FastPath_MP: Low Overhead & Energy-efficient FPGA-based Storage Multi-paths

ATHANASIOS STRATIKOPOULOS, CHRISTOS KOTSELIDIS, JOHN GOODACRE, and MIKEL LUJÁN, University of Manchester, United Kingdom

In this article, we present FastPath_MP, a novel low-overhead and energy-efficient storage multi-path architecture that leverages FPGAs to operate transparently to the main processor and improve the performance and energy efficiency of accessing storage devices. We prototyped FastPath_MP on both Arm-FPGA Zynq 7000 SoC and Zynq UltraScale+ MPSoC and evaluated its performance against standard microbenchmarks as well as the real-world in-memory Redis database. Our results show that FastPath_MP achieves up to 82% lower latency, up to 12× higher throughput, and up to 10× more energy efficiency against the baseline storage path of the Linux kernel.

CCS Concepts: • **Computer systems organization** → **System on a chip**; • **Hardware** → **External storage**;

Additional Key Words and Phrases: NVMe, SSDs, block I/O, storage path, FPGA, system on a chip

ACM Reference format:

Athanasios Stratikopoulos, Christos Kotselidis, John Goodacre, and Mikel Luján. 2020. FastPath_MP: Low Overhead & Energy-efficient FPGA-based Storage Multi-paths. *ACM Trans. Archit. Code Optim.* 17, 4, Article 37 (November 2020), 23 pages.

<https://doi.org/10.1145/3423134>

1 INTRODUCTION

During the last decades, the performance of processors as well as the number of transistors in dense integrated circuits have been doubling every two years [29]. Following the stall of Moore's law, computer systems initially became multi-cores and lately heterogeneous in an attempt to keep pushing the performance boundaries. The emergence of heterogeneous computer systems enabled simpler processing cores to be combined with hardware accelerators, such as *GPUs* and *FPGAs* [16]. While accelerators have been mostly connected via PCIe to x86 computer systems, a new trend has recently emerged where System-On-Chips (SoCs) integrate multicore CPUs and FPGA accelerators on the same die via an on-chip interconnect (e.g., Xilinx Zynq family and Intel/Altera Stratix 10

37

This work was partially supported by EU Horizon 2020 grants: (i) EuroEXA (ID: 754337); and (ii) E2Data (ID: 780245) with hardware platforms from Kaleao Limited. A. Stratikopoulos was funded by an Arm Ltd. & EPSRC iCASE PhD Scholarship during this work. Prof. Mikel Luján is funded by an Arm/RAEng Research Chair Award and a Royal Society Wolfson Fellowship.

Authors' addresses: A. Stratikopoulos, C. Kotselidis, J. Goodacre, and M. Luján, Department of Computer Science, University of Manchester, Manchester, United Kingdom; emails: {athanasios.stratikopoulos, christos.kotselidis, john.goodacre, mikel.lujan}@manchester.ac.uk.

This work is licensed under a Creative Commons Attribution International 4.0 License.

© 2020 Copyright held by the owner/author(s).

1544-3566/2020/11-ART37

<https://doi.org/10.1145/3423134>

SoC, including future products prototyped by Intel-Altera *HARP* using Xeon class multi-cores and facilitated by the *EMIB* [28]).

In tandem with the deployment of the FPGA technology in datacenters [7, 32], the advent of faster storage technologies, such as *solid state drives (SSD)*, has enabled storage devices to perform at higher I/O rates and constituted established legacy interfaces (e.g., *SCSI*, *SATA*, and *SAS*) as insufficient [39]. Consequently, storage vendors standardized the *Non-Volatile Memory Express (NVMe)* [2] specification, which leverages high throughput interconnect (i.e., *PCIe*) providing significant I/O performance improvements against conventional SSD devices [39].

Computer systems require data to move from storage devices to main memory and vice versa, for processing. Depending on which level of the memory hierarchy data is accessed, latency can vary significantly. For instance, the latency for accessing data stored in the processor's registers is less than 1 ns (1 clock cycle), whereas the latency for accessing data in the cache memory can be up to 10 ns; and in the main memory up to 100 ns. However, advancements in non-volatile memory (i.e., SSDs) exhibit latencies that range from 10 μ s to 115 μ s, while the latency of legacy magnetic disks is in the scale of milliseconds. Although SSDs have become ubiquitous in data centers, significant performance overheads still exist that need to be tackled. The primary reason is that computer systems still need to traverse multiple software layers in the Operating System (OS), requiring the processor's cores at first to perform the context switch and then start executing instructions on behalf of the OS to access the storage devices; a process that significantly impacts the latency [23, 41, 42].

This article aims to exploit the FPGA technology to mitigate the aforementioned latency gap towards achieving wire-speed storage access and scalable performance. This article extends our previous work [37] and makes the following contributions:

- It introduces FastPath_MP, a novel system architecture that exploits multiple parallel FPGA-based paths to the NVMe SSDs, resulting in scalable performance.
- It describes how FastPath_MP achieves compatibility and interoperability with the traditional Linux storage path.
- It evaluates FastPath_MP in two classes of Arm-FPGA SoCs against the baseline storage path of the Linux kernel and the FastPath architecture [37] over standard microbenchmarks and the Redis in-memory database. Our results showcase up to 82% lower latency, up to 12 $\times$ higher throughput, and up to 10 $\times$ more energy efficiency.

2 THE NVME SYSTEM ARCHITECTURE

The NVMe system architecture is a two-layer architecture built at the top of the *PCIe* interconnect. The first layer refers to a software stack, which processes user requests and submits them to the storage device via the NVMe driver. The second layer comprises the hardware components in the NVMe SSD (e.g., NVMe controller), which are responsible for dispatching requests from the NVMe driver and executing them on the storage device. The software layers are executed on the cores of the main processor in the system, whereas the storage drive is connected to the host system via the *PCIe Root Complex*. The *PCIe Root Complex* is a hardware unit that delivers:

- *PCIe* operations from the NVMe driver to the NVMe controller via memory.
- Read or write requests from the NVMe controller back to memory.
- Completion responses from the NVMe controller back to NVMe driver via memory.

2.1 The NVMe Software Layer

Figure 1 depicts the NVMe architecture highlighting the most critical parts in the system. At the software layer, user-level applications submit requests (Step 1) to the Block I/O that end up in

Fig. 1. The I/O request path on the NVMe system.

the NVMe driver (Step 2) through the File System or Direct I/O. The driver, in turn, retrieves those requests, creates corresponding NVMe commands, and writes them to memory (Step 3). The NVMe controller reads the commands via the PCIe subsystem and serves the associated I/O requests (Step 4).

2.1.1 The File System & Direct I/O Layer. The Linux 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`).

The file system is an abstraction layer that allows any application to access data within a block device without interacting directly with the physical elements in the storage device, such as sectors and logical block addresses. Data are packed in groups called *files* and stored in a specific logical block address, from which it can be easy to retrieve. The file system includes data structures (e.g., *inode*) that store the logical block addresses along with some information (i.e., access permission, file descriptor, file ownership, and file type) associated with any file in the file system.

The Linux OS uses the technique of “*buffering*,” which stores data temporarily within buffers (in kernel memory address space) to increase the I/O performance and then copies the buffer contents in the block device. Due to this technique, the successful execution of a `write` system call does not necessarily mean that the data are implicitly written in a block device. Applications that require the write operations to avoid any intermediate copies from the applications’ memory to the kernel buffers or the read operations to access the block device rather than hit the memory cache can use the *Direct I/O* feature of a file system.

2.1.2 The Block I/O Layer. The Block I/O (BIO) subsystem is a software layer in the Linux OS that performs I/O requests onto block devices [19]. 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.

Fig. 2. The block I/O multiple queue scheme.

Following the trend of modern multi-core systems, the BIO subsystem has evolved to support multiple request queues, which created an opportunity for latest storage systems to leverage the PCIe interface; previously limited by the legacy kernel I/O stack [5]. Figure 2 depicts the multiple queue block I/O scheme including two levels of queues (in the kernel address space): the software staging queues and the hardware dispatch queues. The software staging queues can be configured to be one per socket or one per core, whereas the number of hardware dispatch queues matches the amount of storage device queues.

BIO receives I/O requests from the processor’s cores on behalf of a user-level application and adds these requests on the software dispatch queues (staging). After staging the requests, BIO may perform I/O scheduling [17], including merging I/O requests or forwarding I/O requests with high priority level to be executed quickly. Afterwards, the requests are dispatched to a hardware dispatch queue based on the queue mapping stage. The mapping between the two levels of queues allows BIO to operate even when the number of software staging queues and hardware dispatch queues differ. A hardware dispatch queue collects multiple I/O requests targeting a specific storage queue and forwards them serially to the NVMe driver for execution. After the fulfillment of a request, the NVMe driver notifies BIO, which consequently returns the completion status (success/error) to the user applications.

2.1.3 The NVMe Driver. The NVMe block device is a “hot plug” component in the Linux OS, since it can be inserted and removed to a running computer system without requiring any significant interruption. Thus, the Linux OS *probes* the device and invokes the NVMe device driver to perform the initialization phase. During the initialization phase, the NVMe driver allocates specific data structures to establish the communication between the processor and the NVMe controller. In the default configuration, the allocated data structures have the form of in-memory circular queue pairs (i.e., submission, completion) with 64 KB size, as shown in Figure 3. The number of queue pairs is equal to the number of physical cores of the processor to avoid synchronization issues

Fig. 3. The data flow in the NVMe driver.

and ensure that queue entries are not dispersed among different core caches [1]. For example, a system with four cores allocates one pair of administrative queues and four pairs of I/O queues. Nevertheless, the submission/completion scheme of the NVMe queues is configurable and can allow multiple cores to share one completion queue, leading to more efficient memory utilization [21].

After the initialization phase, the administrative queues are used to perform the administrative operations (e.g., creation/deletion of an I/O queue pair), while the remaining queues are used to perform I/O transactions on behalf of the respective processing cores. Figure 3 illustrates the hardware flow of the NVMe driver functionality in five steps. At first, it dispatches an I/O request, synthesizes an NVMe command for the request, and submits the command in the submission queue (Step 1). Then the new tail is written in a memory-mapped PCIe register called “doorbell” (Step 2), which notifies the NVMe controller for new submission entries. Thereafter, the NVMe controller starts fetching the submitted commands (Step 3) and processing them (Step 4). After the execution of each command, the NVMe controller issues a completion entry in the completion queue of the same pair of queues from which it fetched the command and triggers an interrupt to notify the associated processing core of the completion (Step 5). Finally, the interrupt is handled by the interrupt handler in the NVMe driver, which analyzes the completion entry and delivers information about the outcome back to the BIO. The interrupt-driven completion method in the NVMe driver can also be replaced by a polling method.

2.2 The NVMe Hardware Components

The NVMe SSD architecture consists of two main components: the *NVMe controller* and the *flash memory* (Figure 4).

2.2.1 NVMe Controller. The NVMe controller is an embedded standard processor in the storage device, and it is responsible for ensuring the correct operation of the driver. Operations performed by the controller include:

Fig. 4. The block diagram of internals of an NVMe SSD [11].

- Establishment of the host interface (i.e., PCIe, SATA, SAS), along with the initialization of the necessary data structures.
- Initialization of the Flash File System (i.e., Wear Leveling, Garbage Collection, Bad Block Management).
- Error correction of the cells in the flash memory.
- Execution of submitted NVMe commands.
- Push-back notifications to the driver for the successful or unsuccessful completion of the requested operations.

2.2.2 Flash Memory. Flash memory is arranged as an array of floating gate transistors called *memory cells*. The placement of the memory cells determines the flash memory as a NAND-based or NOR-based flash memory. These two types of memory have significant performance differences. For example, NOR-based memory performs faster reads than NAND-based memory, while the latter performs significantly faster write operations than the first [33]. This article focuses on NAND-based flash but more information on NOR-based flash can be seen in Reference [13].

Following the advancements of the memory-cache hierarchy in computer architecture, many storage architects have integrated a fast DDR memory next to the flash memory chips to perform data caching as presented in Figure 4. Data caching includes the storage of data in DDR memory via a *DDR controller* before writing data back in the flash memory. In case of overwriting data in the flash memory, the *Garbage Collection* task is performed in the background.

2.3 Performance Limitations

Several studies [37, 39, 42] have shown the impact of the Linux OS storage software stack on modern SSDs. As shown in our previous study [37] that characterizes the NVMe performance on the Xilinx Zynq-7000 SoC, up to 10% of the total latency is spent in the storage device. The remaining time is distributed among the software layers in the Linux OS, such as the SYSTEM CALL, the BIO layer, and the NVMe driver. Although, the software overhead in the Zynq-7000 SoC is further exacerbated due to the low IPC of the 32 bit ARM Cortex A9 processors, even in higher performing x86 systems this overhead can be up to 30% of the total execution time [42].

3 THE FASTPATH_MP ARCHITECTURE

As presented in Section 2.3, the Linux OS storage software stack adds a significant performance overhead when accessing modern SSDs. Prior studies [39, 42], have attributed the software overhead of storage systems to: (i) context switching, (ii) data copy (from kernel to user and back), (iii) interrupt handling, and (iv) lock contention of shared resources (e.g., the I/O request queue). Therefore, bypassing or accelerating this stack creates an opportunity towards achieving wire-speed performance. In previous work, we presented FastPath [37], a direct FPGA-based storage path that accelerates the access to the NVMe SSDs by: (1) bypassing the Linux software stack through the design and exposure of an API to the user level that enables applications to place NVMe requests to the FPGA, and (2) offloading the BIO and driver logic into the FPGA for acceleration, achieving near wire-speed performance.

In this article, we extend the FastPath architecture to leverage the availability of multiple queues from the NVMe specification and exploit the internal parallelism offered by the reconfigurable logic. Hence, the proposed multi-path system, called FastPath_MP, allows multiple FPGA-based fast paths to access the NVMe SSD concurrently.

3.1 FastPath_MP Design Challenges

The design and implementation of storage systems that can offer direct access to the storage media has been recently studied. Unlike prior studies, FastPath_MP showcases the performance improvements of accessing NVMe SSDs from the FPGA accelerator while eliminating any requirement to traverse the traditional software stack running on the main processor. During the design and implementation of FastPath_MP, we encountered the following design challenges:

- (1) **Compatibility with the legacy Linux kernel I/O path:** The extension of new storage I/O paths that can operate in parallel to the traditional Linux software stack is not trivial and requires design choices to preserve compatibility. Section 3.2 presents how FastPath_MP coexists with the legacy Linux kernel I/O path.
- (2) **Design of a scalable FPGA architecture:** The design of an architecture that can utilize the spare storage queues offered by the NVMe specification without adding further overhead. Section 3.3 discusses the proposed FPGA-based architecture that enabled FastPath_MP to achieve scalable performance.
- (3) **Support of multi-threaded applications:** The vast majority of real-world applications spawn multiple threads to harness the current hardware capacity of processors. Therefore, FastPath_MP has been designed to interface with applications in a thread-safe manner. Section 3.4 presents the API that facilitates FastPath_MP to be ported to different applications.

3.2 The FastPath_MP I/O Path Configuration

FastPath_MP is implemented to co-exist with the current Linux kernel I/O path without adding any further overhead. The NVMe system specification provides the *namespace* feature to allow multiple I/O paths to access the NVMe device. Consequently, the proposed system is configured to use a shared *namespace* between the Linux I/O path and FastPath_MP. The following sections present the interface between the system software at the user/kernel level and the FPGA logic (Section 3.2.1), as well as the required modifications to the NVMe driver for enabling FastPath_MP to run as an individual storage path (Section 3.2.2).

3.2.1 FPGA Interface for Applications & Driver. The FPGA device is physically pinned to a specific part of the physical memory that can be accessed from user applications and kernel modules

Fig. 5. The memory mapping of FPGA-based modules into user-level applications and NVMe driver.

if it is memory mapped in an address region in virtual memory, as illustrated in Figure 5. User applications can map physical devices in the virtual memory address space by calling the standard Linux `mmap` system call and passing the file descriptor of the device as a parameter. Likewise, kernel modules map physical devices by calling the Linux kernel `ioremap` function, which accepts as parameters the physical address and size of the memory to map. The return value of both functions is a virtual address that can be stored in a pointer.

During the initialization of `FastPath_MP`, the NVMe driver communicates with the FPGA-based architecture via a memory-mapped interface. At first, the driver maps the physical address of a particular FPGA module such as the `FastPath_Submit` (red dashed arrow in Figure 5). Thereafter, it uses the memory mapped registers of each module in the `FastPath` IP blocks to pass the base address of the DMA regions that are associated with the command queues and the addresses of the corresponding doorbells. Applications can use the `mmap` system call (black dashed arrow in Figure 5) to map the DMA regions, which are allocated for each fast path, to the virtual address space of the applications. 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 application and is transparently called by the `FastPath` library upon initialization.

3.2.2 Linux NVMe Driver Modifications. The driver modifications presented below are executed during the probing phase when the NVMe driver allocates the NVMe queues to establish the communication between the main processor and the NVMe controller (Section 2.1.3). In detail, `FastPath_MP` applies the following driver modifications:

- (1) **The number of the NVMe queue pairs** is equal to the physical processor's cores of the system. Considering that the maximum allowed number of pairs in the NVMe specification is 64K, the Linux NVMe driver is modified to allocate extra number of queue pairs for `FastPath_MP`.

Fig. 6. The I/O request path on the FastPath_MP NVMe system.

- (2) **The physical region pages (PRPs)** are built as DMA memory regions allocated with a maximum size of 64 MB. Every FastPath_MP submission queue uses one DMA memory region as a PRP field in NVMe commands. The DMA physical address is stored in a register in the FastPath_Submit module via the memory mapped interface and can be reused once the submission queue is idle.
- (3) **The submission queue address** is stored in a register in the FastPath_Submit module via the memory mapped interface, as shown in Figure 5. This is the submission base address used for submitting NVMe commands.
- (4) **The submission doorbell address** is stored in a register in the FastPath_Submit module via the memory mapped interface. This is the *doorbell* base address used for writing the new submission queue tail, thereby notifying the NVMe Controller for the new submitted commands in the submission queue.
- (5) **The completion queue address** is stored in a register in the FastPath_Complete module via the memory mapped interface. This is the completion base address used for polling the completion queue entries.

3.3 FastPath_MP Logic

The FastPath_MP architecture extends the FastPath architecture to submit concurrent NVMe commands by using multiple parallel NVMe queues. Figure 6 presents the FastPath_MP architecture designed to accommodate four fast paths, due to the four parallel memory ports available on the hardware board. The FastPath_Submit and FastPath_Complete logic of each fast path is an exact replica of the FastPath IP block [37].

As shown in Figure 6, applications can use the FastPath API (Step 1) to perform an *mmap* system call (Step 2) to the Linux kernel. As discussed in Section 3.2.1, this system call will map the DMA buffers into the applications' virtual memory space. Step 3 shows that the requests are submitted to the FastPath_Control_MP module, which is responsible for distributing concurrent multiple requests for submission. This module is the primary extension to the FastPath architecture,

Fig. 7. Design of FastPath_Control_MP module.

enabling FastPath_MP to achieve performance scalability due to the FPGA-based parallel fast paths (FP_0 - FP_3) that can simultaneously submit requests (Step 4). Finally, the NVMe controller in the SSD reads and serves the submitted requests from FastPath_MP. Upon completing the requests, the FastPath_Complete modules of each fast path collect and forward the state of the service to the FastPath_Control_MP, which in turn returns this information to the applications through the FastPath API. The programming model of FastPath_MP along with the FastPath API are discussed in Section 3.4.

3.3.1 FastPath_Control_MP Module. The FastPath_Control_MP module accepts NVMe requests from the FastPath API and forwards them for submission. In addition, it maintains the number of NVMe commands that have to be completed. The FastPath_Control_MP module can forward multiple I/O requests for submission, as shown in Figure 7. The module retrieves requests out of the request queue and distributes them based on the incoming request size. Thus, in case a request has 8 KB size, the FastPath_Control_MP module will forward two parallel requests in the two assigned FastPath_Submit modules. For every forwarded request, the associated Command Accumulator block is updated to submit the desired number of commands. Accordingly, the number of commands to be completed is passed to the respective FastPath_Complete module. Considering that the smallest block size that the FastPath architecture supports is 4 KB, the maximum request size that can be served simultaneously by the FastPath_Control_MP module is 16 KB.

3.3.2 Breakdown Analysis. Figure 8 presents the distribution of the latency amongst various layers when running the standard *Flexible I/O (fio)* benchmark [18] for a single I/O request with 4 KB block size. In particular, Figure 8(a) shows the breakdown of the latency across the Linux OS layers on the Xilinx Zynq-7000 SoC, as presented in our previous work [37]. As shown in Figure 8(a), up to 10% of the overall latency is spent in the device, while the remaining 90% is consumed in the SYSCALL (SUBMIT I/O) to the BIO layer, the processing at the BIO layer in the kernel (KERNEL), and the NVMe driver (DRIVER). However, Figure 8(b) presents the breakdown of the latency for a single I/O request that is served by one of the FPGA-based parallel fast paths (FP_0-FP_3). In this case, the system call for submission is replaced by a call to the FastPath API that submits the I/O request to the FastPath_Control_MP module (Figure 6, Step 3). The I/O request is initially distributed by FastPath_Control_MP to a single FastPath IP block that, in turn, creates the I/O commands and submits them to an NVMe queue (Figure 6, Step 5). This process along with the initiation of the polling mechanism that monitors the completion of the request (Figure 6, Step 6) are reported as FastPath in Figure 8(b).

Fig. 8. Breakdown latency for a 4 KB I/O request on Zynq 7000 SoC.

Table 1. The FastPath API

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

Thus, FastPath can improve the overall latency for read and write operations by up to 67% and 71%, respectively. The reasons behind the reduction in latency are: (i) the replacement of the system call with the FastPath API (ranging from 75% up to 80% reduction); and (ii) the fact that the FastPath IP block creates and processes the I/O requests up to 80% faster than the Linux I/O path. Section 4 presents an analysis of the Linux I/O path versus the scalable FastPath_MP architecture in terms of performance, power, and area.

3.4 FastPath_MP Programming Model

FastPath_MP has been designed for achieving high raw performance and thus it does not implement any filesystem underneath. Instead it is focused on a high-performance direct read/write path to the NVMe SSDs in a raw form. This approach is most suited for applications that require disk persistence such as in-memory databases [23]. For simplicity, this section discusses the programmability of FastPath_MP that is described in more detail in Reference [37].

3.4.1 FastPath_API. 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-level programs via the API shown in Table 1, and it 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 safely and securely. For example, only eight lines of code were modified to run the *fio* benchmark with FastPath.

3.4.2 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 1, accepts as input parameter the *size* of the requested disk space and returns a descriptor `fp_id` of type `fastpath` (shown in Listing 1).

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

Listing 1. The definition of the `fastpath` struct type.

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

- (1) `dma_address`: A pointer to the DMA address for direct read/write 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 has the 4 KB as default value but can be updated by the user before 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. Finally, the release of an allocated partition is performed by the `fastpath_free` function.

3.4.3 Memory Allocation. In contrast to the baseline system, FastPath_MP enables zero copy of data during disk I/O by mapping DMA buffers into the virtual address space of the applications. Applications can submit a write/read request to the NVMe SSD by using the `fastpath_write/fastpath_read` functions provided by the FastPath_API. These functions create I/O requests to the FastPath_Control_MP module that resides in the FPGA. Applications can use the available `FP_FLAGS` to indicate information about the I/O request. Regarding `FP_FLAGS`, FastPath_MP supports direct or indirect calls (`FP_DIRECT` flag) and synchronous or asynchronous requests (`FP_BLOCKING` flag).

For example, an application can use the `FP_DIRECT` flag to inform the “libfnvme” library to submit a request for data that will be written in the `dma_address` field of the `fastpath` struct. If this flag is not set, the “libfnvme” library internally copies the data pointed by the `buffer` parameter to the DMA buffers. Every *FastPath_Submit* module that resides in the FPGA has an assigned DMA buffer. For the `fastpath_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 the `fastpath_read` read operations, the data fetched from the disk are placed into the DMA buffers, which can be directly accessed by applications. The DMA buffers are pre-allocated in the driver’s memory address space during the initialization phase of the device, and they are used as the Physical Region Pages (PRPs) during the creation of the NVMe commands. As shown in Figure 6 (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).

Table 2. Experimental Testbeds

Characteristics	Xilinx Zynq-7000 SoC	Xilinx Zynq UltraScale+ MPSoC
Processor Core	Dual-core Arm Cortex-A9 MPCore @667 MHz	Quad-core Arm Cortex-A53 MPCore @1.5 GHz
FPGA Device	Xilinx Zynq-7000 SoC (Device Name: Z-7035)	Xilinx Zynq UltraScale+ MPSoC (Device Name: ZU9EG)
L1 Cache	32 KB Instruction, 32 KB Data per core	32 KB Instruction, 32 KB Data per core
L2 Cache	512 KB	1 MB
On-Chip Memory	256 KB	256 KB
External Memory	1 GB DDR3	4 GB DDR4
DMA Channels	8 (4 dedicated to Programmable Logic)	8
PCI Express	Xilinx Root Complex IP operating at 125 MHz	Arm PCIe Root Port
	(PCIe-Gen2 speeds, 4 lanes)	(PCIe-Gen2 speeds, 1 lane)
NVMe Storage	Samsung PM953 SSD 480 GB NVMe v1.1 (attached via U.2 connector)	
Operating System	Linux Kernel 4.4	Linux Kernel 4.9

Table 3. The Experimental Systems

System	Description
Baseline	The unmodified Linux system that uses the mainline kernel I/O path.
FastPath [37]	Includes the FastPath controller that dispatches I/O requests to one fast path.
FastPath_MP_p	Uses p parallel fast paths similar to FastPath, interfacing with FastPath_Control_MP module.

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 the programmer the ability to inquire about the successful request completion by using the `fastpath_polling` function.

4 EVALUATION

4.1 Experimental Methodology

We evaluate FastPath_MP against the *fio* benchmark and the Redis Key-Value-Store (KVS) database on the Xilinx Zynq-7000 SoC and the Zynq UltraScale+ MPSoC, respectively, (Table 2 presents the hardware and software configurations of the evaluated platforms). Additionally, the presented results are the arithmetic means of 15 iterations; a sufficient number to indicate the standard deviation of our measurements.

We employ the widely used [4, 6, 23, 25, 39, 42] Flexible I/O tester (*fio-2.99*) [18] to characterize the performance and latency of the systems presented in Table 3. We configure *fio* to use the *libaio* Linux library to generate asynchronous I/O traffic for both read and write requests to the NVMe drive. In the baseline system, the I/O requests traverse the Linux software layers, whereas in the FastPath and FastPath_MP systems the I/O requests are submitted in the respective FPGA-based architecture with the `FP_DIRECT` and `FP_BLOCKING` flags.

For our second experiment, we used the *Redis* v4.0.8 [3] KVS database to evaluate the highest-performing and energy-efficient configuration of the FastPath_MP architecture derived from the

Fig. 9. The average block latency (in μs) of FastPath, FastPath_MP_p, and baseline systems for 1 MB of data with 4 KB block size on Zynq 7000 SoC.

fio experiments. In addition, we used the higher-performing Zynq UltraScale+ MPSoC to which we ported FastPath_MP, showcasing its architectural portability.

In all experiments, we compare three systems summarized in Table 3: (1) the baseline system, which uses the traditional Linux OS software stack for accessing NVMe SSDs; (2) the original FastPath implementation; and (3) variants of the proposed FastPath_MP_p implementations using different numbers of parallel FastPath units in the FPGA. For example, FastPath_MP_1 uses one FastPath module while FastPath_MP_4 uses four. Please note that the FastPath_MP_1 variant is identical in functionality with the original FastPath implementation with the only difference being that it uses the FastPath_MP controller. We also provide comparative evaluations of these two systems to quantify the overhead of the FastPath_MP controller.

4.2 Results

4.2.1 FIO Results. We evaluate *fio* in terms of performance, power, and area analysis in both types of I/O read/write operations: *sequential* and *random*.

Performance Analysis. For our performance analysis, we use the most common I/O performance metrics found in the literature [30]: latency, bandwidth, and I/O operations per second (IOPS).

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. Thus, the *overall latency* is the summation of *submission* and *completion* latencies (slat and clat, respectively), excluding the time spent for generating I/O requests in the user space to achieve more precise comparisons. Figure 9 shows the average overall block latency for 1 MB of data per configuration. We present the latency for this data size, as this is the maximum capacity of the FPGA request FIFO. FastPath_MP supports the execution of larger data sizes, but it may exhibit higher latency due to the stalls in the FIFO. Both FastPath and FastPath_MP_p (1 to 4 paths) reduce the overall latency for read and write operations by up to 95% (FastPath_MP_4 for seq-read) compared with the baseline system. The difference between FastPath and FastPath_MP_1 is negligible, showing that the modifications applied to the control module for integrating more fast paths introduce almost no overhead. Figure 9 also presents the average submission block latency on the evaluated systems, showing that both FastPath and FastPath_MP_p outperform the baseline Linux kernel submission path up to 89% and 97%, respec-

Fig. 10. Average I/O bandwidth for 1 MB of data with 4 KB block size on Zynq 7000 SoC.

tively. The reason is that both of them optimize the submission path by replacing the system calls (i.e., `submit_io`) of the baseline system with a lightweight copy of the request into the respective FPGA-based architecture. Additionally, both the FastPath and FastPath_MP_p systems reduce the completion block latency compared to the baseline Linux completion path. The completion latency improvement in FastPath is up to 84% (seq-read), while in FastPath_MP_4 it is up to 94% (seq-read). Considering that the number of I/O requests is equal for every system to ensure fairness, the difference in the completion block latency among the compared systems is attributed to the following reasons:

- The completion latency of multiple I/O requests is strongly related to the submission rate of the requests, as the completion latency is the time spent to serve all submitted I/O requests. Thus, the baseline system, which uses two 32-bit Arm cores to submit the I/O requests, is incapable of fully utilizing the processing power of the NVMe controller. However, the FastPath and FastPath_MP_p systems have significantly lower latency, as the submission of new I/O requests can take up to three FPGA cycles (three pipeline stages in the FastPath_Submit module). In particular, FastPath_MP_p allows more I/O requests to be served by the NVMe controller in the NVMe SSD, because of the increase of the parallel FPGA-based storage paths.
- FastPath and FastPath_MP_p bypass the software layers in the Linux OS, in contrast to the baseline system that needs to propagate the completion from the NVMe driver to the BIO and from BIO to the user-level application.

Bandwidth: The bandwidth is the rate that data is transferred from the storage device to memory and vice versa. Figure 10 presents the I/O bandwidth for each evaluated system on a 4 KB block size (equal to the Linux kernel's page size). As shown in Figure 10, both FastPath and FastPath_MP_p systems outperform the baseline system due to the elimination of all the expensive system calls to the kernel I/O stack and the acceleration of the creation and issue of the NVMe commands performed on the FPGA. The FastPath_MP_4 system achieves the highest throughput ranging from 426 MBps (seq-read) to 338 MBps (seq-write). Please note that according to the release specification of the particular SSD drive used in our experiments, the maximum bandwidth is 1 GBps for read operations and 800 MBps for write operations [26]. However, the reported experiments have been conducted on an x86-64 system with PCIe Gen3, and the maximum bandwidth has been achieved with 128 KB block sizes. Although FastPath_MP currently utilizes only 4 KB block sizes, we opt to extend its support for smaller and larger page sizes to meet the requirements of diverse

Fig. 11. The I/O operations per request on Zynq 7000 SoC.

Table 4. The Total On-chip Power of Three Systems (Baseline, FastPath, FastPath_MP) on Zynq 7000 SoC

System	Baseline	FastPath	FastPath_MP_1	FastPath_MP_2	FastPath_MP_4
Power (W)	3.483	3.842	3.904	3.940	4.155

The power is reported in Watt (W).

applications. Nevertheless, from our experiments, we can see that even with a slower system configuration and smaller block size, FastPath_MP_4 outperforms the baseline system by over 9× while achieving up to 50% of the maximum reported bandwidth.

I/O Operations per Second (IOPS): The IOPS metric is used to express the throughput in terms of the number of block-sized I/O requests performed within a second. IOPS are strongly connected to both the latency and the block sizes of the system, since small block sizes lead to higher number of I/O requests served by the system within a time window. Hence, the fulfilment of the same data size with larger block sizes can lead to a lower number of IOPS, since a smaller number of I/O requests is created within a second. Figure 11 presents the IOPS for all configurations with a 4 KB block size for every evaluated system. As shown in Figure 11, FastPath_MP_1, 2, 4 outperform the baseline system by up to 5×, 8×, and 12×, respectively. Please note that both metrics of IOPS and 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 *fiio* methodology of reporting performance numbers.

Scalability: The NVMe system architecture has been designed to enable scalable performance by supporting up to 64K NVMe queues with one queue dedicated per processing core. FastPath_MP has been designed to exploit the inherent scalability of NVMe by allocating multiple NVMe queues, which are controlled by the FastPath_Control_MP module. As shown in Figure 11, the throughput scales with the number of FPGA-based fast paths. All the FastPath_MP_p configurations outperform the baseline system, with the latest configuration that operates on four NVMe queues (FastPath_MP_4) achieving up to 12× and 9× higher IOPS for read and write operations, respectively.

Power Analysis. To assess the energy efficiency that FPGA processing achieves, we provide a power analysis among the evaluated systems. Table 4 presents the total on-chip power usage for every configuration on the Zynq 7000 SoC, as extracted from Xilinx Power Estimator (XPE) [38]. We calculate the energy of transferring 1 MB of data based on the reported power in Table 4

Fig. 12. The energy efficiency of FastPath and FastPath_MP (configured with 1, 2, 4 fast paths) against the baseline on Zynq 7000 SoC.

Table 5. Utilization of Logic Slices, Memory Slices, and Register Slices on Zynq 7000 SoC

System	LUTs as Logic		LUTs as Memory		CLB Registers	
	Used	Available	Used	Available	Used	Available
Baseline	20,793	171,900	887	70,400	19,995	343,800
FastPath	42,443	171,900	24,931	70,400	39,327	343,800
FastPath_MP_4	60,549	171,900	48,375	70,400	62,024	343,800

and the total execution time of every system. Figure 12 presents a normalized comparative evaluation of the energy efficiency of both FastPath and FastPath_MP against the baseline system that performs the standard Linux kernel storage path. As shown, the FastPath systems achieve over $5\times$ (seq-read) more energy efficiency than the baseline system. Additionally, both FastPath and FastPath_MP_1 consume similar power (Table 4) and perform with similar energy efficiency. This similarity indicates that the modifications introduced in the FastPath_Control_MP module are insignificant in terms of power consumption and execution time. Regarding the FastPath_MP_p systems, FastPath_MP_2 and FastPath_MP_4, achieve up to $7.3\times$ and $10\times$ higher energy efficiency, respectively. Finally, Figure 12 shows that the energy efficiency of the FastPath_MP system architecture is proportional to the number of the fast paths in the FPGA.

Area Analysis. Table 5 presents the utilization of three basic types of FPGA resources such as the Look-Up Tables (LUTs) as logic and memory and the Configurable Logic Block (CLB) registers for every evaluated system. Please note that both FastPath and FastPath_MP_p architectures were deliberately designed to grant the block memory in the FPGA to application-level functions that require FPGA acceleration. Nevertheless, since the FastPath architecture has been prototyped in Bluespec, it can be modified to convert the LUTs that are used for memory into block RAM, if required. By default, the Zynq 7000 SoC accommodates the Root Complex IP in its FPGA and occupies 12.1% of the logic resources, 1.26% of memory slices, and 5.82% of the registers. By subtracting these portions of each slice unit from the total percentage numbers reported in Table 5, the occupied resource units for FastPath_MP_4 are: 23.10% of logic slices, 67.44% of memory slices, and 12.22% of the slice registers. Comparing the original FastPath IP block [37] with FastPath_MP_4, we would naturally expect a $4\times$ increase in resource utilization. However, we observe that the increase was

Table 6. The YCSB Workloads

Workloads	A	B	C	D	F
Operations	50% read 50% update	95% read 5% update	100% read	95% read 5% insert	95% scan 5% insert

Fig. 13. Throughput of *Redis* YCSB benchmark on Baseline and FastPath_MP systems on Zynq UltraScale+ MPSoC.

only up to 2.1 $\times$. The reason behind that is that the request FIFO in the FastPath_Control_MP module has replaced the internal FIFOs in submission and completion modules of FastPath; thus, the request FIFO in the control module is shared among multiple paths.

4.2.2 Redis KVS Database Results. This paragraph presents the evaluation of FastPath_MP system in the context of the in-memory *Redis* database [3]. The *Redis* server is modified and compiled to use the FastPath library functions to write the append-only file (AOF) into the NVMe SSD. Thus, FastPath_MP is added as an option in the configuration file that is managed by the server administrator. As soon as the *Redis* server is active, it initializes connections with the client processes that require communication with the *Redis* server. Consequently, the *Redis* server starts the service of the incoming requests from the client processes. In this experiment, the YCSB benchmark is used as a client process that issues requests of four types of operations: read, insert, update, and scan, as presented in Table 6. Due to memory limitations in the Zynq 7000 SoC, we ported the FastPath_MP system to a Zynq UltraScale+ MPSoC. Hence, all experiments associated with *Redis* have been performed on the UltraScale+ MPSoC (Table 2). Finally, we compare the baseline system against FastPath_MP_4, which achieves the highest performance between all FastPath_MP_p configurations, as showcased in the previous subsection.

Performance Analysis. Figure 13 presents the throughput at which the *Redis* server executes the operations issued by the YCSB clients. Each workload presented in Figure 13 runs in isolation on the Zynq UltraScale+ MPSoC for fair comparisons. As shown in Figure 13, FastPath_MP_4 outperforms the baseline system in the majority of the workloads, including *Workload A*, *Workload B*, *Workload D*. The maximum increase in throughput is noticed in *Workload A*, and it is up to 15% higher.

Percentile Latency. The *percentile* is a statistical term used in memory and storage systems to represent the tail of the latency distribution. For example, the 90th *percentile* defines the latency threshold that separates the fastest 90% from the slowest 10% of the total samples. The range of

Fig. 14. Latency of four operations (read, insert, update, scan) of the *Redis* YCSB benchmark on the Baseline and FastPath_MP systems on Zynq UltraScale+ MPSoC. The lower, the better.

values for *percentile* is used to indicate the distribution of the I/O rate for the requests served by the memory or the disk controller. The following paragraphs present the outgrowth latency in three percentiles (90%, 99%, 99.9%) on the baseline and FastPath_MP_4 systems for four types of operations: read, insert, update, and scan, as shown in Figure 14.

Read Operations: Figure 14(a) presents the read latency of the *Redis* server when four YCSB workloads (Workloads A–D) are used as clients to perform read requests on the database. Figure 14(a) reports three snapshots of latency in the 90%, 99%, and 99.9% of the completion. As shown in Figure 14(a), there is no significant difference between the baseline and FastPath_MP_4 systems for 90th and 99th percentiles, while for 99.9th percentile FastPath_MP_4 outperforms the baseline system for most of the workloads (A, B, D). In particular, FastPath_MP_4 achieves up to 60% tail latency reduction for Workload A, which is considered a heavy workload with both read and update operations [10]. However, on Workload C FastPath_MP_4 performs up to 33% higher tail latency than the baseline system. The reason is that this workload includes only read operations and the block size used to perform the operations is important. The baseline system can perform the read operations in the granularity of 512 Bytes, while the minimum supported block size in FastPath_MP_4 is 4 KB.

Insert Operations: The insert operation is used to write data for the first time in the memory database. As shown in Figure 14(b), the 90th percentile latency shows that FastPath_MP_4 achieves up to 13% lower latency than the baseline system for both workloads, while in the 99th percentile FastPath_MP_4 shows a 12% increase on Workload D and 36% reduction on Workload E. The final percentile shows that FastPath_MP_4 outperforms the baseline systems, with latency reduction ranging from 4.5% (Workload D) up to 26% (Workload E).

Update Operations: Unlike the insert operation, which writes data for the first time in the database, the update operation is used to modify the fields of an existing entry in the database.

As shown in Figure 14(c), FastPath_MP_4 outperforms the baseline system in every percentile of both Workload A and Workload B, and the maximum latency reduction is up to 48% in Workload A.

Scan Operations: The scan operation can be considered a more complex operation than read, as it scans a specified number of entries in the database memory and returns all the data fields of a value associated with a particular *key*. As shown in Figure 14(d), the baseline and FastPath_MP_4 systems have similar latency for the scan operation, with FastPath_MP_4 reducing the latency up to 3%.

5 RELATED WORK

There is a broad range of studies that propose approaches to assist architects to deal with important decision tradeoffs [39] or novel architectural designs that mitigate the inefficiency in the state-of-the-art flash storage systems [25, 32, 36]. Broadly, these studies fall into three groups: Application/OS, In-Storage, and KVS optimizations.

5.1 Application-level & OS-level Optimizations

The first group of related work includes application-specific I/O software stacks at the user level that aim to reduce latency by adopting a lightweight stack model. Thus, many APIs have been exposed, enabling direct access to the NVMe from the user space [23, 25, 42]. Both Intel SPDK [42] and NVMeDirect [23] aim at relocating driver functionality from the OS kernel to user level, thereby bypassing completely the OS kernel. Consequently, the overall latency is improved as applications avoid the context switch and the OS software stack traversal overheads. In addition, Lee et al. [25] proposed the Application Managed Flash (AMF) architecture and Ouyang et al. [32] proposed the Software-Defined Flash (SDF) system, which both exposed flash capability at the user level. Furthermore, Seshadri et al. [36] proposed to expose the programmability of the storage device to the applications. The proposed system is named Willow and uses DRAM technology along-with FPGA-based memory controllers to emulate the NVM system. Caulfield et al. [8] proposed to bypass the BIO layer and implemented a separate driver as a way to increase performance. Lee et al. [24] presented a novel queue isolation scheme, considering the write interference and increasing the read performance in read-dominated workloads. Bjørling et al. [5] redesigned the BIO layer to enable scalability and exploit the spare hardware capabilities by implemented multi-queue support, as discussed previously in Section 2.1.2. The parallel block I/O implementation has been integrated in the Linux kernel since version 3.16.

Although previous approaches managed to bypass a significant part of the kernel's software stack, an additional user-level code is required to perform various functionalities such as command creation and scheduling. Our work, however, attempts to offload and accelerate the whole process onto the FPGA, which not only improves performance but also enables the integration of further functionality (e.g., encryption, decryption, sorting, user-defined functions) directly into the storage data path, bypassing the processors' cores completely. Schmid et al. [35] proposed MetalFS, a lightweight file system that enables near-storage processing when combining NVMe SSDs with FPGAs. Unlike MetalFS, FastPath has been designed to support raw data transfers. Therefore, a hybrid combination of both systems can be foreseen to provide support for different types of applications.

Finally, Zhang et al. [43] proposed a design that extended the mapping of the NVMe driver to the FPGA logic with an optical fiber SSD. Although, this system resulted in increased bandwidth of the data transfers to the NVMe SSD, a forced air-cooling system was required to mitigate the on-board heat increase.

5.2 In-storage Optimizations

A second group of work relates to the in-storage software system that is used by storage vendors to perform device-specific tasks, such as wear-leveling or garbage collection. Several studies have identified that *Flash File System (FFS)* makes crucial decisions that can affect both storage performance and lifetime while being unaware of the application characteristics [9, 15, 22, 32]. Thus, researchers proposed either FPGA-based [25, 32, 34] or CPU-based [14] optimizations that allow the in-storage software system to perform more efficiently.

Such an example is the AMF [25] system, which provides applications with flash functionality. The AMF architecture is prototyped on the BlueDBM system [20], which offloads in-storage flash architecture onto an FPGA and combines a network of the composed FPGA-flash nodes to aggregate multiple flash drives. Hence, the FPGA architecture can perform wear-leveling and garbage collection tasks to perform more efficiently. The BlueDBM system architecture sustains up to 2.7× higher bandwidth when using three nodes in contrast to the single node architecture.

Ouyank et al. [32] exposed the hardware SSD channels to the user level, achieving almost 300% higher I/O bandwidth than the conventional system. The proposed SDF architecture deploys FPGAs to accommodate the lightweight memory controller, which does not perform common flash operations, such as *Wear-Leveling* and *Garbage Collection*. However an erasure command is provided to applications that can erase blocks in the flash memory. Ruan et al. [34] proposed INSIDER, a system that utilizes the FPGA technology within the storage drive to enhance the capacity of the storage controller. The proposed system exploits the FPGA spatial parallelism and allocates different slots for different applications. Unlike INSIDER, we present an architecture that utilizes the available FPGA parallelism by increasing the number of storage paths on the FPGA.

Gu et al. [14] leveraged the cores of the processor in the storage device and proposed a system named Biscuit, which installs a runtime system, as software running on the SSD Controller. Biscuit was capable of processing data on the controller, without requiring the costly transfers to the processors of the servers, achieving up to 15× average speedup against conventional host-SSD systems.

The main difference of our work with this group of related work is that we propose a system architecture that is orthogonal to the in-storage architecture, as our work does not interfere with the internal path in the storage block device. However, it could be extended to accelerate both the OS-level and the in-storage system software.

5.3 KVS Optimizations on Memory/Storage Systems

The third group of related work includes References [6, 12, 27, 31, 40] that propose novelties on KVS in-memory database configurations (e.g., Redis, memcached) with the aim of scaling up the performance in distributed systems. The difference with our work is that we propose a general-purpose FPGA-based storage multi-path system instead of an application-specific hardware design. However, our system can be extended to explore the KVS case studies over distributed nodes.

6 CONCLUSION

In recent years, storage research has pushed effort towards new technologies (e.g., NVMe SSDs) to tackle the inefficiency of conventional hard disks. Although the adoption of the PCIe interconnect enabled contemporary disks to deliver higher performance, there are still performance bottlenecks to be tackled due to the inefficient legacy interfaces. This article proposes FastPath_MP, a scalable and portable FPGA-based storage architecture that operates with no dependency on the main processor of the system. The experimental analysis on the Xilinx Zynq-7000 SoC showed that FastPath_MP outperforms the baseline system with up to 82% lower latency, up to 12× higher

throughput, and up to 10× more energy efficiency. Further experiments on Xilinx Zynq UltraScale+ MPSoC using the Redis in-memory database showed that FastPath_MP achieves up to 60% lower tail latency and 15% higher throughput than the baseline Linux storage path.

In the future work, we plan to explore the FastPath_MP in multiple directions, such as *near-data processing* and facilitating the remote access to the storage device via FastPath_MP.

ACKNOWLEDGMENTS

We would like to thank Andrew Attwood and Joshua Lant for initial discussions.

REFERENCES

- [1] 2012. NVMe Express Explained. http://nvmexpress.org/wp-content/uploads/2013/04/NVMe_whitepaper.pdf.
- [2] 2014. NVMe Express Specification. <http://nvmexpress.org/resources/specifications/>.
- [3] 2019. Redis. <http://redis.io/>.
- [4] J. Bhimani, J. Yang, Z. Yang, N. Mi, Q. Xu, M. Awasthi, R. Pandurangan, and V. Balakrishnan. 2016. Understanding performance of I/O intensive containerized applications for NVMe SSDs. In *Proceedings of the IEEE 35th International Performance Computing and Communications Conference (IPCCC'16)*.
- [5] Matias Björling, Jens Axboe, David Nellans, and Philippe Bonnet. 2013. Linux block IO: Introducing multi-queue SSD access on multi-core systems. In *Proceedings of the 6th International Systems and Storage Conference (SYSTOR'13)*.
- [6] Michaela Blott, Ling Liu, Kimon Karras, and Kees Visser. 2015. Scaling out to a single-node 80Gbps memcached server with 40Terabytes of memory. In *Proceedings of the 7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'15)*.
- [7] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. 2016. A cloud-scale acceleration architecture. In *Proceedings of the 49th IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*.
- [8] Adrian M. Caulfield, Arup De, Joel Coburn, Todor I. Mollow, Rajesh K. Gupta, and Steven Swanson. 2010. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proceedings of the 43rd IEEE/ACM International Symposium on Microarchitecture (MICRO'10)*.
- [9] Hyun Jin Choi, Seung-Ho Lim, and Kyu Ho Park. 2009. JFTL: A flash translation layer based on a journal remapping for flash memory. *Trans. Stor.* 4, 4 (2009).
- [10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC'10)*. ACM, New York, NY, 143–154. DOI: <https://doi.org/10.1145/1807128.1807152>
- [11] K. Eshghi and R. Micheloni. 2013. *SSD Architecture and PCI Express Interface*. Springer, Dordrecht.
- [12] E. S. Fukuda, H. Inoue, T. Takenaka, Dahoo Kim, T. Sadahisa, T. Asai, and M. Motomura. 2014. Caching memcached at reconfigurable network interface. In *Proceedings of the 24th International Conference on Field Programmable Logic and Applications (FPL'14)*.
- [13] Rino Micheloni, Giovanni Campardo, and David Novosel. 2005. *VLSI-Design of Non-volatile Memories*. Springer-Verlag Berlin Heidelberg.
- [14] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanhoo Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. 2016. Biscuit: A framework for near-data processing of big data workloads. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA'16)*.
- [15] Keonsoo Ha. 2011. A program context-aware data separation technique for reducing garbage collection overhead in NAND flash memory. In *Proceedings of the 27th IEEE Symposium on Massive Storage Systems and Technologies*.
- [16] M. C. Herbordt, T. VanCourt, Y. Gu, B. Sukhwani, A. Conti, J. Model, and D. DiSabello. 2007. Achieving high performance with FPGA-Based computing. *Computer* 40, 3 (2007), 50–57.
- [17] J. Axboe. 2004. Linux Block IO-present and Future. Retrieved from <http://www.landley.net/kdocs/ols/2004/ols2004v1-pages-51-62.pdf>.
- [18] Jens Axboe. [n.d.]. Flexible I/O. Retrieved from <https://github.com/axboe/fio>.
- [19] Jens Axboe. 2004. Linux Block IO Present and Future.
- [20] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, and Arvind. 2015. BlueDBM: An appliance for big data analytics. In *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA'15)*.
- [21] K. Marks. 2013. An NVMe Express Tutorial. Retrieved from https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2013/20130812_PreConfD_Marks.pdf.

- [22] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. 2014. The multi-streamed solid-state drive. In *Proceedings of the 6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'14)*.
- [23] Hyeon-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. 2016. NVMeDirect: A user-space I/O framework for application-specific optimization on NVMe SSDs. In *Proceedings of the 8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'16)*.
- [24] Minkyong Lee, Dong Hyun Kang, Minho Lee, and Young Ik Eom. 2017. Improving read performance by isolating multiple queues in NVMe SSDs. In *Proceedings of the 11th International Conference on Ubiquitous Information Management and Communication (IMCOM'17)*.
- [25] Sungjin Lee, Ming Liu, Sangwoo Jun, Shuotao Xu, Jihong Kim, and Arvind. 2016. Application-managed flash. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*. USENIX Association.
- [26] Lee Won-ju, Chang Jong-baek, Song Sang-hoon, Kim Jaen-eun, Lee Sang-geol, Koh Seung-wan, Park Hae-sung, Kim Sung-wook, Jang You-jin, Na You-jung, Choi Young-gil. 2014. *The Ultimate NVMe SSD for Data Center*. SM953 White Paper. Retrieved from https://s3.ap-northeast-2.amazonaws.com/global.semi.static/SM953_Whitepaper-0.pdf.
- [27] Kevin Lim, David Meisner, Ali G. Saidi, Parthasarathy Ranganathan, and Thomas F. Wenisch. 2013. Thin servers with smart pipes: Designing SoC accelerators for memcached. In *Proceedings of the 40th International Symposium on Computer Architecture (ISCA'13)*.
- [28] R. Mahajan, R. Sankman, N. Patel, D. Kim, K. Aygun, Z. Qian, Y. Mekonnen, I. Salama, S. Sharan, D. Iyengar, and D. Malik. 2016. Embedded multi-die interconnect bridge (EMIB)—A high density, high bandwidth packaging interconnect. In *Proceedings of the IEEE 66th Electronic Components and Technology Conference (ECTC'16)*.
- [29] G. E. Moore. 1998. Cramming more components onto integrated circuits. *Proc. IEEE* 86, 1 (1998), 82–85.
- [30] Mark Moshayedi and Patrick Wilkison. 2008. Enterprise SSDs. *Queue* 6, 4 (July 2008).
- [31] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling memcache at Facebook. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation*.
- [32] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. 2014. SDF: Software-defined flash for web-scale internet storage systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'14)*.
- [33] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. 2009. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA'09)*.
- [34] Zhenyuan Ruan, Tong He, and Jason Cong. 2019. INSIDER: Designing in-storage computing system for emerging high-performance drive. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'19)*.
- [35] Robert Schmid, Max Plauth, Lukas Wenzel, Felix Eberhardt, and Andreas Polze. 2020. Accessible near-storage computing with FPGAs. In *Proceedings of the 15th European Conference on Computer Systems (EuroSys'20)*.
- [36] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. 2014. Willow: A user-programmable SSD. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*.
- [37] A. Stratikopoulos, C. Kotselidis, J. Goodacre, and M. Luján. 2018. FastPath: Towards wire-speed NVMe SSDs. In *Proceedings of the 28th International Conference on Field Programmable Logic and Applications (FPL'18)*.
- [38] Xilinx Inc. 2012. Xilinx Power Estimator (XPE). Retrieved from <http://www.xilinx.com/products/technology/power/xpe.html>.
- [39] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. 2015. Performance analysis of NVMe SSDs and their implication on real world databases. In *Proceedings of the 8th ACM International Systems and Storage Conference (SYSTOR'15)*.
- [40] Shuotao Xu, Sungjin Lee, Sang-Woo Jun, Ming Liu, Jamey Hicks, and Arvind. 2016. BlueCache: A scalable distributed flash-based key-value store. *Proc. VLDB Endow.* 10, 4 (2016), 301–312.
- [41] Jisoo Yang, Dave B. Minton, and Frank Hady. 2012. When poll is better than interrupt. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*.
- [42] Z. Yang, J. R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, V. Verma, and L. E. Paul. 2017. SPDK: A development kit to build high performance storage applications. In *Proceedings of the IEEE International Conference on Cloud Computing Technology and Science (CloudCom'17)*.
- [43] J. Zhang, F. Meng, L. Qiao, and K. Zhu. 2019. Design and implementation of optical fiber SSD exploiting FPGA accelerated NVMe. *IEEE Access* (2019), 152944–152952.

Received March 2020; revised July 2020; accepted September 2020