Designing a Storage Software Stack for Accelerators

Shinichi Awamoto*†, Erich Focht‡ and Michio Honda
†*NEC Labs Europe,* ‡*NEC Deutschland, 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 that our architecture can improve throughputs of LevelDB by 12–89%, 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 share is being pushed towards the peripheral devices. The resulting dispersion increases 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 logic. 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 [11, 12].

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 media. Since storage devices have been improved in both throughput and latency, this device-CPU-device communication overhead is high enough to impact response time (Section 2). A recent approach in 2019, iPipe [11], *masks* this overhead by reconciling other applications, but this approach imposes redesigning the applications, and does not improve actual latency and throughput of individual applications constrained merely by the said overhead. We want to tackle these issues and therefore aim at providing a streamlined, yet simple abstraction of persistent storage to accelerator applications.

*Currently at IIJ.

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 that 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 mutations. 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 are storage performance characteristics of the accelerator?
2. What would the storage stack design for the accelerator 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 entire application codes. These codes are compiled by either open or proprietary compilers, 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 centralizes the system resources, such as storage and networks.

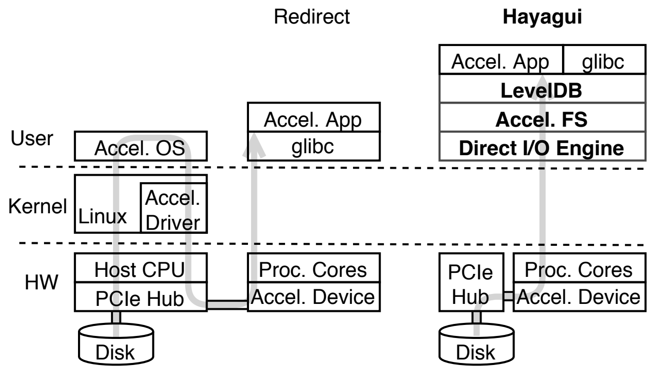

Figure 1: **System Model and Software Architecture.** In today's redirect architecture (left), I/O requests issued by accelerator applications are handled by the host CPU. HAYAGUI (right) bypasses the host CPUs by running the whole stack including the device driver within the accelerator. Gray arrows indicate data paths from the storage to the application.

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, 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 [18], follows this system model. It is equipped with eight pairs of a general purpose core and a vector processor core, 16 MB of last-level cache, 48GB of on-chip HBM2 based DRAM and 1.2 TB/s of the total memory bandwidth. This accelerator is currently used for image processing in the medical sector, risk calculation in the financial sector, TensorFlow acceleration in the logistics sector, and weather prediction in the German weather forecasting service. Komatsu et al. [8] report the parallelism and memory bandwidth advantages of this accelerator in large-scale simulations. Both, LLVM-based open-source compilers [16], which we use throughout this paper, and closed-source ones [18] are available to compile the accelerator applications written in regular C/C++. The vendor also provides a `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 calls is clearly inefficient. Every call must travel the PCIe bus at least twice between the accelerator device and the host CPU, and cross the user-kernel boundary twice between the kernel and the AOS. Moreover, timely execution of the system call requires that host-side CPUs be dedicated to AOS processes. To confirm this, we

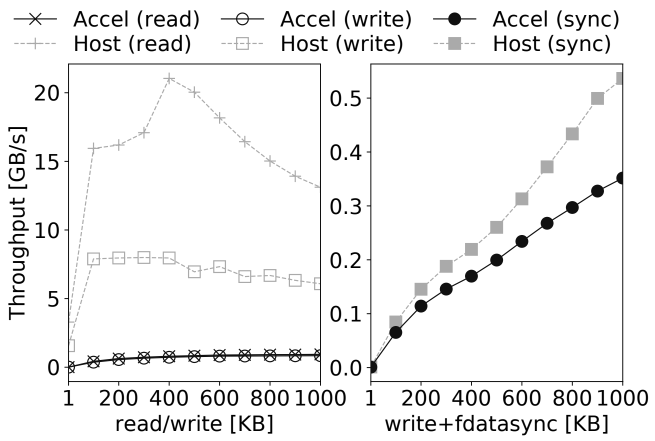

Figure 2: **I/O Throughput in Accelerator and Host.** In the accelerator, `read`, `write` and `write+fdatasync` are 92–99%, 86–99% and 21–34% slower, respectively.

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 the reference ones, because they contain extra data such as quality scores.

measure system call latency inside the accelerator using a tight loop of cheap `fcntl`. 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 left, the light gray arrow indicates the data movement for a `read()` system call. The host kernel reads data from the disk, then moves it to its user space where the AOS runs. The AOS then programs an accelerator DMA engine through a kernel module such that the accelerator can transfer the data to its memory.

Figure 2 plots throughput of read, write and sync operations that run synchronously in a single thread. As a highlight from the left figure, in order to read 1 MB of data from the kernel buffer cache using a single `read()` system call, we observe that it takes 1087 µs (1000000/(0.9197 × 1000)) in the accelerator, while it takes only 77 µs in the host, demonstrating the data copy and transfer costs. These costs are relevant even in the presence of access to the storage medium. As in Figure 2 right, writing 1 MB of data to the NVMe SSD using a `write()` followed by `fdatasync()` needs 2937 µs on the accelerator, while it takes 1847 µs on the host. Since the throughput in the accelerator is constrained by both storage medium and data transfers, the difference from the host case becomes larger as the I/O size increases.

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

Table 2: **End-to-End Genome Sequence Matching Time.** Each step uses general-purpose cores (gen.), vector processor cores (vec.) or both. The application spends most of the time to load the target data. See Table 1 for the size of data.

2.3 Case Study: Genome Sequence Matching

Attractive use cases of SoC-based accelerator devices are 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 is 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.

The genome sequence matching application can be executed end-to-end inside the accelerator thanks to its general-purpose cores alongside the specialized ones. It first reads the reference sequence in small batches (e.g., 64 KB each) using the general purpose cores, wherein delimiters are located using the specialized cores to speed up the later indexing process. The application then *indexes* the outputs, so that the matching process can access arbitrary regions (base pairs) of the sequence in 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. It should be noted that loading and saving data involves, respectively, `read` and `write` system calls redirected to the host.

Table 2 shows the breakdown of the end-to-end execution time and which of general purpose or vector processing cores are used in each step. Since the application uses the advantages of the vector processing engine for locating delimiters, indexing sequences and matching the index and target, it spends most of the time (72–96%) to read the target sequence data. The time to read the reference sequence of killfish is shorter than that of bacteria because of the larger number of delimiters in the bacteria data. We conclude that the storage I/O inefficiencies significantly impact on the end-to-end latency and throughput of such accelerator applications.

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

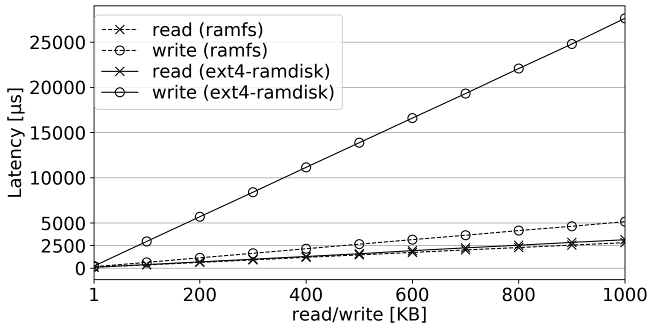

Figure 3: **I/O Latencies of LKL in the Accelerator**.

3 Design Space

Since redirecting storage I/O requests and data to the host comes at a significant cost and 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 our SoC-based accelerator has no kernel context, we cannot consider this design option.

Another option is to run a library operating system that 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 high in the form of library OS, because many kernel features, such as interrupts, kernel threads and synchronization primitives, must be emulated.

Nevertheless, in order to see the feasibility of a general purpose library OS in the accelerator, we ported the Linux Kernel Library (LKL). Figure 3 plots the read and write performance on `ramfs`, which is mainly constrained by the virtual file system and system call emulation because of the simple file system logic that does not implement any persistence property or block device organization, and that on `ext4` formatted on a `ramdisk`, which indicates the overheads of a modern file system logic in comparison to the `ramfs` performance. Since the latencies of 107–5310 µs to read or write 1–1000 KB of data on the `ramfs` are clearly high, we reject the option to run LKL in the accelerator. Further, the high write overheads with `ext4` imply the need for a file system designed specially for the accelerator.

The other option is buffer cache sharing between the host kernel file system and the accelerator device, as with GPUfs [21] and `libveaccio` [5]. 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. Since the SoC-based accelerators, unlike GPUs, are equipped with general purpose cores and thus can execute the entire application code, we examine different approaches where the accelerator device directly accesses the storage medium to minimize the host CPU usage.

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 stores and genome sequence analysis applications. This approach enables *bottom-up* design, taking into account microarchitectural weaknesses 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.

The right side of Figure 1 illustrates our architecture which we call Hayagui. The main components are the Direct I/O Engine that enables direct access to the storage medium, the 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.

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 [15]; Intel SPDK [7], another user-space storage device driver for high performance, was also considered, 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 the `uio` framework to grant PCIe register access to the user-space context of the accelerator. Although the SSD device registers are mapped into the host address space at the system boot time, the direct I/O engine can access these registers using PCIe APIs provided by the accelerator vendor (`ve_register_pci_to_vehva()` on NEC SX-Aurora [14]). 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 policies, such as access control, to accelerator applications. We implement a DMA buffer allocator used by UNVMe, because its default allocator, which is the `vfio` framework, is unavailable in the accelerator. Our allocator is based on a buddy system, because it is trivial to map a physically contiguous memory region into the user space in our accelerator. Our allocator also relies on the APIs to register DMA buffers provided by the vendor (`ve_register_mem_to_pci()` [14]).

Accelerator File System. As we observed in Figure 3, file system design appears crucial in the accelerator environment. 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 it manage read or write buffer caches.

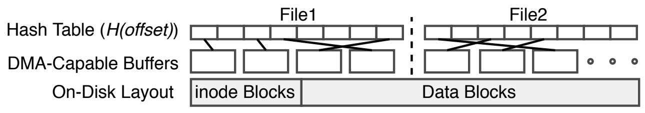

Figure 4: **Accelerator File System.**

Our accelerator file system (AccelFS) currently 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, as well as on-disk data management that we identified as important in Section 3.

Key Value Store Interface. Since many applications today use key-value stores as their storage backend, we ported the 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 ones. Each level has one or more *sstables*, where each contains keys and values sorted for search efficiency. When the highest level becomes full, the *compaction* process reorganizes the sstables into the next level by creating new ones. New data is stored in a *memtable* backed by DRAM and write-ahead log is 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, instead, our AccelFS supports these files, persisting and caching the data as LevelDB wishes.

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

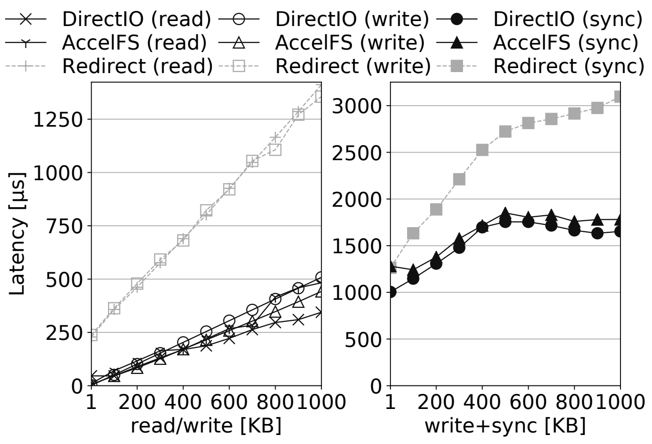

Figure 5: **I/O Latency of Direct I/O Engine and AccelFS.** Direct I/O Engine and AccelFS achieve lower latencies than Redirect (Direct I/O: 74–87% in read, 62–99% in write and 20–46% in sync. AccelFS: 64–95% in read, 67–97% in write and -1–42% in sync).

5 Experimental Results

This section presents how our architecture improve storage performance in a series of experiments. Our prototype is publicly available at `https://github.com/liva/hayagui`.

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 [13].

How does the direct I/O architecture improve storage I/O performance? Figure 5 plots latency of read, write or sync operation measured by a single threaded tight loop; next request starts after the completion of the previous one. In the DirectIO cases, reads always read data from the storage medium because buffer caches are not available. Writes issue I/O requests, but persistence is not guaranteed. Syncs explicitly flush written data, thereby taking much longer than the other operations. In the AccelFS cases, reads always read new data from the storage medium, although it looks up the buffer caches (and fails). In the system call redirection cases, we use `read`, `write` and `write+fdatasync`. The key take away is that direct I/O significantly improves performance in all the operations, and AccelFS incurs little overhead.

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 that HAYAGUI improves the request latencies by 33–81%. 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 the requested data available in the memtable of LevelDB.

As we have understood the performance of individual operations, we now evaluate HAYAGUI against realistic workloads using the C++ variant [19] of YCSB [4] benchmark tool running inside the accelerator. Figure 7 plots the results and we observe 12–89 % of improvements. Therefore, we conclude that, despite the additional software overhead, 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 thus 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.

Figure 8 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

Near storage computing. INSIDER [20], CognitiveSSD [10] and X-Engine [23] implement storage controller, deep learning processing and LSM-tree compaction in FPGA, respectively. Morpheus [22] implements data deserialization inside the SSD device using ARM embedded cores. These approaches are complementary to HAYAGUI, whose code that runs in the general purpose cores would utilize their host-side APIs and library. We have already discussed the direct I/O approaches for GPU applications in Section 3.

Accelerator management systems. iPipe [11] schedules applications across SmartNICs and host CPUs based on portable context descriptors. However, it does not support direct device access, and relies on the host-side process when storage access is needed. DLibOS offers a library OS abstraction to accelerator applications, but it does not support storage devices. M^3X [1] allows the accelerator applications to access storage and network stack directly via DMA, but relies on custom hardware. Further, it does not consider performance problems with general purpose cores of accelerator devices.

Accelerator applications. Applications currently explored to benefit from SoC-based accelerators include bioinformatics applications, which are featured in this paper, scientific computing [8], machine learning [17], data analytics [17], query processing of relational database [6] and Network Function Virtualization applications [9]. All of these applications, ex-

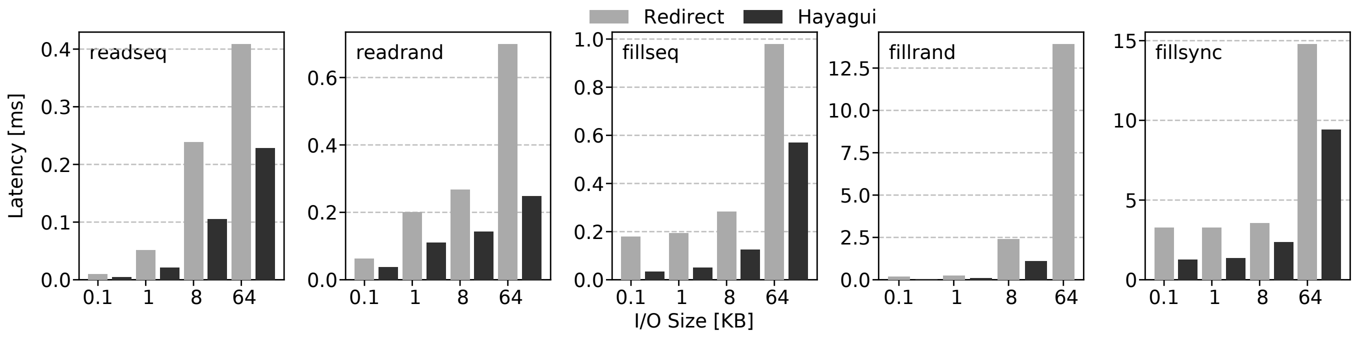

Figure 6: **db_bench Results.** Workloads represent sequential/random reads/writes, and sync to the storage medium. HAYAGUI improves the baselines by 32–80%. In 0.1 KB requests, readseq and fillrand are invisible but improved by 49 and 79%, respectively.

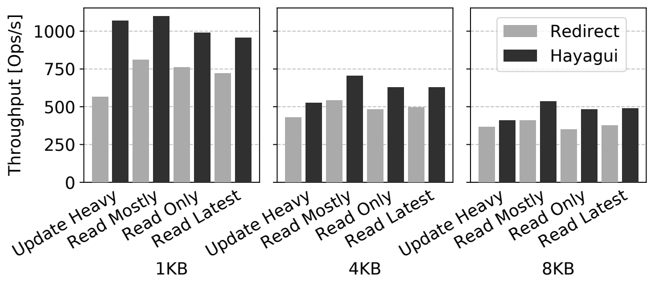

Figure 7: **YCSB Results.** HAYAGUI outperforms the system-call redirection baselines by 12–89%.

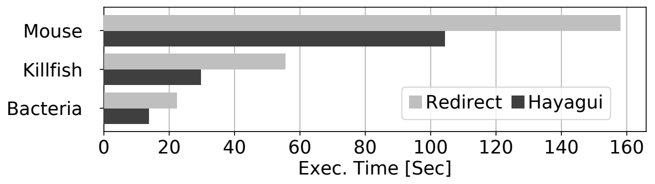

Figure 8: **Genome Sequence Matching Performance.** HAYAGUI reduces the end-to-end execution time by 33–46% or by 8–53 seconds. Data sets are are 2.1–15.0 GB in size (see Table 1 in Section 2.3).

cept for the last two ones, would need to handle large data, which therefore would benefit from HAYAGUI.

7 Conclusion

HAYAGUI is the first system that runs the whole storage stack on the general-purpose cores of a SoC-based accelerator to increase the overall system processing capacity by minimizing the data movement and saving as much resources as possible for other tasks or applications executed on the host-side CPUs. We explored a direct storage I/O architecture in designing HAYAGUI, and empirically demonstrated its efficiency and usefulness using microbenchmarks and a bioinformatics application. The next section discusses further research questions.

8 Discussion

SoC-based accelerators introduce many problems and opportunities to build or use storage systems, as discussed below.

Feasibility of direct storage access architectures. In this paper we used a NEC SX-Aurora Vector Engine equipped with general purpose cores, and enabled direct storage I/O using vendor-specific PCIe APIs (Section 4). Some other accelerators such as SmartNICs would allow for similar organization, but what about the future 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 have already observed notable limitations, such as slow `malloc` and atomic operations. How could we overcome such weakness that would be prevalent also in other accelerator devices? Further, although some accelerators, such as high-end SmartNICs, have rather beefy CPU cores [11], 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 programming models 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. Since it would be painful if we'd need to build an efficient storage stack for every accelerator from scratch, 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?

Acknowledgments

We are grateful to anonymous reviewers for their insightful comments and Hajime Tazaki at IIJ for his help with LKL experiments.

References

[1] N. Asmussen, M. Roitzsch, and H. Härtig. “M³x: Autonomous Accelerators via Context-Enabled Fast-Path Communication”. *Proc. USENIX ATC*. Jul. 2019.

[2] A. Barbalace, B. Ravindran, and D. Katz. “Popcorn: a replicated-kernel OS based on Linux”. *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”. *Proc. ACM SOSP*. 2009.

[4] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. “Benchmarking cloud serving systems with YCSB”. *Proc. ACM SoCC*. 2010.

[5] E. Focht. *VE Accelerated IO*. https://sx-aurora.github.io/posts/accelerated-io/. Mar. 2019.

[6] D. Habich, P. Damme, A. Ungethüm, J. Pietrzyk, A. Krause, J. Hildebrandt, and W. Lehner. “MorphStore-In-Memory Query Processing based on Morphing Compressed Intermediates LIVE”. *Proc. ACM SIGMOD*. 2019.

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

[8] 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”. *Proc. ACM/IEEE SC*. 2018.

[9] Y. Le, H. Chang, S. Mukherjee, L. Wang, A. Akella, M. M. Swift, and T. Lakshman. “UNO: uniflying host and smart NIC offload for flexible packet processing”. *Proc. ACM SoCC*. 2017.

[10] S. Liang, Y. Wang, Y. Lu, Z. Yang, H. Li, and X. Li. “Cognitive SSD: A deep learning engine for in-storage data retrieval”. *Proc. USENIX ATC*. Jul. 2019.

[11] M. Liu, T. Cui, H. Schuh, A. Krishnamurthy, S. Peter, and K. Gupta. “Offloading Distributed Applications Onto smartNICs Using iPipe”. *Proc. ACM SIGCOMM*. 2019.

[12] M. Liu, S. Peter, A. Krishnamurthy, and P. M. Phothilimthana. “E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers”. *Proc. USENIX ATC*. Jul. 2019.

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

[14] *Mechanisms for VE specific system calls for VEOS on SX-Aurora TSUBASA*. https://github.com/veos-sxarr-NEC/libsysve/blob/feature-vepci-v2.2/doc/VEPCI.md.

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

[16] 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.

[17] NEC. *Machine Learning and data analytics MW for SX-Aurora TSUBASA*. https://www.hpc.nec/events/isc-18/aurora-forum/presentations/3_AuroraForum_MLMW4Aurora_Takeo_Hosomi.pdf.

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

[19] J. Ren. *YCSB-C*. https://github.com/wujy-cs/YCSB-C.

[20] Z. Ruan, T. He, and J. Cong. “INSIDER: Designing In-Storage Computing System for Emerging High-Performance Drive”. *Proc. USENIX ATC*. Jul. 2019.

[21] M. Silberstein, B. Ford, I. Keidar, and E. Witchel. “GPUfs: integrating a file system with GPUs”. *ACM SIGPLAN Notices*. 4. 2013.

[22] H.-W. Tseng, Q. Zhao, Y. Zhou, M. Gahagan, and S. Swanson. “Morpheus: creating application objects efficiently for heterogeneous computing”. *Proc. ACM/IEEE ISCA*. 2016.

[23] T. Zhang, J. Wang, X. Cheng, H. Xu, N. Yu, G. Huang, T. Zhang, D. He, F. Li, W. Cao, Z. Huang, and J. Sun. “FPGA-Accelerated Compactions for LSM-based Key-Value Store”. *Proc. USENIX FAST*. Feb. 2020.