Position: Designing a Storage Software Stack for Accelerators

Shinichi Awamoto (IIJ), Erich Focht (NEC Deutschland) and Michio Honda (University of Edinburgh)

Abstract

Although modern accelerator devices, such as vector engines and SmartNICs, are equipped with general purpose CPUs, access to the storage needs the mediation of the host kernel and CPUs, resulting in latency and throughput penalties. In this paper, we explore the case for direct storage access inside the accelerator applications, and discuss the problem, design options and benefits of this architecture. We demonstrate our architecture can improve 12–89% in microbenchmarks or realistic YCSB workloads in LevelDB, and reduce the execution time by 33–46 % in a bioinformatics application in comparison to the baseline where the host system mediates the storage accesses.

1 Introduction

While CPUs have increasingly integrated more specialized units and instructions, economics and scaling demand suggest that more work and even market be pushed to the peripheral devices. The resulting dispersion promotes the relevance of accelerator devices equipped with system-on-chip (SoC) as a co-primary compute resource.

Leading commodities of SoC-based accelerators are SmartNICs and vector processors. Since these devices, unlike GPUs, implement general purpose processor cores in addition to specialized compute units, tiered memory and I/O ports, they can locally execute the vast majority of application logics. The resulting generality largely expands application domains; recent work demonstrates that distributed applications, consensus algorithms, data analytics and storage applications appreciate SoC-based accelerator devices, improving throughput, latency and energy efficiency [7, 8].

The current SoC-accelerator applications are limited to stateless processing. Access to stateful data resident in persistent storage is mediated by the host OS that runs on CPUs and manages storage medium. Since storage devices have been improved in both throughput and latency, this device-CPU-device communication overhead is high enough to impact on the response time (Section 2). A recent approach in 2019, iPipe [7], *masks* this overhead by reconciling other applications, but this approach imposes redesign on the applications, nor does it rectify latency and throughput of individual applications constrained merely by the said overhead. We therefore must provide a streamlined, yet simple abstraction of persistent storage to accelerator applications.

In this paper, we explore the case and need for direct storage access in the SoC-based accelerator devices in designing Hayagui, our storage software stack. Although Hayagui is implemented in a vector engine accelerator device, its architecture could apply to other classes of SoC devices, such as SmartNICs, because they exhibit similar characteristics in performance and architecture (Section 2.1). Hayagui ports and augments a user-space NVMe driver to allow the accelerator applications to access the storage medium without the host CPU or OS being involved. On top of it, Hayagui implements its own file system and ports LevelDB to provide useful interfaces to the applications.

We show Hayagui can accelerate a bioinformatics application that reads genome sequence data generated by a DNA sequencer device and matches the data against reference data, for example, to find mutation. This application utilizes the vector engine accelerator that we use throughout this paper to take advantage of its massive parallelism, but suffers from slow host storage access.

This paper discusses following research questions:

1. What is storage performance characteristics of the accelerator?
2. What the storage stack design for the accelerator would look like?
3. What application could benefit from such a storage stack?
4. What are challenges to design accelerator storage stacks?

2 Motivation

Storage performance matters for SoC-based accelerator applications, because they must read data from the storage medium before performing efficient data parallel jobs, or write back compute results that will be used by the same or other applications. This also means that the higher the compute capacity is, the more the storage I/O would dominate the end-to-end execution time.

2.1 System Model

We assume SoC-based accelerator devices, attached to a PCIe bus of a commodity system, and accompanied by general purpose cores that are able to execute the entire application code. These code are compiled by either open or proprietary compiler, and may include device specific code to take the advantages of specialized compute units on the accelerator device. The accelerator cores entirely operate in a user space context; system calls will be redirected to the host kernel that runs on the host CPUs and centralises the system resources, such as storage and networks.

Figure 1 illustrates this system model. The accelerator OS (AOS), which proxies system calls issued by accelerator applications, would reside in the host kernel, but we believe many accelerators vendors, as with that of our accelerator,

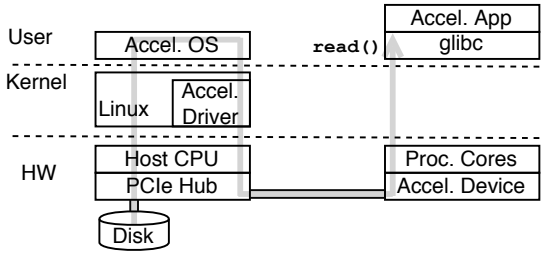

Figure 1: **Accelerator System Model.** Although applications can entirely execute in the accelerator devices, system calls are handled by the Accelerator OS (AOS).

adopt the user space AOS model because of simplicity and integrity with other building blocks, such as initialization of program execution.

Our accelerator, NEC SX-Aurora TSUBASA [13], follows this system model. It is equipped with eight pairs of a general purpose and vector processor core, 16 MB of last-level cache, and 1.2 TB/s of the total memory bandwidth. Komatsu et al. [5] reports the parallelism and memory bandwidth advantages of this accelerator device in large-scale simulation. Both LLVM-based open-source compiler [12], which we use throughout this paper, and closed-source one [13] are available to compile the accelerator applications written in regular C/C++. The vendor also provides `glibc` library optimized to take the advantages of vector processing units. AOS maintains user processes that proxy system calls issued by their corresponding accelerator applications, using the accelerator driver that delivers interrupts and performs DMA to and from the accelerator device.

2.2 Accelerator Data Management Problem

The process of executing system call is clearly inefficient. Every call must travel the PCIe bus at least twice, and crosses the kernel boundary twice, one from the kernel to the AOS and the other from the AOS to the kernel. Moreover, timely execution of the system call requires that host-side CPUs be dedicated to AOS processes. To confirm this, we measure system call latency inside the accelerator using a tight loop of cheap `ioctl`s. We observe 42748 ns per call, whereas the same call takes 105 ns in the host, confirming 420× higher system call costs, from which small data transfers could suffer[1].

Moreover, data must be transferred multiple times. In Figure 1, the light gray arrow indicates the data movement of a `read()` system call. The host kernel reads data from the disk, then moves them to the user space where the AOS runs. The AOS then moves data to the kernel again so that the accelerator driver can transfer the data to the accelerator device via DMA.

Figure 2 plots throughput of read, write and sync operations that run synchronously in a single thread. As a highlight from the left figure, to read 1 MB of data from the kernel buffer

[1]All the experiments in this section use the setup detailed in Section 5.

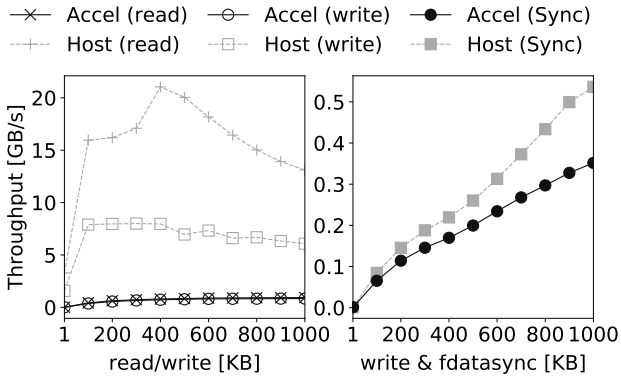

Figure 2: **I/O Throughput in Accelerator and Host.** In the accelerator, `read`, `write` and `write+fdatasync` are 92–99%, 86–99% and 21–37% slower, respectively.

Type	Bacteria	Killfish	Mouse
Reference sequence	3.2 MB	997 MB	2.7 GB
Target sequence	2.1 GB	7.0 GB	15.0 GB

Table 1: **Size of DNA Sequence Data.** Target sequences read by the DNA sequencer are much larger than reference ones, because they contain extra data such as quality scores.

cache using a single `read()` system call, we observe that it takes 1087 µs ($1000000/(0.9197 \times 1000)$) in the accelerator, while it takes only 77 µs in the host, demonstrating the data copy and transfer costs. These costs are nontrivial even in the presence of access to the storage medium. As in Figure 2 right, in writing 1 MB of data to the NVMe SSD using a `write()` followed by `fdatasync()`, it takes 2937 µs in the accelerator, which takes 1847 µs in the host. Since the throughput in the accelerator is constrained by both storage medium and data copies, the difference from the host case becomes larger as the I/O size increases.

2.3 Case Study: Genome Sequence Matching

One of the attractive use of the SoC-based accelerator device includes bioinformatics applications that process data generated by a DNA sequencer apparatus and stored in the disk. One essential operation is genome sequence matching. This task uses two DNA sequences: a *reference* sequence that describes particular species and a *target* sequence that is going to be analysed, for example, to find mutations, based on the reference sequence.

The genome data are represented as text whose size varies depending on species. As shown in Table 1, the reference sequence of bacteria, killfish and mouse is 3.2 MB, 997 MB and 2.7 GB in size, respectively; the target sequence of those species is 2.1 GB, 7.0 GB and 15.0 GB in size, respectively.

Operation	Bacteria	Killfish	Time [s] Mouse
Load reference sequence	0.032	0.010	2.282
Index reference sequence	0.634	0.259	33.859
Save reference index	0.093	0.101	2.019
Load reference index	0.028	0.101	1.463
Load target sequence	19.997	53.943	113.768
Matching	1.500	1.354	3.326
Total	22.284	55.768	156.717

Table 2: **End-to-End Genome Sequence Matching Time.** Since the accelerator can process data efficiently by exploiting its vector engines, the application spends most of the time to load target data to be analysed. See Table 1 for the size of data.

The application running in the accelerator first reads the reference sequence. Since this process also locates delimiters to speed up the later indexing process, it reads data in a batch (e.g., 64 KB). The application then *indexes* them, so that the matching process can access arbitrary regions (base pairs) of the sequence at a constant time. The resulting index is filed in the disk, and read again before the target sequence is analysed. The application then loads the target sequence in the same way with the reference sequence (i.e., with locating delimiters), and matches the data against the indexed reference.

Table 2 shows the breakdown of the end-to-end execution time. Since the application actively takes the advantages of vector processing engine for both locating delimiters and indexing sequences, it spends 72–96% of time to read the target sequence data. The reference sequence read time of killfish is shorter than bacteria. This is because the larger number of delimiters are found in the bacteria data. We conclude that the storage I/O inefficiencies significantly impact on the end-to-end latency and throughput of accelerator applications.

3 Design Space

Since redirecting storage I/O requests and data to the host comes at a significant cost and thus impairs accelerator applications, we need to design a streamlined I/O stack between the application and the storage medium.

Recall that the entire application code can be executed using the general purpose core of the SoC-based accelerator, and their system calls are redirected to the host that manages system resources, including I/O peripheral devices. Thus, it would be an option to run an OS kernel inside the accelerator, as with Multikernel [3] and Popcorn Linux [2]. Unfortunately, since many of SoC-based accelerators, including our one, have no kernel context, we reject this design option.

Another option is to run a library operating system, which often ports a full-fledged OS kernel like Linux in the user-space, alongside the accelerator applications. However, this approach would result in poor performance due to the overheads of the OS kernel code, including the system call, file system and block layer, which are generic and large. These overheads could be particularly higher in the form of library operating system, because many kernel features, such as interrupts, kernel threads and synchronization primitives, must be emulated. For example, Linux kernel library, although it is extremely useful for functional testing or debugging purpose, reports very low performance [1]. However, we might revisit this option to take the advantages of the rich feature set, such as file system implementations, and tackle the inefficiencies specific to the accelerator environment.

The other option is buffer cache sharing between the host kernel file system and the accelerator device, as with GPUfs [15]. Although this approach allows the name space to be shared between the host and accelerator applications, they require data to be loaded from the storage medium to the host-side memory. Given that SoC-based accelerators, unlike GPUs, have general purpose cores and memory, and can execute the entire application code, we examine radical approaches that the accelerator device directly accesses the storage medium.

4 Hayagui Architecture

To understand the fundamental performance benefit of the direct access architecture, we begin with minimalistic software components, primarily focusing on key value store and genome analysis applications. This approach enables *bottom-up* design, taking into account microarchitectural weakness of the general purpose cores of the accelerator, and opportunities of exploiting the specialized engines. Therefore, our approach could be useful when designing an I/O software stack for other SoC-based accelerators.

Figure 3 illustrates the architecture of Hayagui. The main components are Direct I/O Engine that enables direct access to the storage medium, Accelerator File System that manages on-disk data layout and serves buffer caches, and LevelDB that provides simple key-value store interfaces. The applications that use LevelDB as their storage backend do not need to be modified to use Hayagui. In the rest of this section, we briefly describe these components.

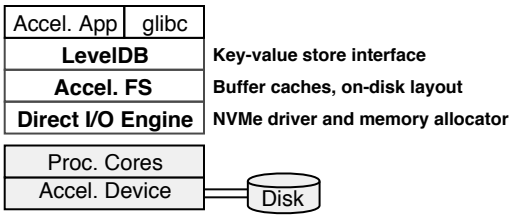

Figure 3: **Hayagui Architecture.** It runs the whole stack from device driver to user interface within the user-space context of accelerator device.

Direct I/O Engine. Hayagui resorts to the user space device driver, because the kernel context is unavailable in the

accelerator, and this approach would allow us to implement minimalistic higher level functionality. HAYAGUI extends UNVMe [11]; Intel SPDK [4], another user-space storage device driver for high performance, was also an option, but we ruled it out because of its large software base that could be, at least partially, incompatible with performance characteristics of the accelerator devices. We thus set off designing our I/O engine by carefully extending the smaller base system.

We rely on `uio` framework to grant PCIe register access to the user-space context of the accelerator. The direct I/O engine accesses these registers using PCIe APIs provided by the accelerator vendor. Although our current design initializes the device inside the accelerator, we may revise this process to do so at the host side so that system administrators can enforce some policy, such as access control, to the accelerator applications. We also implement a buddy-system based DMA buffer allocator used by UNVMe, because the native allocator, `vfio` framework, is unavailable in the accelerator.

Accelerator File System. UNVMe provides low-level block I/O commands on DMA-capable memory claimed by our memory allocator. I/O is asynchronous, and its completions are detected by the notification API similar to POSIX `poll`. Running applications directly on top of UNVMe interfaces is inconvenient, because it does not name or organize data, nor does manage read or write buffer caches.

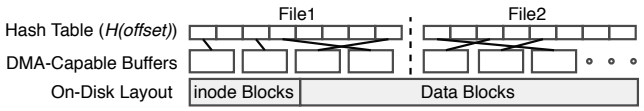

Figure 4: **Accelerator File System.**

Our accelerator file system (AccelFS) resembles a conventional `ext2` file system in on-disk layout, as inode blocks directly or indirectly point to data blocks. Figure 4 depicts our on-disk layout and buffer cache management. The DMA-capable buffers, backed by the accelerator DRAM and mapped into the host address space, are tracked against dirty state. We currently use 8 KB for block size, and allocate at most 16 cache blocks to each file. Since dynamic memory allocation and linked-list manipulation appear expensive in the accelerator core, we employ a per-file hash table to look up a buffer cache for a given file offset. We will optimize this buffer cache management to take the advantages of vector processing engine in the future.

Key Value Store Interface. Since many applications today use key-value stores as their storage backend, we port popular LevelDB on top of our accelerator file system. LevelDB is a write-optimized, embeddable persistent key-value store based on the Log Structured Merge (LSM) tree. It supports `set` and `get` queries, and range queries. LevelDB organizes data into multiple *levels* in which each grows larger than the previous level. Each level has one or more *sstables*, in which each contains keys and values sorted for efficient search. When the highest level becomes full, the *compaction* process reorganizes

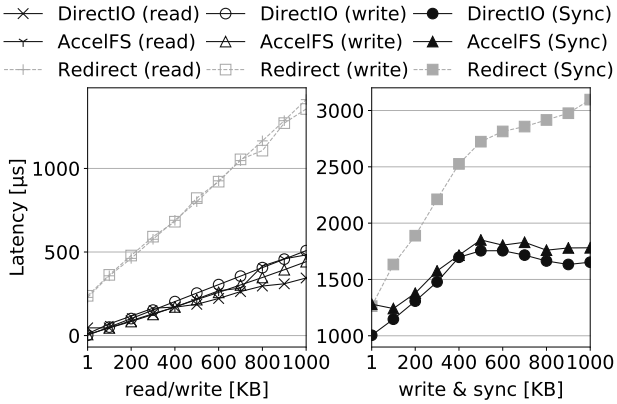

Figure 5: **I/O Latency of Direct I/O Engine and AccelFS (Lower is better).** Direct I/O Engine and AccelFS achieve lower latencies than system call redirects (Direct I/O: 74–87% in read, 69–85% in write and 21–47% in sync. AccelFS: 64–95% in read, 67–97% in write and 0–42% in sync).

the sstables into the next level by creating new ones. New data are stored in *memtable* backed by DRAM and write-ahead log backed by persistent storage.

Since LevelDB generates a number of files, including operation log, WAL and sstables, we cannot run LevelDB directly on our Direct I/O Engine. Our AccelFS nicely supports these files, persisting and caching data as LevelDB wishes.

LevelDB has modular backend architecture. The *memory* backend is the fastest but data are not persisted. The *POSIX* backend manipulates data using POSIX system calls. We implement a new backend that uses AccelFS.

5 Experimental Results

This section presents how our architecture or its components improve storage performance in a series of experiments. Our prototype is publicly available at `https://anonymizedurl.com`.

Hardware, OS and Compiler. We use a single rack-mount server equipped with a 12-core Intel Xeon Gold 6126 processor clocked at 2.60 GHz and 96 GB of RAM. The server runs with CentOS 7.5 Linux kernel 3.10.0, and disables hyperthreading and all the CPU sleep states. It installs a single NEC SX-Aurora TSUBASA accelerator device and Samsung EVO 970 NVMe SSD device. All the accelerator code is compiled by LLVM 1.4.0 for NEC SX-Aurora VE, an open-source compiler implemented as LLVM backend [9].

How does the direct I/O architecture improve storage I/O performance? Figure 5 plots latency of read, write and sync operation measured by a single threaded tight loop; next request starts after the completion of the previous one. Reads always read new data from the storage medium, although it looks up the buffer caches (and fails) in the AccelFS cases.

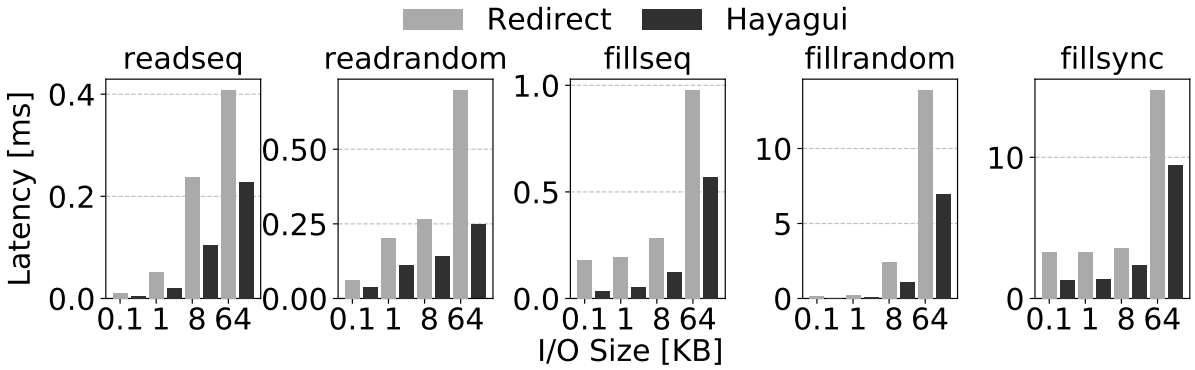

Figure 6: **db_bench Results (Lower is better).** Workloads represent sequential/random reads/writes, and sync to the storage medium. HAYAGUI improves the baselines by 33–81%. In 0.1 KB requests, readseq and fillrandom are invisible but improved by 49 and 79%, respectively.

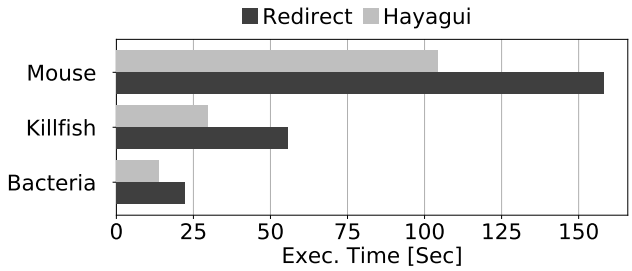

Figure 7: **Genome Sequence Matching Performance.** HAYAGUI reduces the end-to-end execution time by 33–46% or by 8–53 seconds%. Data sets are are 2.1–15.0 GB in size (see Table 1 in Section 2.3).

The key take away is that direct I/O significantly improves performance in all the operations, and AccelFS incurs little overheads.

How does HAYAGUI bring the benefits into a realistic storage system? We measure performance of LevelDB that runs in the accelerator with or without HAYAGUI, using db_bench to generate sequential and random access workloads. Figure 6 plots per-request latencies, and shows HAYAGUI exhibits 33–81% of improvements. The margins largely differ depending on both request sizes and workloads. This is because both system call costs, which are 42 µs per call (Section 2.2), and data copy costs (see the same section) are relevant. In baselines, reads are often faster than the system call costs i.e., < 42 µs, because of memtable or cache of LevelDB itself. We also ran YCSB workloads (update-heavy, read-mostly, read-only, read-latest), and observed 12–89 % of improvements (not in the graphs). Therefore, we conclude that, despite the additional software overheads, HAYAGUI significantly improves storage system performance.

How does HAYAGUI improve a realistic accelerator application? In Section 2.3, we observed that a bioinformatics application that runs in the vector engine accelerator spends the vast majority of time in reading data from the disk (Table 2). We therefore ported the same application to use HAYAGUI. Since this application reads only a few files, we run this application directly on top of our Direct I/O Engine. As described in Section 2.3, this application reads data, which is 2.1–15.0 GB depending on the species (see Table 1), by a 64 KB of chunk, while parsing it to find delimiter characters using the vector engine.

Figure 7 plots the end-to-end execution times of the application against the three species used in Section 2.3 with or without HAYAGUI. HAYAGUI reduces the execution time by 33–46% or by 8–53 seconds. These results demonstrate that HAYAGUI accelerates a realistic application and it serves large data efficiently, minimizing data transfers between the application and storage medium.

6 Related Work and Conclusion

To the best of our knowledge, HAYAGUI is the first system that runs the whole storage stack on the general-purpose cores of SoC-based accelerator. In-storage computing based on FPGAs, such as INSIDER [14] and CognitiveSSD [6], are complementary to HAYAGUI, because their host-side API and library could run in the SoC accelerator. Accelerator management systems, such as iPipe [7] and DlibOS [10], do not support direct device access.

In this paper, we showed system call and data copy problems in accelerator applications. We then explored the direct storage I/O architecture in designing HAYAGUI, and empirically demonstrated its efficiency using microbenchmarks and bioinformatics application. The next section discusses further research questions.

7 Discussion Topic

We believe SoC-based accelerators introduce a number of problems and opportunities to build or use storage systems. Based on the results presented in this paper, we plan to discuss following topics during HotStorage 2020:

- **Feasibility of direct storage access architecture.**
  In this paper, we used NEC SX-Aurora as the SoC-based accelerator device equipped with general purpose cores, and enabled the direct storage I/O using vendor-specific PCie APIs. Some other accelerators such as SmartNICs seem to allow for similar organization, but what about future SoC-based accelerator devices? Will they accept, ease or disable the direct storage access?

- **General purpose core considerations.**
  Our accelerators have relatively low-performance general-purpose cores, and we already observed notable limitations, such as slow `malloc` and atomic operations. How could we overcome such weakness that would be prevalent in accelerator devices? Further, some accelerators, such as high-end SmartNICs, have rather beefy CPU cores, but can we assume the same performance characteristics with the host CPUs?

- **Accelerator engine considerations.**
  Accelerator devices primarily focus on their specialized compute engines, such as vector processors and encryption engines, typically used by specific syntax or APIs. Given that their general-purpose cores can be weak, is it possible to benefit from these specialized engines to design storage stacks?

- **Storage Resource Sharing.**
  Our current design dedicates an entire storage device to the accelerator device. Do we need a shared name space across all the (potentially heterogeneous) accelerator devices and host CPUs, and if so, how could this be achieved? We would design a centralized architecture managed by one of the host or accelerators, or a distributed architecture between the host and accelerators, which would pose challenges in consistency and performance locality.

- **Generic libraries and frameworks for accelerator storage stack.**
  It would be painful if we need to build an efficient storage stack for every accelerator from scratch. Ideally, we wish to have performance-critical, accelerator-dependent libraries that can be utilized by a common framework. Is it possible to identify features that should be implemented in such libraries, and define common interfaces?

References

[1] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, et al. "{SCONE}: Secure Linux Containers with Intel {SGX}". In: *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 2016, pp. 689–703.

[2] A. Barbalace, B. Ravindran, and D. Katz. "Popcorn: a replicated-kernel OS based on Linux". In: *Proceedings of the Linux Symposium, Ottawa, Canada*. 2014.

[3] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. "The multikernel: a new OS architecture for scalable multicore systems". In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM. 2009, pp. 29–44.

[4] Intel. *Storage Performance Development Kit*. https://spdk.io.

[5] K. Komatsu, S. Momose, Y. Isobe, O. Watanabe, A. Musa, M. Yokokawa, T. Aoyama, M. Sato, and H. Kobayashi. "Performance evaluation of a vector supercomputer SX-aurora TSUBASA". In: *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM/IEEE. 2018, pp. 685–696.

[6] S. Liang, Y. Wang, Y. Lu, Z. Yang, H. Li, and X. Li. "Cognitive SSD: A Deep Learning Engine for In-Storage Data Retrieval". In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, July 2019, pp. 395–410. ISBN: 978-1-939133-03-8. URL: https://www.usenix.org/conference/atc19/presentation/liang.

[7] M. Liu, T. Cui, H. Schuh, A. Krishnamurthy, S. Peter, and K. Gupta. "Offloading Distributed Applications Onto smartNICs Using iPipe". In: *Proceedings of the ACM Special Interest Group on Data Communication*. SIGCOMM '19. Beijing, China: ACM, 2019, pp. 318–333. ISBN: 978-1-4503-5956-6. DOI: 10.1145/3341302.3342079. URL: http://doi.acm.org/10.1145/3341302.3342079.

[8] M. Liu, S. Peter, A. Krishnamurthy, and P. M. Phothilimthana. "E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers". In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, July 2019, pp. 363–378. ISBN: 978-1-939133-03-8. URL: https://www.usenix.org/conference/atc19/presentation/liu-ming.

[9] *LLVM for NEC SX-Aurora VE*. https://github.com/sx-aurora-dev/llvm.

[10] S. Mallon, V. Gramoli, and G. Jourjon. “DLibOS: Performance and protection with a network-on-chip”. In: *ACM SIGPLAN Notices* 53.2 (2018), pp. 737–750.

[11] MicronSSD. *UNVMe - A User Space NVMe Driver*. https://github.com/MicronSSD/unvme.

[12] S. Moll, M. Kurtenacker, E. Focht, and S. Hack. *NEC SX-Aurora TSUBASA and the LLVM compiler infrastructure*. https://fs.hlrs.de/projects/teraflop/28thWorkshop_talks/WSSP28_SMoll_llvm-sve.pdf.

[13] NEC. *NEC SX-Aurora TSUBASA - Vector Engine*. https://www.nec.com/en/global/solutions/hpc/sx/vector_engine.html.

[14] Z. Ruan, T. He, and J. Cong. “INSIDER: Designing In-Storage Computing System for Emerging High-Performance Drive”. In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, July 2019, pp. 379–394. ISBN: 978-1-939133-03-8. URL: https://www.usenix.org/conference/atc19/presentation/ruan.

[15] M. Silberstein, B. Ford, I. Keidar, and E. Witchel. “GPUfs: integrating a file system with GPUs”. In: *ACM SIGPLAN Notices*. Vol. 48. 4. ACM. 2013, pp. 485–498.