Computational Storage: Where Are We Today?

Computational Storage: Where Are We Today?

Antonio Barbalace
The University of Edinburgh
antonio.barbalace@ed.ac.uk

Jaeyoung Do
Microsoft Research
jaedo@microsoft.com

ABSTRACT

Computational Storage Devices (CSDs), which are storage devices including general-purpose, special-purpose, and/or re-configurable processing units, are now becoming commercially available from different vendors. CSDs are capable of running software that usually runs on the host CPU – but on the storage device, where the data reside. Thus, a server with one or more CSDs may improve the overall performance and energy consumption of software dealing with a large amount of data.

With the aim of fostering CSD's research and adoption, this position paper argues that commercially available CSDs are still missing a wealth of functionalities that should be carefully considered for their widespread deployment in production data centers. De facto, existing CSDs ignore (heterogeneous) resource management issues, do not fully consider security nor multi-user, nor data consistency, nor usability. Herein, we discuss some of the open research questions, and to what degree several well-known programming models may help solving them – considering also the design of the hardware and software interfaces.

1 Introduction

Computational Storage (CS) is a type of near data processing [16] architecture that enables data to be processed within a storage device in lieu of being transported to the host central processing unit (CPU) [12]. Figure 1 generalizes several CS architectures investigated by SNIA [11].

CS architectures introduce numerous advantages: a) unloading the host CPUs – thus, a cheaper CPU can be installed, or the CPU can run other tasks; b) decreasing data transfers, and increasing performance – only essential data need to be transferred from the storage to the CPU, general- or special-purpose processing elements or reconfigurable units on the CS device(s) may process data instead of the CPU, even in parallel; c) reducing energy consumption – a storage device on PCIe cannot consume more than 25W in total [41], thus processing units on computational storage devices (CSDs) consume just a fraction of it, versus the power consumption of a server-grade host CPU, which floats around 100W; d) preserving data-center infrastructure expenditure – i.e., scaling data-center performance without requiring investments in faster networks.

11th Annual Conference on Innovative Data Systems Research (CIDR '21) January 10-13, 2021, Chaminade, USA.

While research on in-storage processing on HDDs [12, 34] and SSDs [37, 31, 29, 42] has been carried on since the 1990's and 2010's, respectively, only recently CS platforms become commercially viable with a few companies already selling SSDs with CS capabilities – e.g., Samsung [9], NGD [6], and ScaleFlux [10]. Despite CSDs' market appearance, these devices are cumbersome to program and reason with, which may hinder their wide adoption. In fact, there is no software nor hardware support for heterogeneous resource management in CSD, nor security, consistency and general usability consideration.

Based on the authors experience working on several academic and industry CS prototypes in the latest years, this paper is an attempt at reviewing the state-of-the-art, listing the most pressing open research questions with CSD, and analyzing the suitability of different programming models in answering such questions – without forgetting about the hardware/software interface that is still not CSD ready. This work focuses on a single direct-attached CSD, with storage and compute units resident on the same device. However, we believe the same findings would apply widely, such as to smart disk array controllers. Additionally, the work generically looks at CSD with general-purpose CPUs, special-purpose CPUs, as well as CSD with re-configurable hardware (FPGA). Hence, we refer to all of those as "processing units" in the rest of the paper.

Briefly, our conclusion is that hardware and software for CSD is not ready yet, and more have to be done at the hardware and software level to fully leverage the technology at scale.

2 Background and Motivation

Computational storage *reduces the input and output transaction interconnect load* through mitigating the volume of data that must be transferred between the storage and compute planes. As a result, it stands to better serve modern workloads, such as high-volume big data analytics or AI tasks with faster performance [27], to improve data center infrastructure utilization [29], together with many other benefits. We discuss several below.

A primary benefit of computational storage is *faster and more energy-efficient data processing.* Computational storage architectures offload work usually processed by host compute elements – CPU and eventual accelerators, to storage devices. Without CS, for example in the data analytics context, a request made by the host compute elements requires that all data from a storage device be transferred to it. The host compute elements must then thin down the data prior to performing their designated task. In a CS approach, the storage device takes an initial step of qualifying data based on its relevance following an host request – before moving the data to the main compute tier to be processed. Thus, possibly reducing the amount of data to be moved and processed

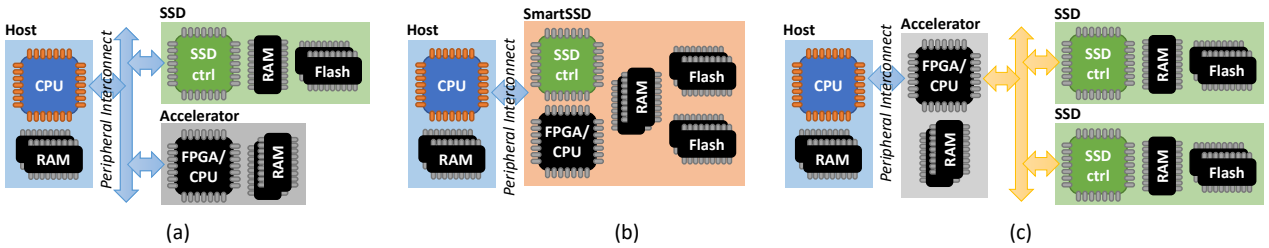

Figure 1: A few architectures of computational storage devices considered in the SNIA CS SIG [43]. (a) An FPGA and/or a CPU (with DRAM, but without storage media) sits along with an SSD on a peripheral interconnect; (b) An FPGA and/or a CPU is bundled together within an SSD (including SSD controller, RAM and storage media); (c) An FPGA and/or a CPU (with DRAM, but without storage media) on a peripheral interconnect connects to one or more SSD via another interconnect.

by the host. The reduced host compute instructions per workload means that the host CPUs, and eventual accelerators, have more processing power available to support other workloads.

Another benefit of computational storage is that it *makes a shared storage environment more beneficial to the most performance-hungry workloads*. Typically, a direct-attached storage approach is used to serve these workloads to avoid storage network latency, but also to increase throughput by spreading the data across many devices. However, this often results in resource underutilization, and also introduces further delay due to the need of searching multiple devices for the relevant data. In contrast, CS allows applications to be executed into each storage device, at the same time. This provides a level of parallel processing to enable a microservice-like approach to running those applications across all the individual devices. Such ability to process the data simultaneously greatly reduces the time to locate the data and provides the host the results needed.

Computational storage can also help *leveraging the existing network infrastructure for much longer*, as well as to truly scale next-generation networks. Because computational capabilities enable the storage to work on a larger data set first, it leverages higher I/O capabilities of modern SSDs and avoids performance being restricted by a network. As a result, the network interconnect is less critical with computational storage. Thus, computational storage stands to add value by enabling multiple applications' performance to be accelerated on the same infrastructure, while at the same time optimizing utilization of infrastructure resources across the stack.

2.1 State of Hardware and Software

Most SSDs have dedicated processing and memory elements – i.e., embedded multicore CPU or FPGA, and DRAM, other than flash memory. Those are used to execute read, write, and erase commands on user data, as well as flash management functions.

With available computing resources on an SSD, several projects [31, 42, 29, 12, 38, 25, 35, 14, 28, 46, 36, 39, 45, 44, 24, 35, 13, 23] explored opportunities to run user-defined data-intensive compute tasks, such as database operations, in the SSD device itself. While performance improvements and energy savings were observed, several challenges prevented the broad usage and adoption of computational SSDs. First, the processing capabilities available were limited by design: low-performance embedded processors, or resource constrained FPGAs, and the high-latency to the in-storage DRAM require extra careful programming to run user-defined code to avoid performance limitations. Second, a flexible and generic interface and programming model to easily execute on an SSD user-defined code written in a high-level programming language (such as C/C++) were never defined. Additionally, the programming model also needs to support the concurrent execution of various in-storage applications with multiple threads to make it an efficient platform for complex user applications. Nevertheless, interfaces at different layers of the hardware and software have to be defined on the host and CSD to make a platform actually usable. Third, none considered how to handle multiple users.

Such works have been carried on a multitude of different hardware and software. To the best of our knowledge, as of today there are just two CSD development boards openly available: the OpenSSD [1] from Hanyang University (implementing the architecture in Figure 1.b), and the DFC [26] developed by a collaboration of DellEMC and NXP (implementing the architecture in Figure 1.c). Both feature a multicore ARM general purpose-processor, accelerators, and re-configurable hardware, other than a sort of flash media and DRAM. Despite open-source, they are based on old standards and are not supported anymore. On the other hand, storage vendors use their own SSD prototypes [31, 38, 15] and FPGA boards have been used for research [42, 46]. Simulation environments from the academic community exist as well [30, 8, 15], but lack support.

Very recently, CSDs become available on the market, including NGD in-situ processing SSD [6] that features multiple 64bit ARM cores, ScaleFlux CSD [10] and Samsung SmartSSD [9] based on FPGA (implementing the architecture in Figure 1.b). Other products, such as Eideticom NoLoad Computation Storage Processor [5], implement storage and compute on different PCIe boards, as depicted in Figure 1.c, enabled by p2p DMA and NVMe's CMB [21]. These products triggered SNIA to working on a proposal to extend the NVMe protocol for CSD [11] – which at the moment is work in progress. Hence, we believe that something should be done soon in order to make computational storage devices more compelling to the masses. Notably, mainly due to companies IP matters, current CSD products provide very limited customizability, and therefore, cannot be easily used for research purposes, such as interface or programming model exploration.

3 Open Research Questions

We believe that showing improved performance and energy reductions is not enough to persuade the widespread adoption of CSD. Hence, herein we present a variety of open research questions identified within existing CS technologies, involving a) resource management, b) security, c) data consistency, and d) usability.

3.1 Resource Management

A server with one or more CSDs is per-se a single system. But from the software point of view, since each (group of identical) processing units, on the motherboard or on any CSD, runs its own software stack, such server looks like a distributed system. Resource management in single systems as well as in distributed systems is fundamental to provide efficient and fair usage of hardware resources, including processing units, memory, and storage. This, for example, implies that at any given time no single resource is overloaded nor applications are starving for resources. Similarly, a single user shouldn't monopolize the usage of all resources. Resource management policies may be needed to balance the workload among compute resources in order to meet performance and power goals. However, this is lacking in emerging CS architectures and in related works.

Questions. a) Where to take resource management decision? On the host CPU, on a CSD, or on both? b) When there is replication among CSDs, which replica maps a certain computation? c) How to maintain workload and energy consumption information across available processing units on the host and the CSDs?– especially, with the aim of taking resource management decisions? d) What to do when CSDs are overloaded? Should the overloaded CSD notify the host or other CSDs, and then what? e) How to provide fairness of resources (such as CPU cycles, FPGA real-estate, memory/flash space, and flash-channel traffics) among different users? f) How to map data to multiple CSDs, and/or to different flash channels inside a CSD? g) For applications that do use a single CSD, what is the break-even point between running on the host CPU and on the CSD? (Is this workload dependent?)

3.2 Security

When a single SSD stores data from different users it is fundamental to deny users to access each other's data. With today's SSDs, there are two different approaches for controlling data access. The first is to use a file system – each file has an owner, etc. The second is to use either hardware virtualization (SRIOV) or NVMe namespaces [7], in order to assign different parts of the storage to different owners. Despite the existence of protection mechanisms, the code running on the CSD's processing unit (CPU, accelerator, FPGA) has potentially access to all data stored on the flash chips. When using a file system for access control, it is fundamental to define secure mechanisms and techniques to maintain the same concept of users for the software running in the CSD, as well as the software running on the host CPU. This is due to the semantic gap between the storage device and software running on the host CPU – the host CPU knows about the file system, and users, but that is not always true for the storage device. It is worth noting that users' knowledge can be asymmetric, in the sense that a CSD doesn't need to know all users' details as the host CPUs in most of the cases. The code running on the CSD should access only what it is allowed to access.

Another problem is trusting the identity and integrity of the reconfigurable hardware, software, and firmware on the CSD itself. Assuming a method to only install proper firmware and systems software exists, it is fundamental that only and exclusively user-submitted code runs on the CSD other than that. Moreover, user-submitted code should not alter the integrity of the firmware and systems software on the CSD.

Questions. a) How to isolate multiple applications among each other on a CSD? b) How to make them safe from side-channel attacks, denial of service, etc? c) Diverse programming models require different isolation techniques. For example, hardware virtualization can be used to isolate different software stacks. d) How much does software isolation cost? Does this cost overshadow the benefits? e) How to make sure the code running on the CSD is legit? Not just at boot, but also during runtime?

3.3 Data Consistency

When data is read and written by multiple parties, consistency problems may arise. For example, the same block of data can be concurrently read by the host and CSD's CPUs, and an application running on one of these two modifies the block's content. After such modification, which is unilateral, each CPU would operate on different data – while assuming the data is the same. In fact, immediately after a CPU modifies the data, it should inform the other. The same issue applies not just to file content but also to file system meta-content. For example, when the software running on the host CPU or on the CSD's CPU creates a new file, the creation must be notified to the other CPU. The same extends to almost all file system operations. Obviously, classic file systems for personal computer do not address such issues. Furthermore, common disk interfaces (e.g., SATA, SAS, NVMe) are not prepared to manage such situations as well — interfaces were built assuming a disk controller strictly executes commands from the host CPU. Similarly, data may be replicated or sharded among several disks. For consistency, modifications on replicated or sharded data should happen in parallel – the same applies to erasure-code or parity blocks [13].

Finally, a single CSD may fail during any operation, including in-storage data processing. Replication, sharding, and parity/erasure coding may definitely be used to tolerate failures other than improving performance. Such techniques are likely be implemented atop CSDs. However, to support that, CSD may need to implement additional features , for example, to communicate failures to the host CPU or other CSDs.

Questions. a) How to use the same file system on the host and on a CSD, consistently? b) What about file system changes? How to update the software on the host about file modifications on the CSD? c) Is a new file system needed? Can classic file systems be extended to support that? Do distributed file systems solve this problem already? If yes, what about performance? d) How to extend current storage interfaces to provide notifications for data changes into the CSD? Are notifications needed at all? e) What about explicit transaction management? f) What about hardware and software failures? g) What if there is replication and a new file is added, how to deal with that? h) How CSDs communicate between each other for replication or any other operation that requires coordinated operations on multiple drives? i) Does communication need to go via the host CPU or should it go directly (e.g., via the P2P DMA/RDMA) – which one is better?

3.4 Usability

To widen adoption, usability is certainly of fundamental importance. CSDs should be easy and quick to program, deploy, and debug at any level of software (user-level or kernel-level).

Debugging applications running among the host and CSD's CPUs shouldn't be a nightmare. For example, one of the most complicated steps after the development of a distributed software is the debugging – this is exactly because in those environments programmers end up having multiple debuggers, or log traces coming from different systems, which have to be synchronized to be useful and identify the source of the problem. Debugging in distributed systems may provide some hints.

A major matter is if a CSD should be considered as a totally independent computer node, or as a part of a computer

node? We believe that it strongly depends on the available hardware resources on the CSD – if the computational and memory resources are comparable to the ones available the host, users may be able to run the same workload atop the host and CSDs. However, if the resources are not the same, CSDs should be considered as storage-side accelerators – thus, a full-fledged workload should be executed on the host CPU only. Based on that, a decision on what workload(s) to run on the CSD may be taken by the software running on the host CPU or by a data-center scheduler. However, in the latter case, a CSD should be reachable via network, which requires the CSD to be built with a network interface hardware at an additional cost.

Questions. a) Beyond programmability, how to easily deploy applications on CSDs? b) Can applications be mapped to storage devices transparently? (i.e., Without the applications requiring to know on what storage device to be run.) c) Which information is needed for a resource manager from the application to make the best placement decision? d) How to design an easy to use API that minimizes application modifications? Would a POSIX-like API be easily adopted?

4 Programming Model

Clearly, the programming models available to programmers who want to run his/her application among the host and CSD processing units largely affect the way the questions in Section 3 are answered. We believe that there is no "one-size-fit-all" programming model that work for all kinds of applications, and therefore a quantitative and qualitative analysis of each model is sought. In this section, we discuss a few programming models, and describe for each model how we envision the above research questions would be answered, which is summarized in Table 1.

4.1 Dataflow

Within the dataflow model (e.g., [31, 32]), a sequence of transformation operations is defined for each chunk of data in transit. A transformation operation receives a chunk of data in input, and outputs the transformed chunk of data. In general, "data in transit" identifies blocks of data travelling between different hardware and software layers, including not only blocks of data transferred from flash chips to the CSD/host CPUs, but also blocks of data flowing through the different kernel's and applications' software layers.

When used with CSD. Mapping the dataflow programming model into a CSD environment is straightforward: for each storage command (i.e., disk block read or write), associate one or a set of transformation operations, similarly to [19, 13, 40]. The communication between the host and the CSD's processing units could be handled by extending standard interfaces (e.g., SATA, SAS, NVMe). Extensions include new commands to download transformation operations or modifications to the existent commands to exchange per-session or global data between the host and the CSD's CPUs.

The dataflow model handles well most of the four concerns listed in Section 3. The beauty of this programming model is that operations can be defined at a fine granularity and executed anywhere (the host CPU or CSD's processing units), simplifying resource management. Operations may be moved back to the host for load balancing or can be replicated if data is replicated, transparently. In addition, multiple operations can be merged, split, as well as parallelized. When parallelized, access to per-session or global data must be protected for consistency. For security, a method to associate data with users must be introduced. Finally, dataflow programs can be defined in any language, although a language that provides some sort of formal properties, such as termination and memory safety, is preferred.

Unfortunately, only a bunch of applications are implemented within this programming model, which requires program rewriting when an application was written within the client-server or shared memory model. Therefore, research on compiler tools that automatically convert applications into dataflow is sought.

4.2 Client-Server

This includes applications developed and deployed within clusters, such as the ones based on Message Passing Interface (MPI), Remote Procedure Call (RPC), MapReduce, etc. The only requirement for these applications is a network connection between multiple compute nodes – TCP/IP, UDP, or RDMA are the most common. Such applications are strictly partitioned in multiple programs, each of which runs on a different processing node.

When used with CSD. Applications developed with this programming model can be directly mapped into a CSD setup without any modification – assuming the application can run in the software environment provided by the CSD. This is achieved by establishing a network channel between the host and CSD CPUs, and/or amongst CSD CPUs [6]. However, native compiled applications may need to be recompiled to the instruction set architecture (ISA) of the target CSD's CPU (such as ARM). Although not impossible, this process can be very convoluted. In fact, it may require an entire toolchain, and the recompilation of all libraries required by the program – this is because not all libraries are available for all ISAs. Moreover, many of such distributed applications are based on a very large software base, involving several different libraries. This at runtime usually consume a lot of memory (other than storage) limiting the actual number of applications that can simultaneously run on the CSD, and therefore resulting in degraded performance.

With the client-server model, the resource management granularity is at the program level — this is because only programs can be moved between CPUs (assuming the same program is available for all CPU ISAs present in the system). Algorithms are embedded in the different programs building the application; thus, a resource manager/scheduler cannot act at a finer granularity, hindering automatic optimizations.

Security is provided at the application-level, and applications can be eventually embedded in containers (OS-level virtualization). Standard OS techniques can be used to enable CSD's application to only access data belonging to a specific user.

4.3 Shared Memory

The shared memory programming model is widely adopted on multicore processors. It requires a form of (consistent) shared memory between processing units. When hardware shared memory is not available, software shared memory, or distributed (virtual) shared memory (DSM), may be used [20, 33] . Another assumption of the shared memory programming model is that all CPUs are identical, or at least implement the same ISA.

When used with CSD. The host and CSD's CPUs may not have the same ISA. In fact, many of the existing deployments are characterized by x86 CPUs on the host and ARM CPUs for CSD. Academic projects, such as Popcorn Linux [22, 18] and H-Container [17], enable applications developed for shared memory multicores to run on heterogeneous ISA cores, transparently – without any application modification. This includes starting an application on the host CPU and then migrating all its threads to CSDs.

In this model, resource management can be potentially done at the finest granularity of an assembly instruction. In that way,

	Resource Management	Security	Consistency	Usability
Dataflow	Automatic, very efficient	Very good, with low overhead	Controlled by resource manager	Application dependent
Client-Server	Quite inefficient	Program-level	Must be handled by programmers	Very good (just reuse apps, or at most recompile)
Shared Memory	Automatic, very efficient	Thread/program-level (but also basic block)	Transparent Software consistency, maybe expensive	Very good (just reuse apps) but needs special system software

Table 1: A qualitative summary and comparison of programming models vs. research questions described in Section 3.

the computation can swing between the host CSD's CPUs at any time, based on, for example, the compute-intensiveness and where the data is.

The issue with this programming model is that when hardware shared memory is not available, it should be provided by software, which may be expensive. However, future PCIe advancements may offer consistent shared memory among a device's CPU on PCIe and the host CPU [2, 3, 4]. In addition, the shared memory programming model also needs to take care about the consistency problem, and computation in this model cannot be parallelized or optimized by an external resource manager. Finally, security can be achieved via the approaches available in classic operating systems (such as using file systems or hardware virtualization/NVMe namespaces descrbed in Section 3.2), but further research is sought.

5 Storage Interface

In this section, we review the hardware and software interface implications of each programming model. Currently there are different types of storage interfaces that are widely adopted in industry and research communities (e.g., block-based, file-based, and object-based interfaces), but we believe our findings apply to any of these interfaces.

5.1 Hardware Interface

The ***dataflow programming model*** requires limited storage interface modifications. As discussed in Section 4.1, a transformation operation should just be assigned to a dataflow, and for each data block of the dataflow the operation is called. Therefore, the minimal requirement to today's storage interfaces, such as NVMe, is to provide additional commands to: 1) download a program; 2) attach/detach a program to a dataflow; 3) define a dataflow; 4) debugging/logging. All such commands can be implemented by simply extending the exiting NVMe protocol/interface as supporting legacy software. Note that with this interface, developers don't need to care about which software stack is running on the CSD's CPUs, e.g., it can be Linux, or be a firmware.

Server-client programming models may also require limited storage interface modifications. As described in Section 4.2, only a message-passing channel to emulate network communication is necessary – which can be implemented with sending and receiving queues (similarly to RDMA, NVMe, etc). Such change may extend the NVMe protocol/interface by adding a new command set to it, which includes commands not only for the message communication, but also for downloading, running, and monitoring in-storage programs.

The shared memory programming model (Section 4.3) is per-se the one that demands more changes to the current storage interfaces when a performant implementation is needed, i.e., no software DSM overheads. If that is not needed, the same interface modifications required by the client-server programming model are demanded. Such modifications are enough to implement software DSM. Instead, when a performant implementation is needed, a sort of hardware shared memory should be available among the host CPU and the CSD's CPUs, provided by new coherent peripheral bus interconnects [2, 3, 4], for example. Hardware shared memory does not have to be consistent – the consistency can be provided via software.

5.2 Software Interface

An application written within the ***dataflow programming model*** doesn't require any specific software interface – in fact, programs declare the input source as well as the output source that is a stream of data from or to storage media. An application developer is not directly exposed to any CSD-specific hardware interface, and a runtime system shields programmers from these technicalities.

When an application is written instead within the ***server-client*** or ***shared memory programming model***, it is expected to directly interface with the flash array when executing on CSDs. Note that this necessary for performance. Hence, a software interface to access the flash array should be defined. A naïve solution is to abstract each different flash array's channel with a UNIX device. Thus, each user may be assigned with a different channel. Despite a practical solution, which maps quite well with the underlying hardware, and can be used for protection/isolation, it is highly unlikely that different users are assigned with different NAND channels for performance reasons – this is because writing and reading several channels in parallel gives high performance.

Moreover, an user accesses data on storage via files. Files are a file system abstraction, usually provided by the host operating system, which is not necessary known on the SSD. Therefore, a more fine-grain solution is needed for protecting/isolating the data of different users, which requires another abstraction for programmers. The concept of stream, such as a list of (non-sequential) flash blocks could be a solution, but existent hardware doesn't provide any related mechanism.

On the host side, the same CS hardware interface should be usable by software at the user- and kernel-level. This is to support traditional file system in kernel-space as well as modern storage software in user-space (e.g., SPDK). A "symmetric" kernel/user interface is needed.

Finally, in order to support scenarios where the code cannot be moved to the CSD, and therefore it should run on the host operating system, a symmetric software interface should be implementable on the host machine and on the CSD.

6 Concluding Remarks

In this paper, we briefly survey the state of the art of computational storage and we raised several open challenges that might need to be considered to facilitate the adoption of the computational storage technology in both research and industry communi-

ties – existing computational storage hardware and software are not ready to be used in production at scale. We then discussed how the most widely-used programming models with hardware/software interfaces can help solving such challenges. We believe that our discussions and lessons presented in this paper can provide a higher degree of clarity about what's likely to be needed for the mass adoption of the computational storage technology.

References

[1] The OpenSSD Project. http://www.openssd-project.org/, 2016.

[2] Cache Coherent Interconnect for Accelerators (CCIX). http://www.ccixconsortium.com/, 2017.

[3] Gen-Z Consortium. http://genzconsortium.org/, 2017.

[4] OpenCAPI Consortium. http://opencapi.org/, 2017.

[5] Eideticom. NoLoad Computational Storage Processor. https://www.eideticom.com/uploads/images/NoLoad_Product_Spec.pdf, 2020.

[6] NGD Systems. https://www.ngdsystems.com/, 2020.

[7] NVMe Specifications. https://nvmexpress.org/specifications/, 2020.

[8] OX: Computational Storage SSD Controller. https://github.com/DFC-OpenSource/ox-ctrl, 2020.

[9] Samsung. https://samsungsemiconductor-us.com/smartssd, 2020.

[10] ScaleFlux. https://scaleflux.com/, 2020.

[11] SNIA. Computational Storage. https://www.snia.org/computational, 2020.

[12] A. Acharya, M. Uysal, and J. Saltz. Active disks: Programming model, algorithms and evaluation. *ACM SIGOPS Operating Systems Review*, 32(5):81–91, 1998.

[13] I. F. Adams, J. Keys, and M. P. Mesnier. Respecting the block interface–computational storage using virtual objects. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, 2019.

[14] D.-H. Bae, J.-H. Kim, S.-W. Kim, H. Oh, and C. Park. Intelligent ssd: a turbo for big data mining. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, pages 1573–1576, 2013.

[15] A. Barbalace, M. Decky, J. Picorel, and P. Bhatotia. BlockNDP: Block-storage Near Data Processing. ACM/IFIP Middleware '20, New York, NY, USA, 2020.

[16] A. Barbalace, A. Iliopoulos, H. Rauchfuss, and G. Brasche. It's time to think about an operating system for near data processing architectures. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pages 56–61, 2017.

[17] A. Barbalace, M. L. Karaoui, W. Wang, T. Xing, P. Olivier, and B. Ravindran. Edge computing: the case for heterogeneous-isa container migration. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 73–87, 2020.

[18] A. Barbalace, R. Lyerly, C. Jelesnianski, A. Carno, H.-r. Chuang, and B. Ravindran. Breaking the boundaries in heterogeneous-isa datacenters. In *Proceedings of the 22th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, 2017.

[19] A. Barbalace, J. Picorel, and P. Bhatotia. Extos: Data-centric extensible os. In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '19, page 31–39, New York, NY, USA, 2019. Association for Computing Machinery.

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

[21] S. Bates and O. Duer. Enabling the NVMe CMB and PMR ecosystem. https://nvmexpress.org/wp-content/uploads/Session-2-Enabling-the-NVMe-CMB-and-PMR-Ecosystem-Eideticom-and-Mell....pdf, 2018.

[22] S. K. Bhat, A. Saya, H. K. Rawat, A. Barbalace, and B. Ravindran. Harnessing energy efficiency of heterogeneous-isa platforms. *SIGOPS Oper. Syst. Rev.*, 49(2):65–69, Jan. 2016.

[23] W. Cao, Y. Liu, Z. Cheng, N. Zheng, W. Li, W. Wu, L. Ouyang, P. Wang, Y. Wang, R. Kuan, et al. Polardb meets computational storage: Efficiently support analytical workloads in cloud-native relational database. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 29–41, 2020.

[24] S. Cho, C. Park, H. Oh, S. Kim, Y. Yi, and G. R. Ganger. Active disk meets flash: A case for intelligent ssds. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 91–102, 2013.

[25] A. De, M. Gokhale, R. Gupta, and S. Swanson. Minerva: Accelerating data analysis in next-generation ssds. In *2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 9–16. IEEE, 2013.

[26] J. Do. Softflash: Programmable storage in future data centers. https://www.snia.org/sites/default/files/SDC/2017/presentations/Storage_Architecture/Do_Jae_Young_SoftFlash_Programmable_Storage_in_Future_Data_Centers.pdf, 2017.

[27] J. Do, V. C. Ferreira, H. Bobarshad, M. Torabzadehkashi, S. Rezaei, A. Heydarigorji, D. Souza, B. F. Goldstein, L. Santiago, M. S. Kim, et al. Cost-effective, energy-efficient, and scalable storage computing for large-scale ai applications. *ACM Transactions on Storage (TOS)*, 16(4):1–37, 2020.

[28] J. Do, Y.-S. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt. Query processing on smart ssds: opportunities and challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1221–1230, 2013.

[29] J. Do, S. Sengupta, and S. Swanson. Programmable solid-state storage in future cloud datacenters. *Communications of the ACM*, 62(6):54–62, 2019.

[30] D. Gouk, M. Kwon, J. Zhang, S. Koh, W. Choi, N. S. Kim, M. Kandemir, and M. Jung. Amber: Enabling precise full-system simulation with detailed modeling of all ssd resources. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 469–481. IEEE, 2018.

[31] B. Gu, A. S. Yoon, D. Bae, I. Jo, J. Lee, J. Yoon, J. Kang, M. Kwon, C. Yoon, S. Cho, J. Jeong, and D. Chang. Biscuit: A framework for near-data processing of big data workloads. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 153–165, 2016.

[32] I. Jo, D.-H. Bae, A. S. Yoon, J.-U. Kang, S. Cho, D. D. Lee, and J. Jeong. YourSQL: a high-performance database system leveraging in-storage computing. *Proceedings of the VLDB Endowment*, 9(12):924–935, 2016.

[33] D. Katz, A. Barbalace, S. Ansary, A. Ravichandran, and B. Ravindran. Thread migration in a replicated-kernel os. In *2015 IEEE 35th International Conference on Distributed Computing Systems*, pages 278–287. IEEE, 2015.

[34] K. Keeton, D. A. Patterson, and J. M. Hellerstein. A case for intelligent disks (idisks). *Acm Sigmod Record*, 27(3):42–52, 1998.

[35] G. Koo, K. K. Matam, I. Te, H. K. G. Narra, J. Li, H.-W. Tseng, S. Swanson, and M. Annavaram. Summarizer: trading communication with computing near storage. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 219–231. IEEE, 2017.

[36] Y.-S. Lee, L. C. Quero, Y. Lee, J.-S. Kim, and S. Maeng. Accelerating external sorting via on-the-fly data merge in active ssds. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*, 2014.

[37] H. Li, M. Hao, S. Novakovic, V. Gogte, S. Govindan, D. R. Ports, I. Zhang, R. Bianchini, H. S. Gunawi, and A. Badam. Leapio: Efficient and portable virtual nvme storage on arm socs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 591–605, 2020.

[38] S. Pei, J. Yang, and Q. Yang. Registor: A platform for unstructured data processing inside ssd storage. *ACM Transactions on Storage (TOS)*, 15(1):1–24, 2019.

[39] L. C. Quero, Y.-S. Lee, and J.-S. Kim. Self-sorting ssd: Producing sorted data inside active ssds. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–7. IEEE, 2015.

[40] R. Schmid, M. Plauth, L. Wenzel, F. Eberhardt, and A. Polze. Accessible near-storage computing with fpgas. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[41] Z. Schoenborn. Board Design Guidelines for PCI Express Architecture. https://web.archive.org/web/20160327185412/http://e2e.ti.com/cfs-file/__key/communityserver-discussions-components-, 2004.

[42] S. Seshadri, M. Gahagan, S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, and S. Swanson. Willow: A user-programmable ssd. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 67–80, 2014.

[43] S. Shadley and N. Adams. What happens when compute meets storage, 2019.

[44] J. Wang, E. Lo, M. L. Yiu, J. Tong, G. Wang, and X. Liu. Cache design of ssd-based search engine architectures: An experimental study. *ACM Transactions on Information Systems (TOIS)*, 32(4):1–26, 2014.

[45] J. Wang, D. Park, Y.-S. Kee, Y. Papakonstantinou, and S. Swanson. Ssd in-storage computing for list intersection. In *Proceedings of the 12th International Workshop on Data Management on New Hardware*, pages 1–7, 2016.

[46] L. Woods, Z. István, and G. Alonso. Ibex: An intelligent storage engine with support for advanced sql offloading. *Proceedings of the VLDB Endowment*, 7(11):963–974, 2014.