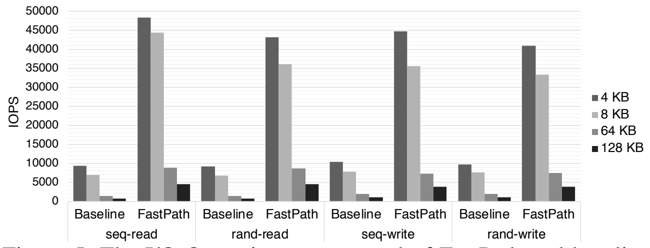


Figure 5: The I/O Operations per second of FastPath and baseline systems.

Table III presents the total latency for all types of operations on 4KB block size requests (averaged over 15 iterations) for both the baseline and FastPath systems. As shown in Table III, FastPath improves the total latency of read and write operations by up to 67% and 71%, respectively. The reason is that FastPath optimizes the submission and completion paths of the baseline system, described in Section II-C, by accelerating them on the FastPath IP block. In the submission path, the `SYSCALL` (SUBMIT I/O) is replaced by a lightweight copy of the request into the `FastPath_Control` module, while in the completion path the `BIO` and the driver layers of the kernel are replaced by the `FastPath_Complete` module. Please note that Table III and Figure 2 show only the latency (submission to completion), excluding the time spent to create I/O requests in order to achieve more precise comparisons.

2) Bandwidth: Figure 4 illustrates the I/O bandwidth for both FastPath and the baseline system. We evaluate the bandwidth for various block sizes, starting from 4KB (equal to the kernel's page size) up to 128KB.

As shown in Figure 4, FastPath achieves higher throughput than the baseline configuration ranging from 160 MBps (rand-write) to 575 MBps (rand-read). Similarly to the speedup results, the increase in bandwidth is due to the elimination of all the expensive system calls to the kernel I/O stack, as well as the acceleration of the creation and issue of the NVMe commands performed on the FPGA.

The maximum advertised bandwidth of the particular SSD we used in our experiments is 1GBps for read operations and 800MBps for write operations [14]. That means that FastPath is currently at 40% of the theoretical maximum bandwidth. The gap in performance is mainly attributed to the slow 32 bit ARM core used during request creation. We tested our hypothesis by implementing a custom microbenchmark that issues pre-generated I/O requests onto the FPGA (bypassing `libaio`), and we managed to achieve I/O bandwidth close to 800 MBps. We anticipate that even with this overhead, when FastPath ported to high end x86 systems [15], as well as when multiple FastPath IP blocks are placed into the FPGA, we will manage to achieve "out-of-the-box" near wire-speed performance.

3) I/O Operations per Second (IOPS): The number of IOPS, a common metric for assessing SSD performance, is the number of block sized I/O requests performed within a second and is strongly connected to both the latency and the block sizes of the system; lower latency leads to more I/O requests served by the system within a time window.

Figure 5 presents the IOPS for all configurations and various block sizes for both FastPath and the baseline system. As shown, FastPath achieves, at least five times more IOPS for read and write operations. Furthermore, we notice that the number of IOPS decreases when the block size increases. This is because the fulfilment of the same data size with larger block sizes can lead to lower number of IOPS since a smaller number of I/O requests are created within a second.

Finally, please note that both the IOPS and the bandwidth are linked to the performance of the processor since the time spent to create I/O requests is factored into the results; as per the *fio* methodology of reporting performance numbers.

V. Related Work

A first group of related work can be classified as *Near Data Processing (NDP)* [16] to FPGA acceleration of large scale data [17], [18]. FastPath differs from the aforementioned approach by targeting OS code, rather than application code and by accelerating I/O requests to NVMe SSDs on FPGAs.

A second category of related work has focused on accelerating the kernel's I/O software stack. For example, Caulfield *et al.* [19] proposed to bypass the `BIO` layer and implemented a separate driver, as a way to increase performance. Lee *et al.* [9] presented a novel queue isolation scheme, considering the write interference and increasing the read performance in heavy read workloads. Bjørling *et al.* [11] redesigned the `BIO` layer in order to enable scalability and exploit the spare hardware capabilities by implemented multi-queue support. Nonetheless, FastPath differs from these approaches by bypassing completely the kernel's software I/O stack and offloading the functionality directly onto the FPGA, taking advantage of modern SoCs.

Finally, a last group of related work has exposed APIs, aiming to enable direct access to the NVMe from the user space [13], [20], [21]. Although these approaches manage to bypass a significant part of the kernel's software stack, extra user space code is required in order to perform various functionalities such as command creation and scheduling.

FastPath, on the other hand, attempts to offload and accelerate the whole process onto the FPGA, which not only improves performance but also enables the implementation and pipelining of extra functionalities (e.g. encryption, decryption, sorting, user defined functions) directly into the data path between the disk drive and memory, bypassing completely the CPU.

VI. Conclusions

In this paper we introduced FastPath: an FPGA-based solution for accelerating NVMe-based SSDs. After we illustrated the performance bottlenecks of current kernels' I/O software stacks, we presented the design and our first prototype implementation. The evaluation of FastPath against standard I/O benchmarks demonstrated up to 5x higher IOPS and up to 71% lower latency than the baseline implementation.

In the future we plan to investigate user defined functionality on the FPGA. At the moment FastPath supports raw I/O transactions since the focus of this paper is to examine the I/O performance and not the file system functionality. We plan to investigate the best match between file systems and FastPath.

Acknowledgments

A. Stratikopoulos is funded by an ARM Ltd. & EPSRC iCASE PhD Scholarship.

References

[1] M. Nanavati, M. Schwarzkopf, J. Wires, and A. Warfield, "Non-volatile storage," *Commun. ACM*.

[2] D. Andrews, W. Peck, J. Agron, K. Preston, E. Komp, M. Finley, and R. Sass, "hthreads: a hardware/software co-designed multithreaded rtos kernel," in *2005 IEEE Conference on Emerging Technologies and Factory Automation*, 2005.

[3] L. Woods, Z. István, and G. Alonso, "Ibex: An intelligent storage engine with support for advanced sql offloading," *Proc. VLDB Endow.*, 2014.

[4] S. Xu, S. Lee, S.-W. Jun, M. Liu, J. Hicks, and Arvind, "Bluecache: A scalable distributed flash-based key-value store," *Proc. VLDB Endow.*, 2016.

[5] J. A. Stankovic and R. Rajkumar, "Real-time operating systems," *Real-Time Syst.*, 2004.

[6] A. Huffman, "NVM Express Revision 1.1." http://www.nvmexpress.org/wp-content/uploads/NVM-Express-1_1.pdf, 2012.

[7] "NVM Express Explained," http://nvmexpress.org/wp-content/uploads/2013/04/NVM_whitepaper.pdf, 2012.

[8] K. Marks, "An NVM Express Tutorial," https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2013/20130812_PreConfD_Marks.pdf, 2013.

[9] M. Lee, D. H. Kang, M. Lee, and Y. I. Eom, "Improving read performance by isolating multiple queues in nvme ssds," in *Proceedings of the 11th International Conference on Ubiquitous Information Management and Communication*, ser. IMCOM '17.

[10] J. Axboe Suse, "Linux block iopresent and future," 2004.

[11] M. Bjørling, J. Axboe, D. Nellans, and P. Bonnet, "Linux block io: Introducing multi-queue ssd access on multi-core systems," in *Proceedings of the 6th International Systems and Storage Conference*, ser. SYSTOR '13.

[12] Jens Axboe, "Flexible I/O," https://github.com/axboe/fio.

[13] H.-J. Kim, Y.-S. Lee, and J.-S. Kim, "Nvmedirect: A user-space i/o framework for application-specific optimization on nvme ssds," in *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, 2016.

[14] Lee Won-ju, Chang Jong-baek, Song Sang-hoon, Kim Jaen-eun, Lee Sang-geol, Koh Seung-wan, Park Hae-sung, Kin Sung-wook, Jang You-jin, Na You-jung, Choi Young-gil, "SM953 White Paper The Ultimate NVMe SSD for Data Center," https://s3.ap-northeast-2.amazonaws.com/global.semi.static/SM953_Whitepaper-0.pdf, 2014.

[15] Q. Xu, H. Siyamwala, M. Ghosh, T. Suri, M. Awasthi, Z. Guz, A. Shayesteh, and V. Balakrishnan, "Performance analysis of nvme ssds and their implication on real world databases," in *Proceedings of the 8th ACM International Systems and Storage Conference*, ser. SYSTOR '15.

[16] B. Gu, A. S. Yoon, D.-H. Bae, I. Jo, J. Lee, J. Yoon, J.-U. Kang, M. Kwon, C. Yoon, S. Cho, J. Jeong, and D. Chang, "Biscuit: A framework for near-data processing of big data workloads," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16.

[17] S.-W. Jun, M. Liu, S. Lee, J. Hicks, J. Ankcorn, M. King, S. Xu, and Arvind, "Bluedbm: An appliance for big data analytics," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA '15.

[18] S. Seshadri, M. Gahagan, S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, and S. Swanson, "Willow: A user-programmable ssd," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14.

[19] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson, "Moneta: A high-performance storage array architecture for next-generation, non-volatile memories," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '43, 2010.

[20] S. Lee, M. Liu, S. Jun, S. Xu, J. Kim, and Arvind, "Application-managed flash," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*, ser. FAST'16. USENIX Association.

[21] Z. Yang, J. R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, V. Verma, and L. E. Paul, "Spdk: A development kit to build high performance storage applications," in *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2017.