Past, Present and Future of Computational Storage

Lukken, Corne; Trivedi, Animesh

2021

Past, Present and Future of Computational Storage: A Survey

CORNE LUKKEN and ANIMESH TRIVEDI, Vrije Universiteit (VU), Netherlands

We live in a data-centric world where we are heading to generate close to 200 Zettabytes of data by the year 2025 [56]. Our data processing requirements have also increased as we push to build data processing frameworks that can process large volumes of data in a short duration, a few milli- and even micro-seconds. In the prevalent computer systems designs, data is stored passively in storage devices which is brought in for processing and then the results are written out. As the volume of data explodes this constant data movement has led to a *data movement wall* which hinders further process and optimizations in data processing systems designs. One promising alternative to this architecture is to push computation to the data (instead of the other way around), and design a computational-storage device or CSD. The idea of CSD is not new and can trace its root to the pioneering work done in the 1970s and 1990s. More recently, with the emergence of non-volatile memory (NVM) storage in the mainstream computing (e.g., NAND flash and Optane), the idea has again gained a lot of traction with multiple academic and commercial prototypes being available now. In this brief survey we present a systematic analysis of work done in the area of computation storage and present future directions.

Additional Key Words and Phrases: Programmable Flash Storage, Computational Storage, Near-data Processing, In-storage Computing, User-programmable Storage, Active Disk

ACM Reference Format:
Corne Lukken and Animesh Trivedi. 2021. Past, Present and Future of Computational Storage: A Survey. *J. ACM* 0, 0, Article 000 (2021), 32 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Modern storage technologies such as flash or Intel optane [84] allow for increasingly higher bandwidth. Due to the Von Neumann architecture this results in increasing amounts of data being moved between the storage device, Central Processing Unit (CPU) and Dynamic Random Access Memory (DRAM). However, not all these technologies are scaling at the same rate. Primarily CPU and DRAM are projected to scale slower in the coming years. Other technological improvements such as those of link and interconnect bandwidth might suddenly stagnate[1]. The consequence is that excessive data movement is rapidly becoming a bottleneck for data intensive workloads [7, 44, 67, 72]. Even if the technological capabilities keep scaling to keep up with flash storage technology, unnecessarily moving data remains wasteful.

A promising solution to this problem is pushing compute to the storage layer. Today this is researched under a large variety of terms such as "Near-data processing" or "Computational Storage".

[1]The PCI-SIG group projected in 2019 that PCIe gen6 would be available in 2021 [60]. Currently a draft exists but no matching electromechanical specification making it unclear if this theoretical protocol will be feasible in practice. The combination of a closed drafting process and the specifications being hidden behind a paywall make further predictions impractical.

Authors' address: Corne Lukken, info@dantalion.nl; Animesh Trivedi, a.trivedi@vu.nl, Vrije Universiteit (VU), De Boelelaan 1105, Amsterdam, Netherlands, 1081 HV.

0004-5411/2021/0-ART000 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

However, the concept of Computational Storage (CS) is not new and has previously been explored for mainframes [12], distributed storage, databases, buses, links, interconnects and harddrives (HDD) [2, 3]. The problem in identifying CS is that until recently an exact definition was not available [83]. Throughout this work the definitions set out by the SNIA Computational Storage Architecture and Programming Model will be used [83]. This association is driving research and standardization in storage and networking with a number of standards being drafted on Computational Storage. In addition SNIA hosts several conferences.

In light of these recent definitions it is apparent that CS is ubiquitous, as will be demonstrated in a later section. However, the adoption of CS varies greatly across different types of storage technologies. Particularly in flash based Computational Storage Devices (CSx) such as Solid State Drives (SSD). Here we have seen nearly a decade of research with regards to CSx with no wide-spread market adoption. In fact, it is only recently that the first Programmable Flash Storage (PFS) Devicess have become commercially available [29, 57, 75, 76]. The underlying flash architecture is substantially different to that of convential harddrives while this providing the same block level access interface[16]. This difference is known as the semantic gap and is a potential reason for slow research progress in CS.

All currently available PFS devices either commercial or for research purposes ignore fundamental requirements such as security, multi-user tenancy, usability and data consistency [10]. Why is it that even after a decade even the latest works do not propose solutions that meet these requirements?

2 RELATED WORK

We intentionally limit this section to surveys and notice that surveys on CS are very limited. However, there is one key work from Antonio Barbalace & Jaeyoung Do [10]. Several other fields would likely benefit from CS applications such as those of big data storage and Software-Defined Storage (SDS) systems. SDS uses software abstractions to provide provisioning and data management indenpent of the underyling hardware. In these fields we see notable surveys that describe a similar problem of data increase and performance requirements [55, 80].

Even the concept of programmability is directly addressed as an issue in SDS systems. We believe the field of SDS and CS have many similarities but that SDS does not typically employ computations close to the storage elements themselves. Given the computations that are required for provisioning, policies and management these could potentially be partially offloaded to the underlying drives in the case of CS. This is a potential future opportunity in SDS to utilize CS to improve energy efficiency and prevent unnecessary data movement. Especially as many of SDS functions such as erase coding and data compression are excellent CS applications.

Similarly in big data systems we see large scale databases and distributed filesystems. In these applications we typically require fast retrieval and realtime processing. Yet the architectures employed typically involves moving the data over interconnects and even networks. Often the data retrieved is only used once or temporarily and most applications such as those of MapReduce are reductive in nature. Resulting in that these big data applications would also likely benefit from utilizing CS.

While CS is able to help advance certain fields of research there is also work being done that helps advance CS. Notable examples are the design of host-managed interfaces for SSDs such as on Open-Channel SSD (OCSSD) and more recently Zoned Namespaces (ZNS) [47][2]. However, there is no general agreement that ZNS and CS are complementary technologies and some even argue they

are contradictory [87]. In this study we will elaborate extensively on why these two technologies work excellently together.

Beyond the storage interface we see research in the programming interface as well [40]. In recent works a case is being made to use BPF for heterogeneity. BPF is typically used within the context of the Linux kernel but in essence it is an open Instruction Set Architecture (ISA). BPF bytecode can be executed in a VM, presenting a stable Application Binary Interface (ABI) that is completely decoupled from the host ISA and vendor implementation. Allowing for reusable applications that can be written in any programming langauge as long as they are compiled into BPF bytecode. This programming flexibility for users combined with implementation flexibility for vendors is ideal for CS.

3 STUDY DESIGN

In this section we describe our research goal as well as the questions that support this. Afterwards we show a selection of initial papers and describe a procedure to find related literature. Furthermore, we define the inclusion and exclusion criteria for this literature as well as any exceptions. In addition, we show an overview of the selected literature, the keywords used and a general timeline from when these works where published. To finally, offering a dedicated section on reproduceability.

3.1 Research Goal

Our study aims at creating a brief historical overview of CS given the new definitions set out by SNIA [83]. From this, a case is made to demonstrate the ubiquity of CS and all its current different applications. Afterwards we draw our attention to PFS and thoroughly investigate all different research prototypes over the past decade. Furthermore, we identify the limitations of these prototypes and describe in detail why there is still no general availability of PFS device. The main focus of this study is on the subject of PFS. It is only on this subject that we provide future predictions and suggestions.

From these goals the following survey questions are drawn:

(1) What is Computational Storage (CS)?
(2) What type of applications are there for CS?
(3) How has Programmable Flash Storage (PFS) advanced over the past decade?
(4) What are the challenges involving PFS?
(5) How will PFS evolve in the near future?

3.2 Seed Papers

A large selection of research papers is needed to be able to completely answer all the questions set out in our study design. However, an exhaustive approach is simply not possible due to time constraints. Similarly the large variety of keywords used throughout the history of CS makes a simple term based exploratory study approach impractical.

Our approach relies on identifying a few initial seed papers to create a set of keywords used for exploration. These keywords are not only extracted from the seed papers themselves but also from any citations. Subsequently the keywords are used to search across ACM, IEEE, SNIA, Usenix and Semantic Scholar. The initial selection is done based on the definition for CS as set out by SNIA [83], however, it should be noted that this definition is very broad. Filtering of this selection is

[2]This reference refers to Computational Storage as if it solely relates to black-box devices without any programmability, so called fixed function devices. In this work we use the broader definition as set out by the SNIA model [83]. additionally, the work its Computational Devices section explains concepts of SNIA model 0.5 which has changed significantly since then (0.8 at time of writing).

done using several inclusion and exclusion criteria that will be detailed later. In total 50 papers will be selected of which 25 dedicated to PFS. This manual selection procedure allows to identify the key historical contributions as well as the state of the art while taking into account any time constraints.

Our selection of initial seed papers is shown sorted by publish date in ascending order in table 1.

Table 1. Seed Papers

Paper	Related Keyword	Publish Date
Database Computers? A Step Towards Data Utilities [12]	Database Computer	xx-12-1976
Active disks: programming model, algorithms and evaluation [3][3]	Active Disks	xx-10-1998
The Necessary Death of the Block Device Interface [16]	Host-managed	xx-06-2012
Active Flash: Out-of-core data analytics on flash storage [17]	Active Flash	16-04-2012
Near-Data Processing: Insights from a MICRO-46 Workshop [7]	Near-Data Processing (NDP)	06-08-2014
Computational Storage: Where Are We Today? [10]	Computational Storage	11-01-2021
Computational Storage Architecture and Programming Model v0.8 [83]	Computational Storage	xx-06-2021

List of initial seed papers used to construct list of keywords.

Prior to identifying keywords we first define the inclusion and exclusion criteria in the next section.

3.3 Inclusion & Exclusion criteria

This section defines the inclusion and exclusion criteria used throughout this work. For a work to be considered for inclusions it must match all inclusion and none of the exclusion criteria. Any notable exceptions to this will be indicated explicitly.

- I.1 - The work brings computations closer to the (non-volatile) storage layer.
- I.2 - The proposed architecture must match the SNIA Computational Storage definition [83].
- I.3 - The proposed solution must be novel.
- E.1 - The work involves Processing-in-Memory (PIM).
- E.2 - The work involves micro-architectural or computer-architectural changes to physically bring computational elements closer to memory (on die).
- E.3 - The work involves a literature survey.
- E.4 - The work is a continuation or improvement of a previous work.

These criteria aim at extracting only the most notable works reducing data movement or offloading host computations with a limited selection. The inclusion criteria should allow to achieve this. In addition we want to be able to identify fundamental characteristics as well as limit the scope so all works have similar challenges. Similarly, the exclusion criteria are designed to achieve this.

As mentioned we allow for a limited selection of 50 papers of which 25 dedicated to PFS. That does not mean that the entire list of references will be limited to these 50 papers. Rather, only these works are used for answering our research questions. Any support references to further amplify our arguments or better illustrate our case will still be included.

Apart from this selection methodology we have one notable exception that clearly breaks our criteria. This is the work by Antonio Barbalace & Jaeyoung Do [10]. This work provides considerable contribution that strongly overlaps with our *What are the challenges involving PFS* research question. Given this overlap it would be unfair to exclude this work.

[3]Several works around utilizing embedded processors on hard disks for offloading computations have been introduced around the same time. We have chosen the work on Active Disks as a technical report on the same topic by the same authors was released 7 months prior.

With these criteria we can now perform some initial unstructured exploration using the seed papers to determine a list of relevant keywords. These keywords are described in the next section.

3.4 Exploratory Keywords

As mentioned the keywords are extracted from seed papers and their citations. The list of keywords is by no means exhaustive and aims at using the most relevant keywords while also allowing for variety. An example of such variety would be works on fixed CS which are less common than those on programmable CS. The key difference between fixed and programmable storage is that in programmable storage the program running on the device can be changed. This can be done to various *degrees of programmability* which will be described in a later section.

The keywords are shown in table 2 also showing the amount of works initially selected per keyword as well as the platform. Also shown are several compound keywords showing substitutions such as *User-Programmable (KVS / Storage)*. This means that we search for both *User-Programmable KVS* and *User-Programmable Storage*. Finally any characters commonly used to substitute spaces such as - are used in searches both as is and substituted with space.

For each keyword we limit the number of evaluated works to about 10 on each individual platform. With the exception for the *Near-data processing*, and *Computational storage* keywords. The selection of works considered for evaluation is based on the content of the abstract and references.

Several libraries also include work of other publishers. Such is the case with the ACM library. In those cases we count the work towards the publisher not the maintainer of the library. However, should the platform itself not be a publisher as with Semantic scholar Then it will still count towards Semantic Scholar. Our approach is not used to conclude anything about the distribution of CS works across platforms. Rather, it is used to prevent missing any relevant works.

In total 163 different works are selected across all platforms and keywords. The distribution of these works is shown in table 2. However, the summation of these totals does not result in 167, the reasoning behind this is explained in the following reproduceability section.

Table 2. Initial Keyword Selection Overview

Keyword	ACM	IEEE	SNIA	Usenix	Semantic Scholar	Total
Active Disks	5	2	0	1	9	17
Active Flash	5	4	0	4	2	15
Active Storage	4	5	0	3	7	19
Intelligent Disk	1	0	0	0	1	2
Near-data Processing	10	6	2	0	2	20
Computational Storage	7	0	18	1	7	33
in-storage (computing / computation)	5	10	4	0	4	23
(User-)Programmable (KVS / Storage)	2	1	0	1	2	6
Smart SSD	7	2	3	0	0	12
Storage Computing	2	1	0	1	0	4
Totals	48	31	27	10	34	150

Overview of initially selected works across keywords used in literature study.

3.4.1 Selected Literature. From these initial works the exclusions and inclusion criteria are used to create a limited set of around 50 works. The distribution across keywords and platforms for this limited selection is shown in table 3. For each keyword and platform we also show how many works apply as PFS.

Table 3. Filtered Keyword Selection Overview

Keyword	ACM	IEEE	SNIA	Usenix	Semantic Scholar	Total
Active Disks	3 (1)	1 (0)	0	0	1 (0)	5 (1)
Active Flash	0	2 (2)	0	1 (1)	0	3 (3)
Active Storage	1 (0)	0	0	1 (0)	2 (0)	4 (0)
Intelligent Disk	1 (0)	0	0	0	0	1 (0)
Near-data Processing	7 (5)	1 (0)	0	0	0	8 (5)
Computational Storage	2 (2)	0	7 (0)	0	3 (2)	12 (4)
In-storage (Computing / Computation)	4 (3)	1 (1)	2 (0)	1 (0)	2 (2)	10 (6)
(User-)programmable (KVS / Storage)	2 (2)	0	0	1 (1)	3 (0)	6 (3)
Smart SSD	2 (2)	1 (1)	1 (1)	0	0	4 (4)
Storage Computing	2 (1)	1 (1)	0	0	0	3 (2)
Totals	28 (16)	7 (5)	10 (1)	4 (2)	11 (4)	60 (28)

Overview of selected works across keywords in literature study. The number of works categorized as PFS are shown using the round brackets.

3.5 Literature Analysis

The thorough evaluation of works allows for comprehensive insight in to how particular keywords are used. We share this disambiguation here as it can greatly improve readability, especially for those less familiar with the field of CS. Together we show a visual timeline of when works where published and to which keyword they relate as well as to what degree they are incorporated in this work.

3.5.1 Active Disks. Starting with *Active Disks* we see two uses for this keyword. First is the initial use around 1998 till 2002 here we see the physical combination of a compute element and a HDD. Second, is used to indicate the revival of this paradigm with SSDs. Similarly in nature to the first use is the use of the *Intelligent Disk* keyword all be it less popular.

3.5.2 Active Flash. This paradigm of combining computational elements continues with *Active Flash* but naturally this keyword is focused on combining compute With SSDs. In addition this is also how the *Smart SSD* keyword is used. We observe the first use of the *Active Flash* keyword around 2011 well into 2013 after which it seems to have become increasingly less popular. While the use of *Smart SSD* is observed later starting around 2013.

3.5.3 Active Storage. For *Active Storage* we found works on scheduling task parallelism for FPGAs [8] as well as general distributed object storage [14]. This is at least partly due to inexperience of the reader as none of these works actually contain the keyword *Active Storage* in the title or abstract. We will address further discrepancies like these in the reproduceability section. The term is perhaps best described by Ilia Petrov, et al in their survey on *Active Storage*:

> *Active Storage refers to an architectural hardware and software paradigm, based on colocation storage and compute units. Ideally, it will allow to execute application-defined data- or compute-intensive operations in situ.* (Ilia Petrov, et al, 2018 [65])

With this definition in mind it is clear the keyword is relevant to find related works. Although we more typically see *Active Storage* used in networked, distributed, HPC and big-data applications such as with MapReduce [24].

3.5.4 Near-data Processing. *Near-data Processing* [7] is used for both storage and memory systems. Such as processors in memory channels or on 3D stacked memory dies. Additionally, this term is also used in micro-architectural designs where techniques are employed to bring the computational element, such as an Arithmetic Logic Unit (ALU), closer to (memory) storage. Nevertheless, this keyword is often used in influential CS works so all be it more labor intensive it can not be omitted.

3.5.5 Computational Storage. With regards to other keywords, *Computational Storage* (CS) is a relatively new term. During our exploration the earliest occurrence was found in 2017 with the fast majority starting from 2019. The large amounts of work found for this keyword is mainly due to the many presentations given at SNIA SDC [86] or the more recent SNIA PM+CS Summit [85]. Over 15 of the works from the initial selection on CS came from SNIA with 13, the vast majority, published in 2020 or 2021. It is unclear however, if we will see this momentum for CS extend outside of SNIA.

3.5.6 In-storage (Computing / Computation). With *In-storage (Computing / Computation)* we are likely to find many works related to PFS. We find the first works back in 2012 [30] although the use of this term in literature remains sporadic until 2016 from which point on we see significant increase in its usage [37, 62, 92].

3.5.7 (User-)programmable (KVS / Storage). The use of *(User-)programmable (KVS / Storage)* throughout our selected literature is a bit mixed. We see the expected use of end-user programmable CS systems. However, we also see works appear in these searches related to MySQl query offloading. Still, the majority of works selected from this keyword is on CS that offers some degree of user programmability.

3.5.8 Smart SSD. With *Smart SSD* we found a similar amount of supportive literature as well as literature directly related to CS. With supportive literature we mean technologies our case studies that aid the implementation of CS our show its necessity. In a later section we will go into more detail about some areas of supportive literature.

3.5.9 Storage Computing. Although very few results are found for *Storage Computing* as keyword, overall it seemed to be effective at finding CS works. The use of this keyword relates mainly to *Active Storage* as almost all works proposed an architecture in a distributed setting.

3.5.10 Visual Timeline. From these tables we derive a visual timeline that not only shows the initial works found, selected or those attributed to PFS. In addition, it shows the seed papers and surveys. This timeline is shown in figure 1.

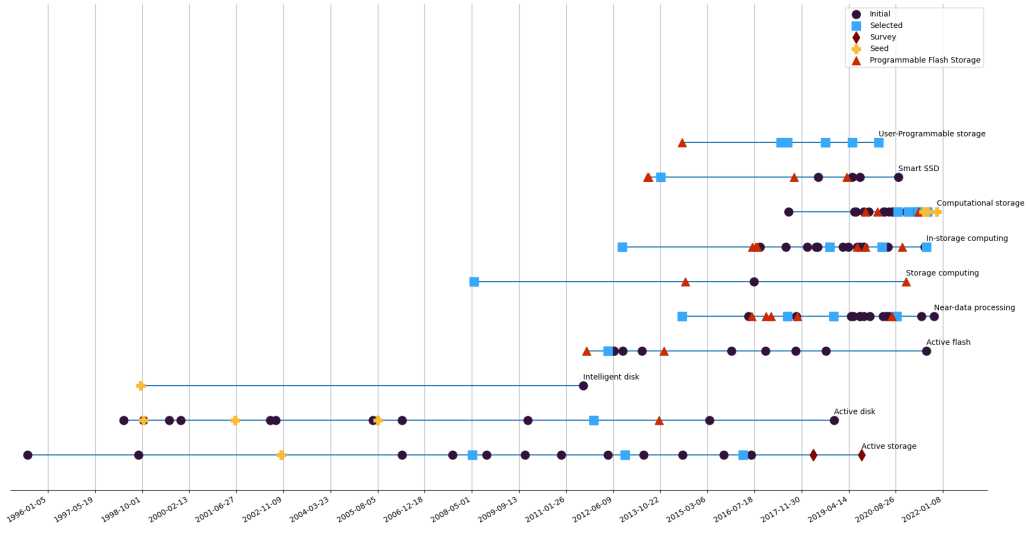

Fig. 1. Overview of literature and their respective keywords projected on a timeline.

3.6 Reproduceability

The literature selection method introduces several activities that could hinder reproduceability. In this section we describe these aspects and our way to go about them.

3.6.1 Search Order. Firstly is platform and keyword search order. Finding related literature is done by searching with keywords across several platforms. Naturally, some platforms can return the same results as others. This will affect the distribution as shown in tables 2 and 3.
Our approach searches through the platforms in order of:

(1) ACM
(2) IEEE
(3) Usenix
(4) SNIA
(5) Semantic Scholar

While the keywords are searched in the same order as they appear in the tables 2 and 3, being:

(1) Active Disks
(2) Active Flash
(3) Active Storage
(4) Intelligent Disk
(5) Near-data Processing
(6) (User-)Programmable (KVS/Storage)
(7) Smart SSD
(8) Storage Computing

3.6.2 Keyword Attribution. The methodology employed further influences how keywords are attributed to literature beyond just the search order. This is because our methodology is inherently naive. If a work is found using a given keyword it is always attributed to this keyword no matter the content of the title or abstract. In hindsight, a better approach would be to determine the matching

keyword based on the number of occurrences of keywords in the title and abstract, only resorting to attributing it to the search query if no match is found.

3.6.3 Novelty Assessment. One of the selection criteria requires the work to be novel. Naturally there are many different ways to assess novelty. Inherently, this can make the novelty assessment quite subjective. To alleviate this we define a process for novelty assessment that can easily be replicated by others. In this work novelty is assessed based on the overlap between works released prior compared to the works from our initial selection. Should a scientific contribution be released in 2013 then only works prior to this can invalidate its novelty. Overlap itself is assessed by determining the proposed architecture, its user programming interface, the underlying hardware abstraction layer (HAL) and its guarantees such as multi-user tenancy or access level control (ACL).

3.6.4 Support Files. To orchestrate the collection of literature several support files are used. These files are made available to further improve the reproduceability of this work. The include a color coded overview of the evaluated literature, to which platform it belongs and to which keyword. Other additional information includes the DOI, date and if it regards PFS. These support files can be downloaded online [54].

4 WHAT IS COMPUTATIONAL STORAGE

Computational storage is the broad concept of bringing computations closer to the storage layer. It is the process of coupling Computational Storage Functions (CSF) to storage, thereby, offloading the host or reducing data movement.

> *Computational Storage: Architectures that provide Computational Storage Functions coupled to storage, offloading host processing or reducing data movement.* (SNIA, 2021 [83])

Given this flexibility CS sees numerous different architectures that can be categorized into three predominant types as shown in figure 2. This figures uses examples showing specfic types of computational elements and storage technologies while in practice no restrictions apply here. Starting on the left we see an FPGA being used as a central interface to communicate with multiple SSDs. All communications from the host are done through the interface offered by the FPGA. This architecture is of an array type. Moving one to the right is an FPGA used to interface with a single SSD. This is known as drive type. When the FPGA is replaced with a more conventional CPU it is also known as a drive type. When the compute element and the SSD share the same bus as the host it is known as a processor type. This type is illustrated from the right half of the center.

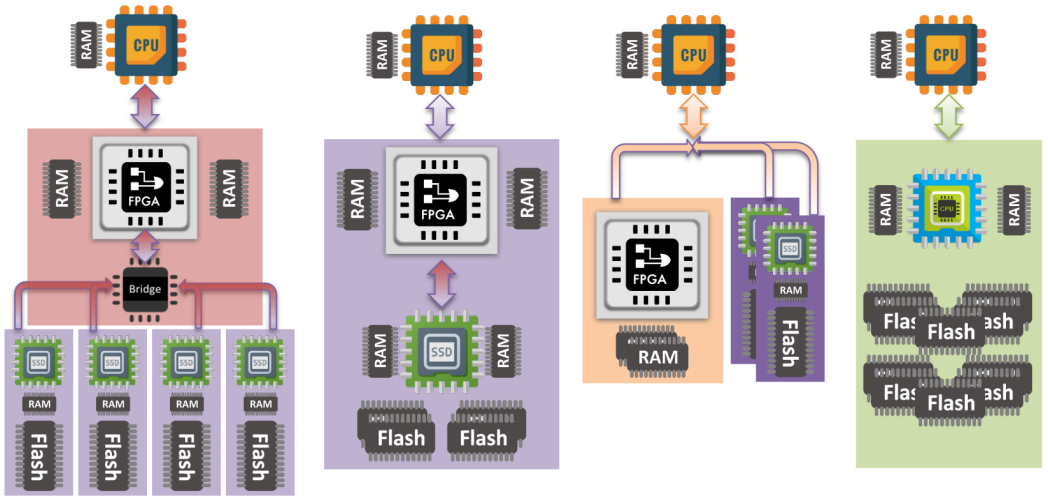

Fig. 2. Overview of different CS architectures, image from Leah Schoeb [77]

Having understood these basic type abstractions we can go into the more formal definitions as set out by SNIA. Here we see Computational Storage Processors (CSP), Computational Storage Arrays (CSA) and Computational Storage Drives (CSD). The more general term Computational Storage Device (CSx) applies to all three categories. In figure 3 we show these architectures and their individual components.

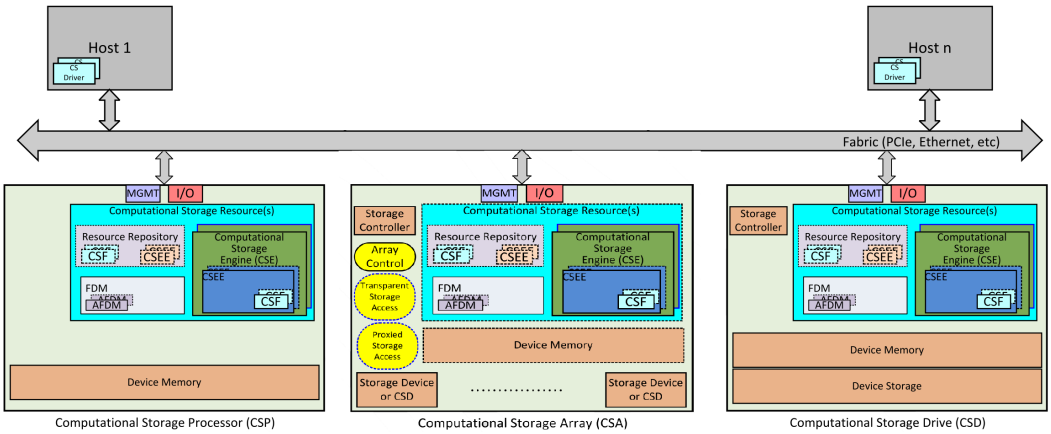

Fig. 3. Overview of different CS architectures using more formal definitions and highlighting composition.

Out of these components only a few are essential to build a CSx. These include a Computational Storage Engine (CSE), Computational Storage Execution Environment (CSEE), Computational Storage Function (CSF), Computational Storage Resource (CSR), Function Data Memory (FDM) and Allocated Function Data Memory (AFDM).

The overall idea is that a CSR has one or more CSEs that contain one or more CSEEs that subsequently contain one or more CSFs. The CSR can allocate FDM to create AFDM used by a CSF during its execution inside a CSEE.

The CSR is the overall part of the device, accessible over an interface, that contains all the components to support CS. Inside the CSE is the computational element, this can be a Field Programmable Gate Array (FPGA), CPU or Digital Signal Processor (DSP). To run CSFs on these CSEs we need an CSEE. These are essentially support libraries, sandboxing or virtualization layers.

Using this basic architecture would create a fixed CSx that has a predefined set of CSFs that can be executed and that can not be changed by the end user. Contrarily, should the CSx contain a resource repository and provide a method to upload new CSFs or CSEEs to this repository than it is a programmable CSx.

This line can become blurry as some devices allow to program a chain or Directed Acyclic Graph (DAG) of unchangeable preprogrammed functions that can arguably be seen as some form of programmability. Using this method it might even be possible to support dataflow programming models.

Lastly, CS should not be confused with Programming-in-Memory (PIM) or smart Network Interface Controllers (NIC). Particularly PIM might be misinterpreted as CS since we have seen many recent works introduce PIM to non-volatile RAM technologies. Similarly, although smart NICs should not be considered CS we see many distributed CS architectures emerge that use a networking interface and should be considered CS. The key difference here is that a smart NIC does not contain any storage technology on the device itself, contrarily to CSxs with a networking interface.

In short CS is the process of coupling any computation to storage thereby offloading host processing or reducing data movement.

5 HISTORY OF COMPUTATIONAL STORAGE

Although the term *Computational Storage* itself is relatively new, the history of CS can be traced back to database computers developed in the 1970s and 1980s [12]. These machines typically offloaded specific database operations although this was often complicated by the infancy of relational databases themselves as well as lack of associative memory.

Overall the entire history of CS is much richer than will be shown in this section. This work has keeps this brief, focusing on the most prominent contributions of the time. This is done to be able to achieve a more in depth analysis on the past decade of CS works. An overview of the works covered in this section is shown in table 4.

Table 4. Historical Computational Storage

	Disc Architecture	SNIA Architecture	Release Date
CASSM [88]	PPT	CSD	1975
RARES [49]	PPT	CSD	1976
RAP [61]	PPT	CSD	1977
DIRECT [25]	Per device	CSP	1979
Active Disks, Riedel et al [70]	Per device	CSD	1998
Active Disks, Acharya et al [3]	Per device	CSD	1998
Intelligent Disk [39]	Per device	CSD	1998

Overview of historical CS works with their architecture and release dates.

5.1 Database Computers

Several database computers employed an architecture to solve the issues of their requirements directly such as *Content-Addressable Segment Sequential Memory* (CASSM) [88]. This database computer used a head-per-track disc and a Processor Per Track (PPT) architecture. In addition Processor Per Head (PPH) and Processor Per Disc (PPD) architectures existed. The difference between a PPT and PPH architecture in a head-per-track disc is that in PPT the heads are fixed and in PPH the head is moveable. There are three fundamental types of database compute architectures as shown in figure 4.

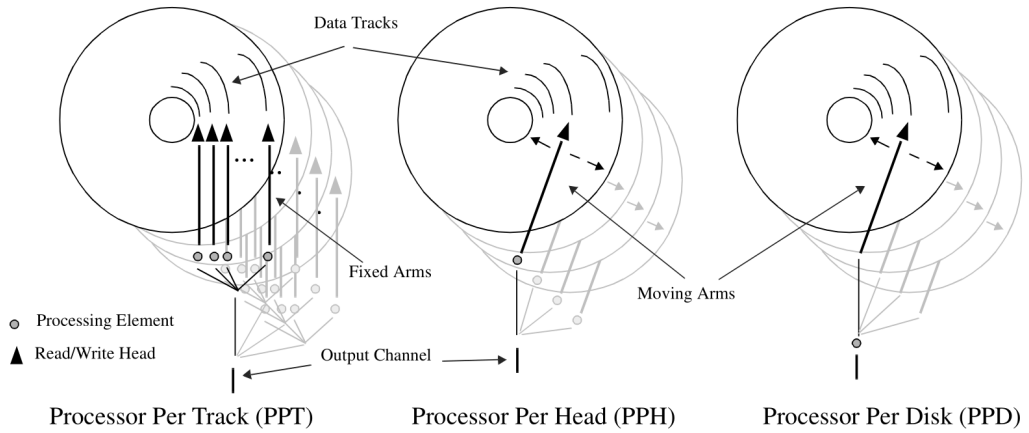

Fig. 4. The three fundamental types of database computers employing different amounts of processing per harddrive component, image from [69].

This terminology of PPT, PPH or PPD is sometimes also referred to as Logic Per Track (LPT), Logic Per Head (LPH) or Logic Per Disc (LPD). These different types of database computer architectures best relate to the CSD architecture from SNIA. The CSA architecture does not really apply as it would be hard to argue that individual tracks or heads on a disc are entire storage devices by themselves.

After CASSM we saw the introduction of *Relational Associative Processor* (RAP) [61]. This allowed for more complex searches through metadata mark bits. In addition, this metadata was no longer stored in RAM as was the case with CASSM but alongside the data itself. Similar to CASSM, RAP used an PPT architecture.

Another machine was the *Rotating Associative Relational Store* (RARES) [49]. RARES used a PPT architecture as it used a head-per-track disc with fixed heads. RARES is different in the sense that it separates the processing into back-end and front-end processing as well as storing data orthogonality to the storage layout. That is to say the data is spread across tracks rather than being written linearly on a track.

Later database computers often did away with the PPT or PPH architectures such as with DIRECT [25]. Here a crossbar switch was used to allow any processor to communicate with any storage unit. In addition it used a Multiple Instruction, Multiple Data (MIMD) instruction set as opposed to Single Instruction, Multiple Data (SIMD) instruction set. This is one of the few database computers to not use a CSD like architecture but instead use a CSP architecture.

Even later machines did away entirely with custom built processors and started using general purpose processors. Some commercially available systems, mostly using general purpose processors are listed below.

- Content-Addressable File Store (CAFS and SCAFS)
- Microsoft TerraServer
- Digital TPC-C
- Teradata optical storage processor

The complexity that arouse from using custom processors with custom microcode often made programming database computers difficult. Their performance also typically did not outweigh their large size and cost. Even though the second generation of database computers used commodity hardware. This still led to most database computers not being attractive for practical applications outside of research areas. Especially once networking, memory and processing power become more powerful and cheaper, the need for these high cost purpose built machines decreased. A noteworthy exception is the Teradata optical storage processor that was still being sold at the end of the 1990s [68].

5.2 Active Disks

From the research of database computers spawned the field of *Active Disks* in the late 1990s. These similarly, introduced processing capabilities on the HDD itself but using a PPD or processor per device architecture. Like database computers these are also CSD architectures. Different from database computers in the sense that a single Active Disk could be installed in a consumer computer or datacentre server. The necessity of a large dedicated rack for a special purpose machine was thereby removed. This also significantly reduced the cost as the additional processing would only be around 10% of the total cost of a HDD at that time. Although it did push the entire responsibility of interfacing and programming these *Active Disks* to the host operating system.

During the start of Active Disks as a research field we saw three primary contributions. Two named after Active Disks [3, 70] and one named Intelligent Disk [39]. All three they offered different architectures and degrees of programmability[4] as shown in figure 5.

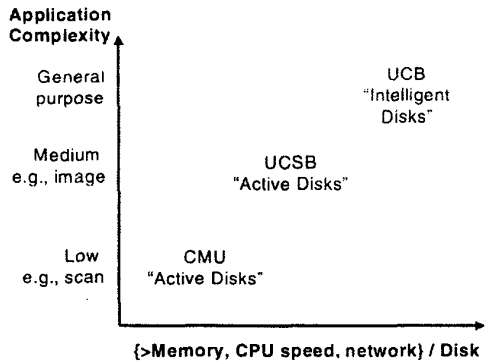

Fig. 5. Different types of Active Disks technologies and their respective degrees of programmability, figure by Keeton et al [39].

In increasing degrees of programmability we first start with Active Disks by Riedel et al (CMU) [70]. These used a PPD architecture and used scan like operations to reduce data transfers. While

[4]We will address *degree of programmability* with much more detail in sections on PFS.

the work by Acharya et al (UCSB) [3]. used a processor per device architecture. A key difference is that a single device can contain multiple discs so PPD potentially has more processors than a processor per device architecture. Their work is also more focused on programming models, particularly streaming. Their methodology expressed operations in so-called disklets.

Finally, Intelligent Disk [39] used an architecture similar to that of DIRECT [25] using a crossbar network of fully connected HDDs. The key difference here is that each HDD has its own dedicated processor and memory while on DIRECT these where separated. While the programmability in intelligent disk is flexible it provides no programming model itself beyond a few basic recommendations. In addition the work does not provide any experimental setup or evaluations unlike the two other works on active disks.

Beyond this initial surge of new interest in CS research related to active disks, the field soon stagnated. There was some work mainly on applications in Redundant Array of Independent Disks (RAID) or Networked Attached Storage (NAS) deployments but those are considered outside the scope of this work.

5.2.1 Active Disks Lack of Success. A number of reasons might have contributed to the lack of success for active disks. One prominent one is the cumbersome programming and development methodology. Especially in a time where OpenMP (2000) and OpenMPI (2004) started to emerge. These libraries significantly reduced the development effort required for multi-threaded and distributed processing. Although it took until around 2011 until we saw support for OpenMPI in HPC batch processing systems like SLURM[5]. The availability of multi-threaded or distributed processing is a direct competition for the case of CS. Since these additional computational capabilities can resolve computational bottlenecks just active disks could.

These initial development methodologies further complicated their use as they poorly addressed multi-user tenancy, security and safety guarantees such as concurrent access and modification.

Beyond the cumbersome software development we saw the cost of HDDs drop over the course of the early 2000s, their capacity increase rapidly while there number of I/O operations per second (IOPS) stagnated. Meanwhile the performance of desktop and server grade processors increased, particularly in clock speed going from 600 MHz in 2000 to 2.8 GHz in 2010. This meant processors could serve much more interrupt requests at any given time as the amount of instructions it takes to serve an interrupt service routine has remained relatively the same. The stagnated IOPS for HDDs combined with faster interrupt processing meant that it was impossible for a modern CPU to be overloaded by one or even a large amount of harddrives. Deployments of 64 or 128 HDDs in a single server where not uncommon around that time.

The peripheral interconnects also saw advancements such as with the introduction of SATA as opposed to IDE. As well as the introduction of AHCI. This simplified the communication with HDDs from the point of the host processor as well as offering significantly higher bandwidth.

6 THE UBIQUITY OF COMPUTATIONAL STORAGE

As seen with the history of CS, the need to reduce data movement and thus move computations to storage has been prevalent for quite some time. In this section we will showcase how many different fields are already actively using this technique to advance their field. Although often not directly attributed CS, it is clear that in these fields we see architectures similar to those of CS. Even more so, should we finally see readily available PFS[6], than it is extremely likely that these fields would start using them as well.

[5]For example, HTCondor still does not support OpenMPI and perhaps never will.

Before describing all the different fields we CS emerge we briefly look at distributed storage architectures.

6.1 Layered Distributed Storage Architectures

One prominent application for CS is in distributed filesystems. These filesystems are often comprised of several fundamental components. Typically explicitly separated into subcomponents but sometimes architectured as a monolith. The typical subcomponents of a distributed filesystem include a key-value store or an object store. Sometimes an object store is a key-value store but almost never the other way around. This object store is than used to support the distributed filesystem. Other components such MetaData Servers (MDS) exist in some architectures although this is sometimes stored alongside the objects in the object store. Lastly, When an entire storage device is claimed for use as an object store it is referred to as an object-based storage device (OSD). In figure 6 an example of a distributed filesystem architecture is shown, Lustre in particular.

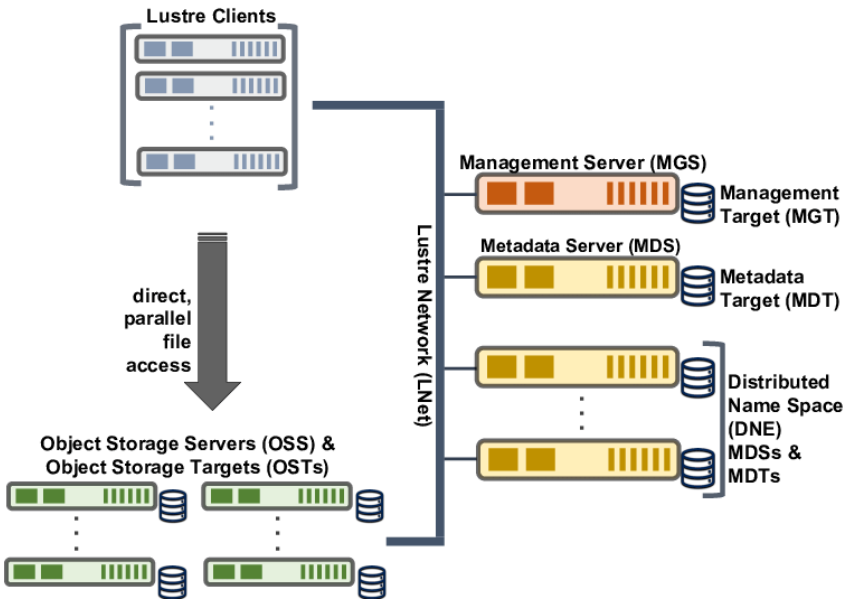

Fig. 6. Overall architecture of Lustre parallel distributed filesystem, figure by Arnab et al [63].

Other configurations are also possible such as building a local filesystem on a distributed object store. Historically the key-value and or object store where written on top of a traditional filesystem but strong evidence suggest that this can severely degrade performance [4]. Lastly, special purpose HDDs exist that run object stores with complimentary API, an example is Seagate their Kinetic HDD [78].

CS is applied in all levels of distributed filesystems to various degrees. The understanding of these basic levels is thus essential in identifying how key CS works are being applied here. In the field of *Active Storage* we often look at the object store or the filesystem as a whole. While sometimes literature on active storage solely looks at the object store. The term active storage is typically not used in the context of key-value stores.

[6]With a stable interface and architecture that is not prohibitively expensive or difficult to use..

6.2 Active Storage

Research in the field of *Active Storage* has allowed object stores and distributed filesystems to better leverage compute capabilities of storage nodes. Starting with the introduction of scriptable Remote Procedure Calls (RPC) in 2002 [82]. sRPC allows to offload computations by transmitting tcl scripts over the network subsequently executed by the receiver. Similarly, Splinter allows to export Rust scripts over the network that can be remotely invoked while also using RPC [45]. Another approach decomposes the computation into predefined steps called *functors* [95], effectively allowing for programmability with a fixed function dataflow model. An intermediate approach between these two allows tying user defined functions to specific objects [98].

Beyond entirely academic applications we also see active storage being applied to existing distributed filesystems [66]. A key benefit of this approach is that implementing important filesystem properties such as multi-user tenancy, concurrent access and modifications, crash recovery and self-healing might become substantially easier. This is because typically, distributed filesystems already come with the majority of these features.

Not every application might be fit for active storage, that is why Chen et al [19] proposed a dynamic active storage system on top of Lustre. In this work they reduced the overall resource contention and improved data dependency management as well as allowing regular distributed filesystem access. We show this high level architecture in figure 7.

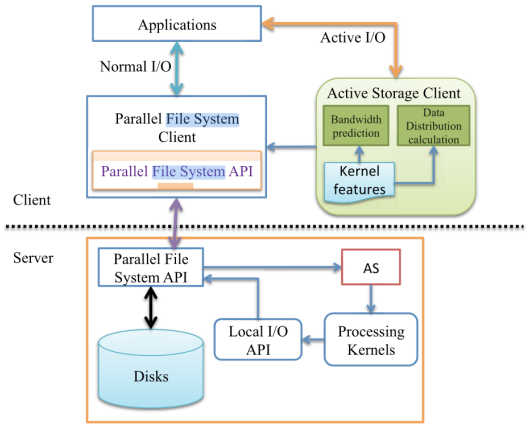

Fig. 7. Active storage architecture supporting both regular access as well as offloaded, figure by Chen et al [19].

6.3 Consumer Applications

Outside of research the Hadoop distributed filesystem, HDFS, actively ensures computations are being executed close to data. While RADOS, the object store used by CephFS utilizes intelligence functions on storage nodes [94]. Similarly, the CephFS Bluestore filesystem supports transparent compressions for which the computations are done on the local storage node [4].

On the protocol side, NVMe has seen the addition of a copy command. Allowing to copy data across SSDs without moving it to the host first [59]. Similarly, the use and applications for peer to peer dma over PCIe are increasing as well [5, 73], particularly due to NVMe-oF and RDMA. With current trends of an increasing amount of special purpose processors, perhaps it has become possible for an heterogeneous computing architecture of autonomous special purpose processors to emerge [58].

Clearly this mix of storage and computation is not to the same degree as in active storage research. However, it might only be a matter of time before we see more prominent CS applications appear across different fields.

A prime example of this new CS frontier is found in the I/O controller of the Playstation 5. Here the use of transparent decompression, supporting zlib or oodle kraken formats, can achieve between 8 and 9 GB/s transfer speeds on a 5.5 GB/s storage device[93].

7 A CASE FOR PROGRAMMABLE FLASH STORAGE

With the Von Neumann architecture we see an increasing amount of data movement that is quickly becoming the bottleneck in many data driven applications [44]. Typically many of these data movements are redundant wasting both time and power, reducing throughput and energy efficiency.

Meanwhile, due to stagnated generational improvements in CPUs [72], we are entering an age of specialization. current generations of processors are being equipped with dedicated machine learning or augmented reality accelerators. Key examples are the neural engines inside Apples A14 and M1 processors. Researchers predict this trend of special purpose hardware is only going to increase [58].

These specialized accelerators can benefit from recent developments in interconnects such CCIX [21], GenZ [22], CXL2 [50] and OpenCAPI [23]. Allowing for simpler or even cache coherent communication in a heterogeneous architecture.

Similarly, we see the development of emerging open Instruction Set Architectures (ISA) such as RISC-V [32] and OpenPower [31]. The key to open standards like these is the ability for anyone to develop, test and iterate prototypes without licensing costs. This greatly accelerates progress in specific fields.

Even more so, we see progress in the field of open FPGA toolchains and parameterized soft cores. This allows anyone to research custom CPU architectures that might better fit these specialized applications.

Clearly, Programmable Flash Storage (PFS), user-programmable computational capabilities placed right on the SSD itself, is an important next step in the heterogeneous architecture of the future.

8 A DECADE OF PROGRAMMABLE FLASH STORAGE

It has been near a decade of research in PFS [1] and yet we see no widespread adoption. Recently Antonio Barbalace et al [10] described several of the reasons that are preventing this adoption.

In this section we go over the past decade of literature and identify several key aspects of each work. These pieces of literature are further categorized into fixed function and programmable flash storage architectures. Moreover, We will define each of these identified aspects being *Degree of programmability*, *Programming model*, *Interface* and *Computational Storage Execution Environment (CSEE)*. First, we define some key software elements to integrate PFS architectures.

8.1 Components of a Successful PFS Software Architecture

A key to a successful PFS architecture is the complete design of a full vertical integration. This is to say that implementing PFS requires making changes at all layers of the software stack. Ranging from the user facing API on the host down to hardware abstraction layers of the CSx we arrive at the following components as shown in figure 8.

First are the user facing APIs, supporting different programming models. These define how a user has to write programs in order to interact with one or multiple CSxs. A specific CSx could support one or more of these APIs with each one or more programming models. Second is the one or multiple layers of hardware abstraction (HAL), these typically interact with the kernel as

well as often being vendor specific. From the HAL the communication is managed over buses and interconnects dictated by their communication protocols. Typical examples for this third layer include PCI-e or CoreLink. Like the familiar TCP/IP networking stack this third layer can contain one or more additional specialized communication protocols, a typical example would be NVMe. From here we arrive at the device with its own HAL layer. With the final sixth layer being the ABI that user written programs must conform to.

It is the responsibility of both the host API and HAL layers to ensure conformity to this ABI. While the device HAL verifies this. Similarly, it is the host API that is responsible for converting the user written program, using their programming model, into the ABI of the device.

This architecture is substantially different than the overview presented by SNIA in figure 3. This is because we are only looking at the essential components for full vertical integration required for a programmable architecture. This architecture should be seen as complementary to that of SNIA with further subdivisions being logical. However, we have intentionally kept this figure relatively simply to more easily explain the transnational interactions between different layers.

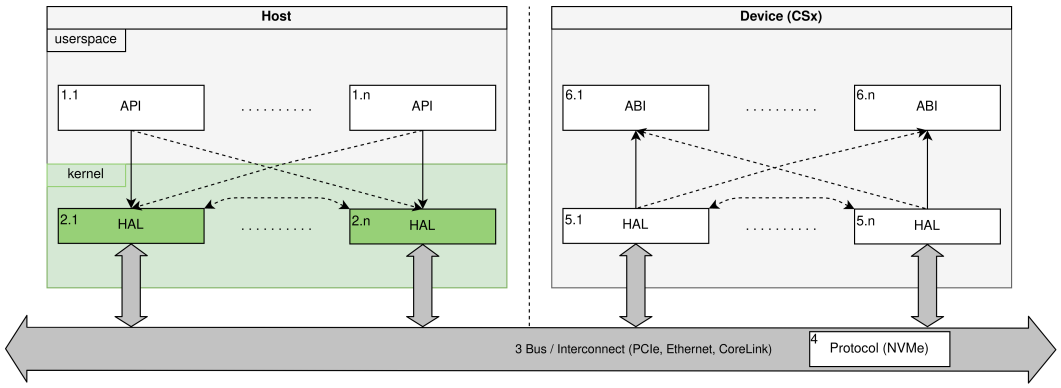

Fig. 8. Overview of basic PFS software architecture allowing for full vertical integration.

Typically the support for the third layer is largely performed in hardware with the remainder carried out by the host operating system. Similarly, large parts of the fourth layer are typically implemented by host operating systems. Sometimes operating systems expose special drivers that allow to implement the fourth or even the third layer in userspace. Examples of these drivers would be uio_pci_generic [52] and io_uring [51] in Linux.

8.1.1 Dynamic Device Discovery. One key element is missing from our previous architecture however. This is because tying many APIs to many HALs is cumbersome, error prone and discourages vendor adoption, instead we propose a dynamic architecture utilizing *Installable Client Drivers (ICD)*. These ICDs are drivers discovered and loaded at runtime, the interface is offered by the user facing API while they are implemented by the vendor. The vendor is allowed to utilize any HAL or additional components. Supporting multiple APIs or multiple devices with the same HAL is also allowed. The only restriction is that the ICD must conform to the interface of the loader which is dictated by the API.

This approach is very similar to those of advanced computer graphics APIs such as OpenGL and Vulkan. We show our mechanism for loading applications in figure 9.

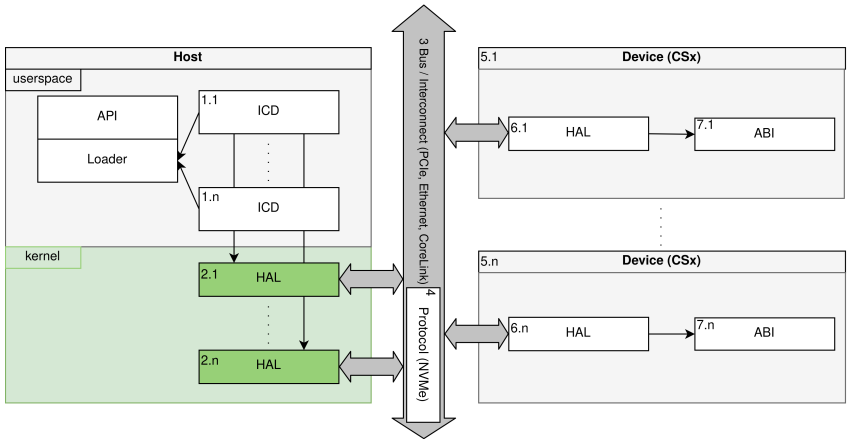

Fig. 9. Overview of PFS loader architecture

8.1.2 We Present OpenCSL. This overall presented system is called OpenCSL and will be released in a future work. It will support all three major programming models as well as allowing for agnostic vendor implementations. Moreover, the user written programs will be portable across different devices of different vendors. Finally, OpenCSL will address common problems of these systems such as multi-user tenancy and security.

8.2 Aspects of Programmable Flash Storage

When identifying different aspects of PFS numerous software layers emerge. From the perspective of the end user we identify each one using a top down approach. First, is the end user API that is used to program or interface with the device. This could be a server using HTTP/Rest or RPC. In the next tables we categorize this as the *programming model*. Next is the *interface*, this is the layer between the host and device often encapsulating additional interfaces. Typical examples are SATA or PCIe with NVMe as a typically encapsulated protocol.

Similar encapsulations are possible for the programming model where RPC calls could encapsulate SQL queries, for example.

Beyond this interface we arrive at the device and its *Computational Storage Execution Environment (CSEE)*. This is the environment in which the commands and programming of the end user are executed. These range in increasing levels of software abstraction. From the perspective of the CSEE we are only looking at the top level of the stack For instance, running a container with CSFs often requires an underlying operating system but will still be classified as *container*.

The last layer is the *degree of programmability* which we will describe in the next section. Before this description we identify each of the possible categories for each layer starting with the programming model.

In no particular order these are *Dataflow (MapReduce, DAG)*, *Client / Server (RPC, HTTP)*, *Shared memory* and *Declarative (Regex, MySQL)*. Although simple categories, the correct attribution of these categories can become quite complicated due to nuances. For instance, if an end user needs to link to a new shared library to call methods with MySQL queries as argument, what should the programming model be? In this work we argue *Shared memory* as the library is introduced in the PFS work itself. If the library is part of preexisting software, SQLite for example, it is categorized as *Declarative*. This is because only the declarative queries are interacting with the CSx. In these cases

the existing libraries are simply used as transparent interfaces to simply integration for the end user.

We believe that the interfaces will be self explanatory enough. Meaning that they won't need specific categories or further clarification. However, CSEEs do and we identify the following categories:

(1) Bistream; bitstream programmed directly unto FPGA.
(2) Embedded; single program, single memory space, no OS just 'real mode'.
(3) Accelerators; OpenCL, Vulkan.
(4) Real-Time operating system;
(5) Operating System;
(6) Container;
(7) Virtual Machine;

We believe most of these categories are self-explanatory. However, the naunce between embedded platforms and accelerators has to be properly explained. While embedded platforms might deal with simple static memory allocations we expect accelerators to support implicit and explicit host-device memory transfers. In addition we expect to have some control over device memory allocation as well as requiring a scheduler to be present.

Finally, before describing the *degrees of programmability* we will elaborate on the difference between this aspect and the *programming model*. These might seem overlapping but they are used to differentiate between the perspective of the device and end user. The programming model refers to the paradigm employed by the user in order to interact with the device. While programmability refers to how the device is reprogrammed internally. Reprogramming the CSx is hereby decoupled from the programming performed by the end user.

8.2.1 Different Degrees of Programmability. In terms of abstraction, systems are often thought of the be either programmable or not. However, distinction between individual PFS works would be lacking using this approach. When more elaborately analyzing PFS different degrees of programmability emerge. These degrees of programmability can be ordered from least programmable to most programmable as listed below.

(1) Transparent operations, (de)compression (Playstation 5 I/O Controller)
(2) Fixed functions, unchangeable, workload specific [89]
(3) Fixed function dataflow programming [95]
(4) Query offloading, SQL, NoSql, Regex [37]
(5) Event driven (hooks) user-programmable functions [9]
(6) Arbitrary code execution, VHDL, eBPF, TCL [43, 82]

Transparent operations include those such as compression and encryption, this degree of programmability belongs to fixed function devices.

From here we arrive at fixed functions which are unchangeable and often workload specific. Typical examples of fixed functions would be database operations [97] or Key-value operations [36, 99]. These could support offloading groupby and aggregate operations or put, get and delete, respectively.

However, it is not impossible to achieve programmability with fixed functions. Using a dataflow programming model in combination with small functions called *functors* general programmability can be achieved. Potentially, this programmability could even be turing complete.

Even more programmable are the systems using query languages to perform query offloading. This can be done using both structured and unstructured query languages such as MySQL and NoSQL.

The next step in programmability is the first to support executing actual user written code. These event driven systems have programmable hooks. When tied these hooks run user written code upon activation of an event. Such events could be read or write I/O operations on the SSD. We typically see this degree of programmability in dataflow or stream based programming paradigms.

The final degree of programmability is arbitrary code execution, turing complete programs that can be scheduled and ran on a device at will of the user.

Using these degrees of programmability we can now formally distinquish between fixed function and programmable flash storage.

8.3 Evolution of Flash Based Computational Storage

Before distinguishing between fixed and programmable flash storage, firstly, we identify all CS works utilizing flash from over the past decade. We present this overview ordered by publication date. In addition, we show the publication research field and the state of implementation of the work.

We categorize implementation state into four categories being *proposal*, *simulation*, *prototype* and *deployment*. A work qualifies for deployment if it is a readily available commercially or has known real world deployments in commercial systems. Mixes between simulation and prototype are appropriately attributed based on the distribution between those two. The overview is shown in table 5.

Table 5. Overview of Flash based CSx works ordered by date

Name	Publication Date	Research Field	Implementation State
Active SSD [1]	09-02-2011	HPC	Prototype
Active Flash [17, 89]	16-04-2012	HPC	Simulation
Smart SSD [28, 38]	06-05-2013	HPC	Prototype
Intelligent SSD [6, 20]	10-06-2013	HPC	Simulation
Ibex [97]	15-07-2014	Databases	Prototype
Willow [79]	06-10-2014	Operating Systems (OS)	Prototype
Biscuit [33]	18-06-2016	Computer Architecture (CA)	Prototype
Hadoop ISC* [62]	28-07-2016	HPC	Prototype
YourSQL [37]	15-08-2016	Databases	Prototype
Caribou [36]	01-08-2017	Databases	Prototype
Summarizer [41]	14-10-2017	Computer Architecture (CA)	Prototype
NDP RE2 regex* [13]	11-06-2018	Databases	Prototype
Registor [64]	26-03-2019	Storage	Prototype
Cognitive SSD [26]	10-07-2019	Machine Learning	Prototype
INSIDER [71]	15-07-2019	Storage	Prototype
Catalina [90]	10-08-2019	HPC	Prototype
THRIFTY [34]	11-02-2020	Computer Aided Design (CAD)	Simulation
POLARDB [18]	14-02-2020	Storage	Deployment
LeapIO [48]	09-03-2020	CA / OS	Simulation
CSD 2000 [91]	15-09-2020	Storage	Prototype
NGD newport [27]	12-10-2020	Storage	Deployment
blockNDP [9]	07-12-2020	Middleware	Prototype
QEMU CSD* [96]	26-04-2021	Storage	Simulation

Overview of selected flash works, their field and implementation state.

Since early 2020 CS has entered a transnational stage moving from prototypes to simulations. Here we it seems that we realize that prototypes are not useful if fundamental problems are not systematically solved using simulations. Excellent examples are flexible, reconfigurable platforms that allow for fast iterations such as QEMU CSD [96].

We also see a shift in research fields. Here earlier works had to be disguised under HPC while later works are published under storage. This is clear evidence that CS is now being taken seriously as a part of storage research.

8.3.1 Fixed Function. As previously mentioned there is less work being done on fixed function CS as evidenced by table 6. Still both of the entries shown are important to the development of CS. The work on Active Flash [17, 89] is on of the first to revive CS after research done on Active Disks.

In addition, Caribou [36] revives an interesting concept of providing an alternative interface to the block layer. With advanced distributed filesystems often using underlying object stores or key-value stores this might significantly simplify integration of CSxs. This is further evidenced by the use of an Ethernet based interface as opposed to SATA or PCIe.

Finally, there is LeapIO that uses a layered offloaded block interface. It uses local connectivity for offloading or Ethernet with RMDA if the data resides on another host. The CSx is responsible for accessing this other node over Ethernet as to not load the host. The host in the evaluation is a guest VM based on QEMU. More practical applications would be an actual X86 host with PCIe connected accelerator card.

Table 6. Fixed function computational storage overview

Name	Programmability	Programming Model	Interface	CSEE
Active Flash [17, 89]	Fixed functions	N.A.	SATA (OpenSSD)	Embedded
Caribou [36]	Fixed functions (key-value store)	Client / Server (RPC)	Ethernet	Bitstream
LeapIO [48]	Fixed functions	Transparent	Ethernet (RDMA)	Embedded
CSD 2000 [91]	Fixed functions (compression)	Transparent	PCIe (NVMe)	Bitstream

Overview of identified fixed function CS works across selected literature.

8.3.2 Programmable Flash Storage. When looking at PFS research a much larger variety of works is identified. These are shown in table 7. Overall we see early works using SATA interfaces and Embedded CSEEs while later works mostly use PCIe (NVMe) and bitstreams, respectively. Another noteworthy observations is a steady decline in the dataflow programming model. We suspect the difficulties from attempting to offload the reduce stage of MapReduce could potentially be a cause for this decline. In terms of degree of programmability the distribution is more or less constant with the majority of works being event driven. Second most common is arbitrary code execution and least is query offloading.

Table 7. Programmable flash storage overview

Name	Programmability	Programming Model	Interface	CSEE
Active SSD [1]	Event driven	Dataflow (streams)	PCIe	Operating system (Custom)
Smart SSD [38]	Event driven	Dataflow (MapReduce)	SATA	Embedded
Smart SSD [28]	Event driven	Shared memory	SATA	Embedded
Intelligent SSD [6, 20]	Arbitrary code execution[7]	Shared memory[7]	N.A.	Operating system (Linux)[7]
Ibex [97]	Query offloading (MySQL)	Declarative	SATA	Bitstream
Willow [79]	Arbitrary code execution	Client / Server (RPC)	PCIe (NVMe)	Operating system (Custom)
Biscuit [33]	Event driven	Dataflow	PCIe	Embedded
Hadoop ISC* [62]	Event driven	Dataflow (MapReduce)	SAS	Embedded
YourSQL [37]	Query offloading (MySQL)	Declarative	PCIe (NVMe)	Bitsream[8]
Summarizer [41]	Event driven	Shared memory	PCIe (NVMe)	Embedded
NDP RE2 regex* [13]	Query offloading (Regex)	N.A.	N.A.	Embedded
Registor [64]	Query offloading (Regex)	Shared memory	PCIe (NVMe)	Bitsream
Cognitive SSD [26]	Arbitrary code execution	Shared memory	PCIe (NVMe, OpenSSD)	Accelerators (Custom)
INSIDER [71]	Event driven	Shared memory (VFS)	PCIe	Bitstream
Catalina [90]	Arbitrary code execution	Client / Server (MPI)	PCIe (NVMe)	Operating system (Linux)
THRIFTY [34]	Event driven[9]	Shared memory (VFS)[9]	PCIe[9]	Bitstream[9]
POLARDB [18]	Query offloading (POLARDB)	Declarative	PCIe	Bitstream
NGD newport [27]	Arbitrary code execution	Client / Server	PCIe (NVMe)	Operating system (Linux)
blockNDP [9]	Event driven	Dataflow (streams)	PCIe (NVMe, OpenSSD)	Virtual Machine (QEMU)
QEMU CSD* [96]	Arbitrary code execution	Shared memory	PCIe (NVMe)	N.A. (Simulated)

Overview of PFS works and various aspects as previously detailed.

Other more specific observations on individual works include those on the use of the MapReduce framework. None of the implementations fully offload each stage. Typically only the mapping stage is offloaded as it has no cross data dependencies. This is unfortunate as the reduce stage would offer the highest reduction in data transferred. Interestingly this is contrarily to the claims of Hadoop ISC [62] that argues most data reduction happens in the *mapping* stage.

In terms of query offloading the work on POLARDB [18] is interesting as it sees actual deployment in Alibabas private cloud infrastructure. This is the only PFS deployment we see across all selected works together with NGD newport [27]. The newport system is a Flash based CSD that supports a wide range of programming models by offering SSH access to the device alongside some client / server based APIs. However, although very flexible we do not think this architecture will lead to widespread adoption as we will explain in a follow up section.

Other interesting architectures include that of Catalina [90] which allows programming using the Messaging Passing Interface (MPI). Similarly, we do not expect widespread adoptions as this approach highly complicates data placement and access.

Out of all these works only tried to overcome the traditional block interface by utilizing OpenSSD. This new storage interface directly exposes the underlying flash architecture significantly reducing the semantic gap between host and device. When done the Flash Translation Layer (FTL) becomes an integrated part of the host as opposed to the device. We call these type of interfaces host-managed. A successor to OpenSSD will be addressed in a following section.

While all these works fail to address some fundamental issues which hinder adoption at least the approach of QEMU CSD [96] allows for rapid prototyping using its simulation framework. Unfortunately, the source code of this simulation platform is not made available. This makes reproduction of their results impractical as well making this rapid prototyping platform inaccessible to others.

[7]Simulations performed by porting workloads unto ARM based processor. No actual hardware on SSDs is used.
[8]The work uses special FCPs with hardware based filtering functions. We assume these must be implemented using FPGAs although the work does not specify.
[9]Build on top of the INSIDER software stack.

Clearly we have seen advancements in flash based CS over the last decade. From early basic prototypes with fixed workload specific functions tied to the FTL [17, 89]. To know commercial use of transparent decompression in the Playstation 5 its I/O controller.

In PFS we also see progression as many prototypes now have led to at least two cases of commercial use. We also see the initial developments for an accelerator like CSEE similar to Vulkan or OpenCL in CognitiveSSD [26]. We believe this type of architecture has a high change of achieving widespread adoption. In the next section we will address key challenges that remain for future developments.

8.4 Complexity and Challenges

Several key-challenges remain throughout the past decade of flash based CS. We see two fundamental works identifying these [10, 11]. In addition we provide some of our own. Unfortunately, it is not possible to go in to depth identifying how each work of the previous section fairs regarding each challenge. This in depth analysis is left as future work.

Firstly, the work by Barbalace et al [11] identify some of the challenges that require explicit support. These are *locality*, *protection*, *scheduling*, *programmability* and *low-latency*. The work does not address if this support is to be implemented by the host operating system or the device apart from scheduling. Here it argues that scheduling should be implemented by the device and we agree. The two primary challenges that remain largely unsolved are *protection* and *low-latency*. However, addressing either of these challenges might additionally complicate *scheduling*.

The subsequent work by the same authors describes *resource management*, *security*, *data consistency* and *usability* as open research questions [10]. Some of these have clear overlap such as *security* and *protection*.

However, this overlap is logical as one of these challenges are from the perspective of what requires additional changes to existing operating systems[10]. While they other aims to identify open research questions which hinder adoption.

Across all these open research questions hindering adoption we see one clear theme. The questions resolve around trying to share information between the host and device. Be it replication maps or filesystem information. We also see this in the different programming models as described in the same work. Similarly we see this problem appear when trying to offload the reduce stage of MapReduce to CSxs [62].

This sharing requirement is natural given that CS requires complete vertical integration. From nand flash interface, FPGA / CPU design, programming methodology, peripheral interface, hardware abstraction layer and user programming interface. Across this entire stack changes are needed.

However, we argue that combining block filesystem access with CS introduces unfeasible levels of complexity in the FTL of the device. Just like two filesystem partitions wont share overlapping storage space, partitions and CS should not share the same storage space. Research should instead focus on using CS platforms to create virtual filesystems on top of them. This is similar to CephFS that moved away from using XFS as underlying filesystem. Instead CephFS now uses BlueStore object storage which uses the whole storage device without a filesystem. Additionally, we already see this approach in a recently proposed CS key-value store [42]. In addition, moving away from this requirement already lead to some works achieving real world deployments [18].

We propose to abolish the idea of sharing fundamental information between the device and host, the attempt to bridge The semantic gap. Instead all effort should be focused on developing CSx interfaces and APIs and how to develop host features such as virtual file systems on top of those.

[10]With operating system being used to refer to both the overall software on the host and the device in the context of this work.

But most important, even more so than the open research questions, is the lack of open-research causing many works to have to reinvent the wheel. The lack of open-source hardware and software design in the field not only severely hinders reproducibility but also the speed at which the field advances. Lack of access to the designs or software of previous works requires new research to often start from scratch. In our opinion any work utilizing newly written software or hardware designs should release these under an open-source license or be rejected for publication period.

9 FUTURE PREDICTION

Full blown heterogeneous architectures will become the norm [58, 81]. We already see this transition in devices such as iPhones. The use of neural accelerators is prevalent in modern mobile SoCs. More recently, we see a push for neural accelerators in datacentres as well. It is only a matter of time before we start seeing these appear in consumer desktop platforms as well.

9.1 Interface Evolution

To support such a heterogeneous architectures several interfaces must evolve. One of these interfaces is the block device interface which will gradually dissapear from flash based devices [16]. We will see two prominent types of interfaces emerge for flash based devices. The first being a key-value store with offloaded scan and gather functionality. The second being an accelerator interface much like Vulkan and OpenCL.

The overall adoption of these accelerator interfaces depends on our performance as software and hardware engineers. We must design a good performing system with easy integration into existing operating systems and programs. The success of this interface must not be tied to any specific programming model supporting dataflow, shared memory and client / server as minimum. The hardware abstraction layer will use a dataflow like programming model. While the support for shared memory is built on top of this dataflow model. Lastly, the client / server model is in turn built on top of the shared memory model.

Using these variety of programming models we will see operating systems built high level abstractions such as virtual filesystems. In addition we will see specialized applications such databases stored entirely on one or multiple CSxs.

Naturally, presenting an entire SSD for a single purpose can be undesirable. To alleviate this new NVme interface extensions like Zoned Namespaces (ZNS) will be relied upon heavily. While ZNS still presents a block like interface it much better represents the underlying nand Flash architecture eliminating complexity in the FTL, excessive garbage collection and write amplification.

ZNS will be the predominant interface for CS until a complete transition away from block like interfaces is made. There is already significant literature suggesting the adoption of ZNS [15, 47, 53, 87]. With already the first CS work utilizing it as well [35].

Beyond storage interfaces such as ZNS we will see more and more use of interconnects being used across various components in desktop and server systems. Some of these will be cache coherent such as OpenCAPI [23] or CCIX [21]. Others aimed specifically at communication for accelerators such as GenZ [22] and CXL2 [50].

No CS will not depend on if it makes sense from an energy efficient or architectural viewpoint to use. Instead it relies entirely on the ease of integration, reliability, predictability and performance of the overall systems we manage to design.

9.2 The Arms Race

Both the throughput of nand flash as well as the bandwidth of interconnects such as PCIe will increase over the coming years. There will be no clear winner which sets the president for requiring a well designed CS system if we want to see adoption. PCIe will start using realtime compression,

similar to how HDMI 2.1 support Display Stream Compression (DSC). The maximum permissible length of PCIe traces as well as DisplayPort and HDMI cables will decrease further and further. Meanwhile the cost of manufacturer of components such as motherboards will increase. This is due to the signal integrity requirements posed by such high-speed differential signaling.

Eventually this arms race will reach a tipping point until the use of onboard fiber interconnects becomes the norm. These fiber channels will be integrated into regular PCBs occupying only several layers of the PCB stack. This will allow for seemingly unlimited bandwidth with nearly no signal degradation and no cross talk. In the near future, copper based differential signaling will die.

10 FUTURE WORK

One key element of a literature survey is the methodology used to identify, categorize and analyze related literature. Future work could better address certain characteristics of our approach. Key areas for improvement being accuracy and fairness.

Firstly, plainly assigning keywords to works based on the search term used to find them is naïve. A better approach would categorize works based on the keywords appearing in title and abstract.

Secondly, the selection procedure is critical when dealing with a limited number of selected literature. Analyzing works ordered by publishing date introduces unwanted bias. This bias will favor selecting more historical works over recent contributions. This is due to a steadily approaching hard limit on the amount of works that can still be selected. A better approach would analyze works from the initial selection at random.

Lastly, the assessment of novelty can be substantially improved to account for the different phases of a research field. This is because preliminary research in a field has a tendency to appear more novel than later works. As example, the first implementation of a PFS architecture for database query offloading can appear more novel than subsequent works. These subsequent works could address multi-user tenancy both being equally important to progress the field. A better approach should separate the works in different phases based on date. As well as identifying the current open research question in later phases of the field. The assessment in later phases should than evaluate if any of the open research questions are being addressed.

Three works stand out that would potentially be included should our approach have been improved as proposed. These are NASCENT [74], FERMAT [100] and COPRAO [46].

Beyond improvements to our approach we see several potential topics that might suit other literature surveys. First is relating the development of CS to support technologies such as ZNS or OpenCAPI. While we addressed such support technologies briefly we feel much more information can be derived here. Similarly, the advancement of open standards and ISAs should form an interesting topic with many recent contributions.

In addition there are several areas of our work that could be easily expanded. One area is the ubiquity of computational storage. Due to constrained amounts of literature that could be selected this part of our work is not nearly as exhaustive as it could be. In addition, we have identified many different fixed and programmable flash based CSxs. However, there is much more information that could be compared across these works. These comparisons include the integration of filesystem support or lack there of. Moreover, how transparent the offloading is to the end user or how much energy and performance improvement the work constituted.

Clearly there is plenty of research opportunity and exciting new technologies on the horizon for the field of CS. Will you help us make it a success?

11 GLOSSARY

(1) CPU - Central Processing Unit
(2) DRAM - Dynamic Random Access Memory

(3) PCIe - Peripheral Component Interconnect Express
(4) CS - Computational Storage
(5) CSx - Computational Storage Device
(6) PFS - Programmable Flash Storage
(7) PFSD - Programmable Flash Storage Device
(8) OCSSD - Open Channel SSD
(9) ZNS - Zoned Namespaces
(10) HAL - Hardware Abstraction Layer
(11) ACL - Access Level Control
(12) CSF - Computational Storage Function
(13) FPGA - Field Programmable Gate Array
(14) DSP - Digital Signal Processor
(15) DAG - Directed Acyclic Graph
(16) NIC - Network Interface Controller
(17) PIM - Programmable in Memory
(18) CSP - Computational Storage Processor
(19) CSA - Computational Storage Array
(20) CSD - Computational Storage Driver
(21) CSE - Computational Storage Engine
(22) CSEE - Computational Storage Execution Environment
(23) CSR - Computational Storage Resource
(24) CSF - Computation Storage Function
(25) FDM - Function Data Memory
(26) ADFM - Allocated Function Data memory
(27) HAL - Hardware Abstraction Layer
(28) ICD - Installable Client Driver
(29) FTL - Flash Translation Layer
(30) MPI - Message Passing Interface
(31) VLIW - Very Long Instruction Word
(32) LUN - Logical Unit
(33) Upstream - When a certain patch or feature has been merged into the main repository of a project.
(34) GUI - Graphical User Interface
(35) IDE - Integrated Development Environment
(36) Target - A target is something that can be compiled or generated and sometimes executed. Examples include PDFs using LaTeX, binaries or libraries.
(37) Binary - A binary is a file containing machine instructions that can be directly executed. A compiled C file containing a main method is a binary while a shell script is not. Typically these binaries contain metadata information to aid the underlying operating system in executing them, a common format for this metadata is ELF.
(38) ABI - Application Binary Interface.
(39) API - Application Programming Interface.
(40) SCSI - Small Computer System Interface
(41) ATA - AT Attachment
(42) ZBC - (SCSI) Zoned Block Command
(43) ZAC - (ATA) Zoned ATA Commands

REFERENCES

[1] Noor Abbani, Ali Ali, Doa'A Al Otoom, Mohamad Jomaa, Mageda Sharafeddine, Hassan Artail, Haitham Akkary, Mazen A.R. Saghir, Mariette Awad, and Hazem Hajj. 2011. A Distributed Reconfigurable Active SSD Platform for Data Intensive Applications. In *2011 IEEE International Conference on High Performance Computing and Communications*. 25–34. https://doi.org/10.1109/HPCC.2011.14

[2] Anurag Acharya, Mustafa Uysal, and Joel Saltz. 1998. *Active Disks.* Technical Report. University of California, Santa Barbara.

[3] Anurag Acharya, Mustafa Uysal, and Joel Saltz. 1998. Active Disks: Programming Model, Algorithms and Evaluation. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS VIII)*. Association for Computing Machinery, New York, NY, USA, 81–91. https://doi.org/10.1145/291069.291026

[4] Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory R. Ganger, and George Amvrosiadis. 2019. File Systems Unfit as Distributed Storage Backends: Lessons from 10 Years of Ceph Evolution. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 353–369. https://doi.org/10.1145/3341301.3359656

[5] AMD. 2021. Smart Access Memory. https://www.amd.com/en/technologies/smart-access-memory

[6] Duck-Ho Bae, Jin-Hyung Kim, Sang-Wook Kim, Hyunok Oh, and Chanik Park. 2013. Intelligent SSD: A Turbo for Big Data Mining. In *Proceedings of the 22nd ACM International Conference on Information and Knowledge Management* (San Francisco, California, USA) *(CIKM '13)*. Association for Computing Machinery, New York, NY, USA, 1573–1576. https://doi.org/10.1145/2505515.2507847

[7] Rajeev Balasubramonian, Jichuan Chang, Troy Manning, Jaime H. Moreno, Richard Murphy, Ravi Nair, and Steven Swanson. 2014. Near-Data Processing: Insights from a MICRO-46 Workshop. *IEEE Micro* 34, 4 (2014), 36–42. https://doi.org/10.1109/MM.2014.55

[8] Nastaran Baradaran and Pedro C. Diniz. 2008. A Compiler Approach to Managing Storage and Memory Bandwidth in Configurable Architectures. *ACM Trans. Des. Autom. Electron. Syst.* 13, 4, Article 61 (Oct. 2008), 26 pages. https://doi.org/10.1145/1391962.1391969

[9] Antonio Barbalace, Martin Decky, Javier Picorel, and Pramod Bhatotia. 2020. BlockNDP: Block-Storage Near Data Processing. In *Proceedings of the 21st International Middleware Conference Industrial Track* (Delft, Netherlands) *(Middleware '20)*. Association for Computing Machinery, New York, NY, USA, 8–15. https://doi.org/10.1145/3429357.3430519

[10] Antonio Barbalace and Jaeyoung Do. 2021. Computational Storage: Where Are We Today?. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2021/papers/cidr2021_paper29.pdf

[11] Antonio Barbalace, Anthony Iliopoulos, Holm Rauchfuss, and Goetz Brasche. 2017. 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* (Whistler, BC, Canada) *(HotOS '17)*. Association for Computing Machinery, New York, NY, USA, 56–61. https://doi.org/10.1145/3102980.3102990

[12] R. Baum and D. Hsiao. 1976. Database Computers? A Step Towards Data Utilities. *IEEE Trans. Comput.* 25, 12 (dec 1976), 1254–1259. https://doi.org/10.1109/TC.1976.1674592

[13] Andreas Becher, Stefan Wildermann, and Jürgen Teich. 2018. Optimistic Regular Expression Matching on FPGAs for Near-Data Processing. In *Proceedings of the 14th International Workshop on Data Management on New Hardware* (Houston, Texas) *(DAMON '18)*. Association for Computing Machinery, New York, NY, USA, Article 4, 3 pages. https://doi.org/10.1145/3211922.3211926

[14] A. Biliris and E. Panagos. 1995. A high performance configurable storage manager. In *Proceedings of the Eleventh International Conference on Data Engineering*. 35–43. https://doi.org/10.1109/ICDE.1995.380412

[15] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R. Ganger, and George Amvrosiadis. 2021. ZNS: Avoiding the Block Interface Tax for Flash-based SSDs. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 689–703. https://www.usenix.org/conference/atc21/presentation/bjorling

[16] Matias Bjørling, Philippe Bonnet, Luc Bouganim, and Niv Dayan. 2012. *The Necessary Death of the Block Device Interface.* Technical Report. IT University of Copenhagen.

[17] Simona Boboila, Youngjae Kim, Sudharshan S. Vazhkudai, Peter Desnoyers, and Galen M. Shipman. 2012. Active Flash: Out-of-core data analytics on flash storage. In *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*. 1–12. https://doi.org/10.1109/MSST.2012.6232366

[18] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, Zhenjun Liu, Feng Zhu, and Tong Zhang. 2020. POLARDB Meets Computational Storage: Efficiently Support Analytical Workloads in Cloud-Native Relational Database. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 29–41. https://www.usenix.org/conference/fast20/presentation/cao-

wei
[19] Chao Chen and Yong Chen. 2012. Dynamic Active Storage for High Performance I/O. In *2012 41st International Conference on Parallel Processing*. 379–388. https://doi.org/10.1109/ICPP.2012.22
[20] Sangyeun Cho, Chanik Park, Hyunok Oh, Sungchan Kim, Youngmin Yi, and Gregory R. Ganger. 2013. Active Disk Meets Flash: A Case for Intelligent SSDs. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing* (Eugene, Oregon, USA) *(ICS '13)*. Association for Computing Machinery, New York, NY, USA, 91–102. https://doi.org/10.1145/2464996.2465003
[21] CCIX Consortium. 2021. Coherent interconnect technologies. https://www.ccixconsortium.com/.
[22] GenZ Consortium. 2021. open memory-semantic access. https://genzconsortium.org/.
[23] OpenCAPI Consortium. 2021. Low Latency Memory. https://opencapi.org/.
[24] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. https://doi.org/10.1145/1327452.1327492
[25] David J. DeWitt. 1979. Query Execution in DIRECT. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data* (Boston, Massachusetts) *(SIGMOD '79)*. Association for Computing Machinery, New York, NY, USA, 13–22. https://doi.org/10.1145/582095.582098
[26] Ashutosh Dhar, Sitao Huang, Jinjun Xiong, Damir Jamsek, Bruno Mesnet, Jian Huang, Nam Sung Kim, Wen-mei Hwu, and Deming Chen. 2019. Near-Memory and In-Storage FPGA Acceleration for Emerging Cognitive Computing Workloads. In *2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 68–75. https://doi.org/10.1109/ISVLSI.2019.00021
[27] Jaeyoung Do, Victor C. Ferreira, Hossein Bobarshad, Mahdi Torabzadehkashi, Siavash Rezaei, Ali Heydarigorji, Diego Souza, Brunno F. Goldstein, Leandro Santiago, Min Soo Kim, Priscila M. V. Lima, Felipe M. G. França, and Vladimir Alves. 2020. Cost-Effective, Energy-Efficient, and Scalable Storage Computing for Large-Scale AI Applications. *ACM Trans. Storage* 16, 4, Article 21 (Oct. 2020), 37 pages. https://doi.org/10.1145/3415580
[28] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. 2013. Query Processing on Smart SSDs: Opportunities and Challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 1221–1230. https://doi.org/10.1145/2463676.2465295
[29] Eideticom. 2021. NoLoad products. https://www.eideticom.com/products.html.
[30] Bryan Fink. 2012. Distributed Computation on Dynamo-Style Distributed Storage: Riak Pipe. In *Proceedings of the Eleventh ACM SIGPLAN Workshop on Erlang Workshop* (Copenhagen, Denmark) *(Erlang '12)*. Association for Computing Machinery, New York, NY, USA, 43–50. https://doi.org/10.1145/2364489.2364497
[31] OpenPower Foundation. 2021. OpenPower. https://openpowerfoundation.org/.
[32] RISC-V Foundation. 2021. RISC-V. https://riscv.org/.
[33] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. 2016. Biscuit: A Framework for near-Data Processing of Big Data Workloads. In *Proceedings of the 43rd International Symposium on Computer Architecture* (Seoul, Republic of Korea) *(ISCA '16)*. IEEE Press, 153–165. https://doi.org/10.1109/ISCA.2016.23
[34] Saransh Gupta, Justin Morris, Mohsen Imani, Ranganathan Ramkumar, Jeffrey Yu, Aniket Tiwari, Baris Aksanli, and Tajana Šimunić Rosing. 2020. THRIFTY: Training with Hyperdimensional Computing across Flash Hierarchy. In *Proceedings of the 39th International Conference on Computer-Aided Design* (Virtual Event, USA) *(ICCAD '20)*. Association for Computing Machinery, New York, NY, USA, Article 27, 9 pages. https://doi.org/10.1145/3400302.3415723
[35] Kyuhwa Han, Hyunho Gwak, Dongkun Shin, and Jooyoung Hwang. 2021. ZNS+: Advanced Zoned Namespace Interface for Supporting In-Storage Zone Compaction. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 147–162. https://www.usenix.org/conference/osdi21/presentation/han
[36] Zsolt István, David Sidler, and Gustavo Alonso. 2017. Caribou: Intelligent Distributed Storage. *Proc. VLDB Endow.* 10, 11 (Aug. 2017), 1202–1213. https://doi.org/10.14778/3137628.3137632
[37] Insoon Jo, Duck-Ho Bae, Andre S. Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel D. G. Lee, and Jaeheon Jeong. 2016. YourSQL: A High-Performance Database System Leveraging in-Storage Computing. *Proc. VLDB Endow.* 9, 12 (Aug. 2016), 924–935. https://doi.org/10.14778/2994509.2994512
[38] Yangwook Kang, Yang-suk Kee, Ethan L. Miller, and Chanik Park. 2013. Enabling cost-effective data processing with smart SSD. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*. 1–12. https://doi.org/10.1109/MSST.2013.6558444
[39] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. 1998. A Case for Intelligent Disks (IDISKs). *SIGMOD Rec.* 27, 3 (Sept. 1998), 42–52. https://doi.org/10.1145/290593.290602
[40] Jakub Kicinski. 2018. Using eBPF as a heterogeneous processing ABI. Linux Plumbers Conference. Accessed: 2021-05-15, http://vger.kernel.org/lpc_bpf2018_talks/Using_eBPF_as_a_heterogeneous_processing_ABI_LPC_2018.pdf.

[41] Gunjae Koo, Kiran Kumar Matam, Te I, H. V. Krishna Giri Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, and Murali Annavaram. 2017. Summarizer: Trading Communication with Computing near Storage. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (Cambridge, Massachusetts) *(MICRO-50 '17)*. Association for Computing Machinery, New York, NY, USA, 219–231. https://doi.org/10.1145/3123939.3124553

[42] Jinhyung Koo, Junsu Im, Jooyoung Song, Juhyung Park, Eunji Lee, Bryan S. Kim, and Sungjin Lee. 2021. Modernizing File System through In-Storage Indexing. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 75–92. https://www.usenix.org/conference/osdi21/presentation/koo

[43] Kornilios Kourtis, Animesh Trivedi, and Nikolas Ioannou. 2020. Safe and Efficient Remote Application Code Execution on Disaggregated NVM Storage with eBPF. arXiv:2002.11528 [cs.DC] https://arxiv.org/abs/2002.11528v1

[44] Fritz Kruger. 2016. CPU Bandwidth – The Worrisome 2020 Trend. https://blog.westerndigital.com/cpu-bandwidth-the-worrisome-2020-trend/.

[45] Chinmay Kulkarni, Sara Moore, Mazhar Naqvi, Tian Zhang, Robert Ricci, and Ryan Stutsman. 2018. Splinter: Bare-Metal Extensions for Multi-Tenant Low-Latency Storage. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 627–643. https://www.usenix.org/conference/osdi18/presentation/kulkarni

[46] B. G. Lekshmi and Klaus Meyer-Wegener. 2021. COPRAO: A Capability Aware Query Optimizer for Reconfigurable Near Data Processors. In *2021 IEEE 37th International Conference on Data Engineering Workshops (ICDEW)*. 54–59. https://doi.org/10.1109/ICDEW53142.2021.00017

[47] Alberto Lerner and Philippe Bonnet. 2021. Not Your Grandpa's SSD: The Era of Co-Designed Storage Devices. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD/PODS '21)*. Association for Computing Machinery, New York, NY, USA, 2852–2858. https://doi.org/10.1145/3448016.3457540

[48] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan R. K. Ports, Irene Zhang, Ricardo Bianchini, Haryadi S. Gunawi, and Anirudh Badam. 2020. 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* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 591–605. https://doi.org/10.1145/3373376.3378531

[49] Chyuan Shiun Lin, Diane C. P. Smith, and John Miles Smith. 1976. The Design of a Rotating Associative Memory for Relational Database Applications. *ACM Trans. Database Syst.* 1, 1 (March 1976), 53–65. https://doi.org/10.1145/320434.320447

[50] Compute Express Link. 2021. CXL 2 specifications. https://www.computeexpresslink.org/download-the-specification.

[51] Linux. 2021. Efficient IO with io uring. https://tinyurl.com/kerneldk-iouring

[52] Linux. 2021. Generic PCI UIO driver. https://www.kernel.org/doc/html/latest/driver-api/uio-howto.html#generic-pci-uio-driver

[53] Chun Liu. 2021. Beyond Zoned Namespace - What Do Applications Want? https://www.snia.org/educational-library/beyond-zoned-namespace-what-do-applications-want-2021

[54] Corne Lukken. 2021. Literature study support file. https://dancloud.dantalion.nl/index.php/s/3T0HkpmfgYPKJb7/download.

[55] Ricardo Macedo, João Paulo, José Pereira, and Alysson Bessani. 2020. A Survey and Classification of Software-Defined Storage Systems. *ACM Comput. Surv.* 53, 3, Article 48 (May 2020), 38 pages. https://doi.org/10.1145/3385896

[56] Steve Morgan. 2021. The 2020 Data Attack Surface Report. https://1c7fab3im83f5gqiow2qqs2k-wpengine.netdna-ssl.com/wp-content/uploads/2020/12/ArcserveDataReport2020.pdf

[57] NGD Systems. 2021. Newport Platform. https://www.ngdsystems.com/solutions.

[58] Joel Nider and Alexandra (Sasha) Fedorova. 2021. The Last CPU. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Ann Arbor, Michigan) *(HotOS '21)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/3458336.3465291

[59] Xiangyong Ouyang, David Nellans, Robert Wipfel, David Flynn, and Dhabaleswar K. Panda. 2011. Beyond Block I/O: Rethinking Traditional Storage Primitives. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA '11)*. IEEE Computer Society, USA, 301–311.

[60] overclock3d. 2019. PCIe 6.0 is on-track for a 2021 release - Launches 0.3 Spec. https://www.overclock3d.net/news/misc_hardware/pcie_6_0_is_on-track_for_a_2021_release_-_launches_0_3_spec/1.

[61] E. A. Ozkarahan, S. A. Schuster, and K. C. Sevcik. 1977. Performance Evaluation of a Relational Associative Processor. *ACM Trans. Database Syst.* 2, 2 (June 1977), 175–195. https://doi.org/10.1145/320544.320553

[62] Dongchul Park, Jianguo Wang, and Yang-Suk Kee. 2016. In-Storage Computing for Hadoop MapReduce Framework: Challenges and Possibilities. *IEEE Trans. Comput.* (2016), 1–1. https://doi.org/10.1109/TC.2016.2595566

[63] Arnab Paul, Ryan Chard, Kyle Chard, Steven Tuecke, Ali Butt, and Ian Foster. 2019. FSMonitor: Scalable File System Monitoring for Arbitrary Storage Systems. 1–11. https://doi.org/10.1109/CLUSTER.2019.8891045

[64] Shuyi Pei, Jing Yang, and Qing Yang. 2019. REGISTOR: A Platform for Unstructured Data Processing Inside SSD Storage. *ACM Trans. Storage* 15, 1, Article 7 (March 2019), 24 pages. https://doi.org/10.1145/3310149

[65] Ilia Petrov, Tobias Vincon, Andreas Koch, Julian Oppermann, Sergey Hardock, and Christian Riegger. 2018. Active Storage. In *Encyclopedia of Big Data Technologies*. Springer International Publishing, 1–8. https://doi.org/10.1007/978-3-319-63962-8_309-1

[66] Juan Piernas, Jarek Nieplocha, and Evan J. Felix. 2007. Evaluation of Active Storage Strategies for the Lustre Parallel File System. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing* (Reno, Nevada) *(SC '07)*. Association for Computing Machinery, New York, NY, USA, Article 28, 10 pages. https://doi.org/10.1145/1362622.1362660

[67] Keshav Pingali. 2018. The von Neumann Bottleneck Revisited. https://www.sigarch.org/the-von-neumann-bottleneck-revisited/.

[68] Nancy C. Ramsay. 1990. Integration of the optical storage processor and the DBC/1012 database computer. *[1990] Digest of papers. Tenth IEEE Symposium on Mass Storage Systems@m_Crisis in Mass Storage* (1990), 94–97.

[69] Erik Riedel. 1999. *Active Disks - Remote Execution for Network-Attached Storage.* Ph. D. Dissertation. School of Computer Science, Carnegie Mellon University.

[70] Erik Riedel, Garth A. Gibson, and Christos Faloutsos. 1998. Active Storage for Large-Scale Data Mining and Multimedia. In *Proceedings of the 24rd International Conference on Very Large Data Bases (VLDB '98)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 62–73.

[71] Zhenyuan Ruan, Tong He, and Jason Cong. 2019. INSIDER: Designing In-Storage Computing System for Emerging High-Performance Drive. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 379–394. https://www.usenix.org/conference/atc19/presentation/ruan

[72] Karl Rupp. 2018. 42 Years of Microprocessor Trend Data. https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/.

[73] Marta Rybczyńska. 2018. Device-to-device memory-transfer offload with P2PDMA. =https://lwn.net/Articles/767281/. Accessed: 2021-08-21.

[74] Sahand Salamat, Armin Haj Aboutalebi, Behnam Khaleghi, Joo Hwan Lee, Yang Seok Ki, and Tajana Rosing. 2021. NASCENT: Near-Storage Acceleration of Database Sort on SmartSSD. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Virtual Event, USA) *(FPGA '21)*. Association for Computing Machinery, New York, NY, USA, 262–272. https://doi.org/10.1145/3431920.3439298

[75] Samsung. 2021. Samsung SmartSSD Computational Storage. https://www.nimbix.net/wp-content/uploads/2019/07/SmartSSD_Product_Brief_Digital.pdf.

[76] Scaleflux. 2021. Scaleflux Products. https://www.scaleflux.com/product/1-1.

[77] Leah Schoeb. 2020. What Happens when Compute meets Storage? – Computational Storage. https://www.snia.org/educational-library/what-happens-when-compute-meets-storage-%E2%80%93-computational-storage-2020

[78] Seagate. 2014. Kinetic HDD. https://www.seagate.com/files/www-content/product-content/hdd-fam/kinetic-hdd/en-us/docs/kinetic-ds1835-1-1110us.pdf

[79] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. 2014. Willow: A User-Programmable SSD. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 67–80. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/seshadri

[80] Aisha Siddiqa, Ahmad Karim, and Abdullah Gani. 2017. Big data storage technologies: a survey. *Frontiers of Information Technology & Electronic Engineering* 18, 8 (Aug. 2017), 1040–1070. https://doi.org/10.1631/fitee.1500441

[81] Mark Silberstein. 2017. OmniX: An Accelerator-Centric OS for Omni-Programmable Systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems* (Whistler, BC, Canada) *(HotOS '17)*. Association for Computing Machinery, New York, NY, USA, 69–75. https://doi.org/10.1145/3102980.3102992

[82] Muthian Sivathanu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2002. Evolving RPC for Active Storage. *SIGPLAN Not.* 37, 10 (Oct. 2002), 264–276. https://doi.org/10.1145/605432.605425

[83] SNIA. 2021. Computational Storage Architecture and Programming Model Version 0.8 Revision 0. https://www.snia.org/sites/default/files/technical_work/PublicReview/SNIA-Computational-Storage-Architecture-and-Programming-Model-0.8R0-2021.06.09.pdf.

[84] SNIA. 2021. Intel Optane. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html.

[85] SNIA. 2021. SNIA PM+CS Summit. https://www.snia.org/pm-summit.

[86] SNIA. 2021. SNIA Storage Developer Conference. https://www.snia.org/news-events/storage-developer-conference.

[87] Theano Stavrinos, Daniel S. Berger, Ethan Katz-Bassett, and Wyatt Lloyd. 2021. Don't Be a Blockhead: Zoned Namespaces Make Work on Conventional SSDs Obsolete. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Ann Arbor, Michigan) *(HotOS '21)*. Association for Computing Machinery, New York, NY, USA, 144–151. https://doi.org/10.1145/3458336.3465300

[88] Stanley Y. W. Su and G. Jack Lipovski. 1975. CASSM: A Cellular System for Very Large Data Bases. In *Proceedings of the 1st International Conference on Very Large Data Bases* (Framingham, Massachusetts) *(VLDB '75)*. Association for Computing Machinery, New York, NY, USA, 456–472. https://doi.org/10.1145/1282480.1282518

[89] Devesh Tiwari, Simona Boboila, Sudharshan S. Vazhkudai, Youngjae Kim, Xiaosong Ma, Peter J. Desnoyers, and Yan Solihin. 2013. Active Flash: Towards Energy-Efficient, in-Situ Data Analytics on Extreme-Scale Machines. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies* (San Jose, CA) *(FAST'13)*. USENIX Association, USA, 119–132.

[90] Mahdi Torabzadehkashi, Ali Heydarigorji, Siavash Rezaei, Hosein Bobarshad, Vladimir Alves, and Nader Bagherzadeh. 2019. Accelerating HPC Applications Using Computational Storage Devices. In *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. 1878–1885. https://doi.org/10.1109/HPCC/SmartCity/DSS.2019.00259

[91] Tobias Vinçon, Arthur Bernhardt, Ilia Petrov, Lukas Weber, and Andreas Koch. 2020. NKV: Near-Data Processing with KV-Stores on Native Computational Storage. In *Proceedings of the 16th International Workshop on Data Management on New Hardware* (Portland, Oregon) *(DaMoN '20)*. Association for Computing Machinery, New York, NY, USA, Article 10, 11 pages. https://doi.org/10.1145/3399666.3399934

[92] Jianguo Wang, Dongchul Park, Yang-Suk Kee, Yannis Papakonstantinou, and Steven Swanson. 2016. SSD In-Storage Computing for List Intersection. In *Proceedings of the 12th International Workshop on Data Management on New Hardware* (San Francisco, California) *(DaMoN '16)*. Association for Computing Machinery, New York, NY, USA, Article 4, 7 pages. https://doi.org/10.1145/2933349.2933353

[93] wccftech. 2021. ps5 io system. https://wccftech.com/ps5-io-system-to-be-supercharged-by-oodle-texture-bandwidth-goes-up-to-17-38gb-s/.

[94] Sage A. Weil, Andrew W. Leung, Scott A. Brandt, and Carlos Maltzahn. 2007. RADOS: A Scalable, Reliable Storage Service for Petabyte-Scale Storage Clusters. In *Proceedings of the 2nd International Workshop on Petascale Data Storage: Held in Conjunction with Supercomputing '07* (Reno, Nevada) *(PDSW '07)*. Association for Computing Machinery, New York, NY, USA, 35–44. https://doi.org/10.1145/1374596.1374606

[95] Rajiv Wickremesinghe, Jeffrey S. Chase, and Jeffrey S. Vitter. 2002. Distributed computing with load-managed active storage. In *In IEEE International Symposium on High Performance Distributed Computing (HPDC-11*. IEEE Computer Society Press, 24–34.

[96] Peter Wilcox and Heiner Litz. 2021. Design for Computational Storage Simulation Platform. In *Proceedings of the Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems* (Online Event, United Kingdom) *(CHEOPS '21)*. Association for Computing Machinery, New York, NY, USA, Article 5, 8 pages. https://doi.org/10.1145/3439839.3459085

[97] Louis Woods, Zsolt István, and Gustavo Alonso. 2014. Ibex: An Intelligent Storage Engine with Support for Advanced SQL Offloading. *Proc. VLDB Endow.* 7, 11 (July 2014), 963–974. https://doi.org/10.14778/2732967.2732972

[98] Yulai Xie, Dan Feng, Yan Li, and Darrell D.E. Long. 2016. Oasis: An active storage framework for object storage platform. *Future Generation Computer Systems* 56 (2016), 746–758. https://doi.org/10.1016/j.future.2015.08.011

[99] Shuotao Xu, Sungjin Lee, Sang-Woo Jun, Ming Liu, Jamey Hicks, and Arvind. 2016. Bluecache: A Scalable Distributed Flash-Based Key-Value Store. *Proc. VLDB Endow.* 10, 4 (Nov. 2016), 301–312. https://doi.org/10.14778/3025111.3025113

[100] Yu Zou and Mingjie Lin. 2021. FERMAT: FPGA-Accelerated Heterogeneous Computing Platform Near NVMe Storage. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 262–262. https://doi.org/10.1109/FCCM51124.2021.00049