Storage Systems (StoSys)
XM_0092

Lecture 8: Programmable Storage

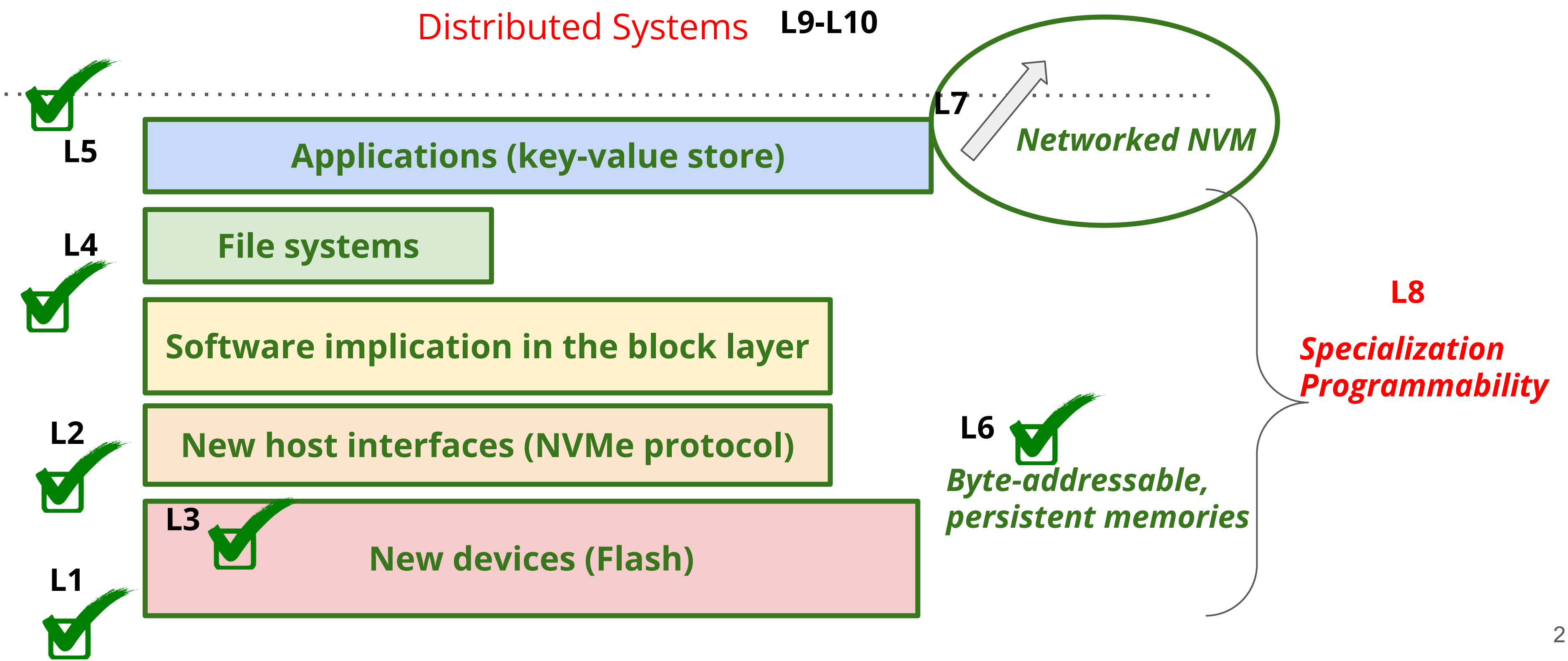
Animesh Trivedi

https://stonet-research.github.io/

Autumn 2023, Period 1

VRIJE UNIVERSITEIT AMSTERDAM

The layered approach in the lectures



Distributed Systems L9-L10

L7 Networked NVM

L5 Applications (key-value store)

L4 File systems

Software implication in the block layer

L2 New host interfaces (NVMe protocol)

L3 L1 New devices (Flash)

L6 Byte-addressable, persistent memories

L8 Specialization Programmability

Guest Lecture on Oct 18th 2023

Title: Ten things we've learned from running production infrastructure at Google

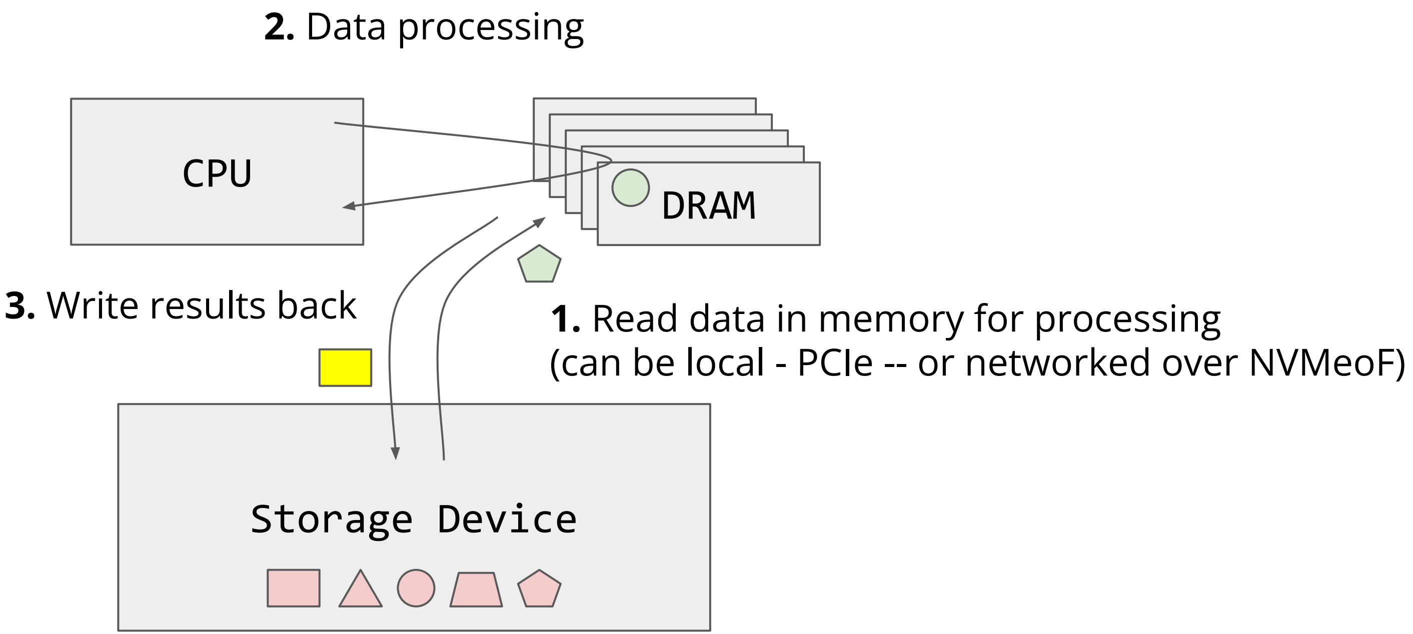
Abstract: This talk covers 10 key lessons we have learned in 20 years of practising SRE at Google and the themes that have emerged throughout these lessons, which cover areas such as SRE culture, operations and change management. Attendees will take away specific learnings that have contributed to SRE being successful at Google.

Any Guesses?

Why would we need programmable storage? And what is it actually?

Jaeyoung Do, Sudipta Sengupta, and Steven Swanson. 2019. Programmable solid-state storage in future cloud datacenters. *Commun. ACM* 62, 6 (June 2019), 54–62.

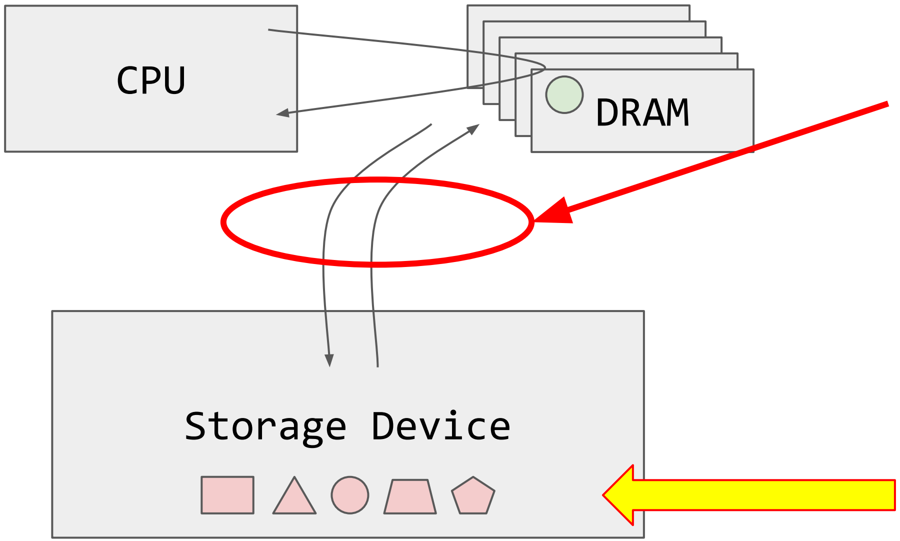
Conventional Data Processing (simplified)


2. Data processing
CPU
DRAM
3. Write results back
1. Read data in memory for processing
(can be local - PCIe -- or networked over NVMeoF)
Storage Device

A basic model of how storage and data processing is organized typically

What are the challenges here?

Key Challenge - Data Movement Wall



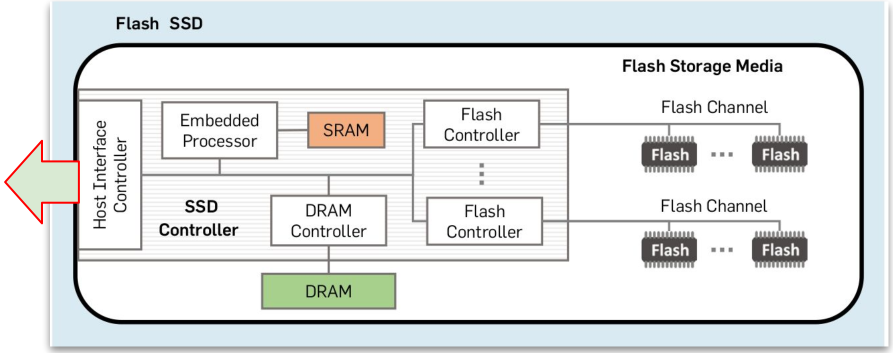
The <u>network</u> (local or external) is a bottleneck.

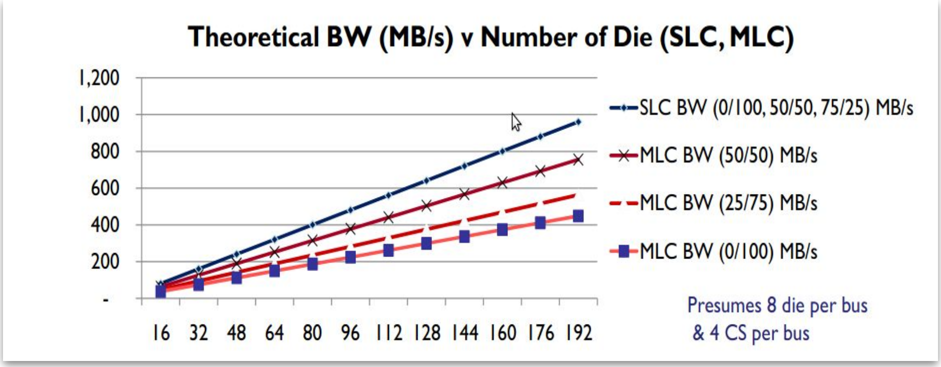
Why now? Emergence of Flash and internal device parallelism creates a data movement bottleneck!

The amount of data generated and processed is increasing significantly Recall: 200 Zettabytes by 2025

Also see Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. *Pump Up the Volume: Processing Large Data on GPUs with Fast Interconnects.* In Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20).

Recall: Flash Internal Structure





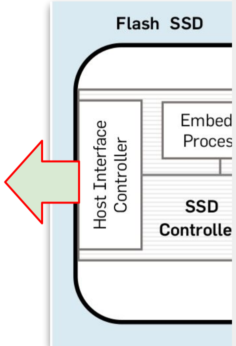
Flash devices consist of multiple independent packages, die, or planes

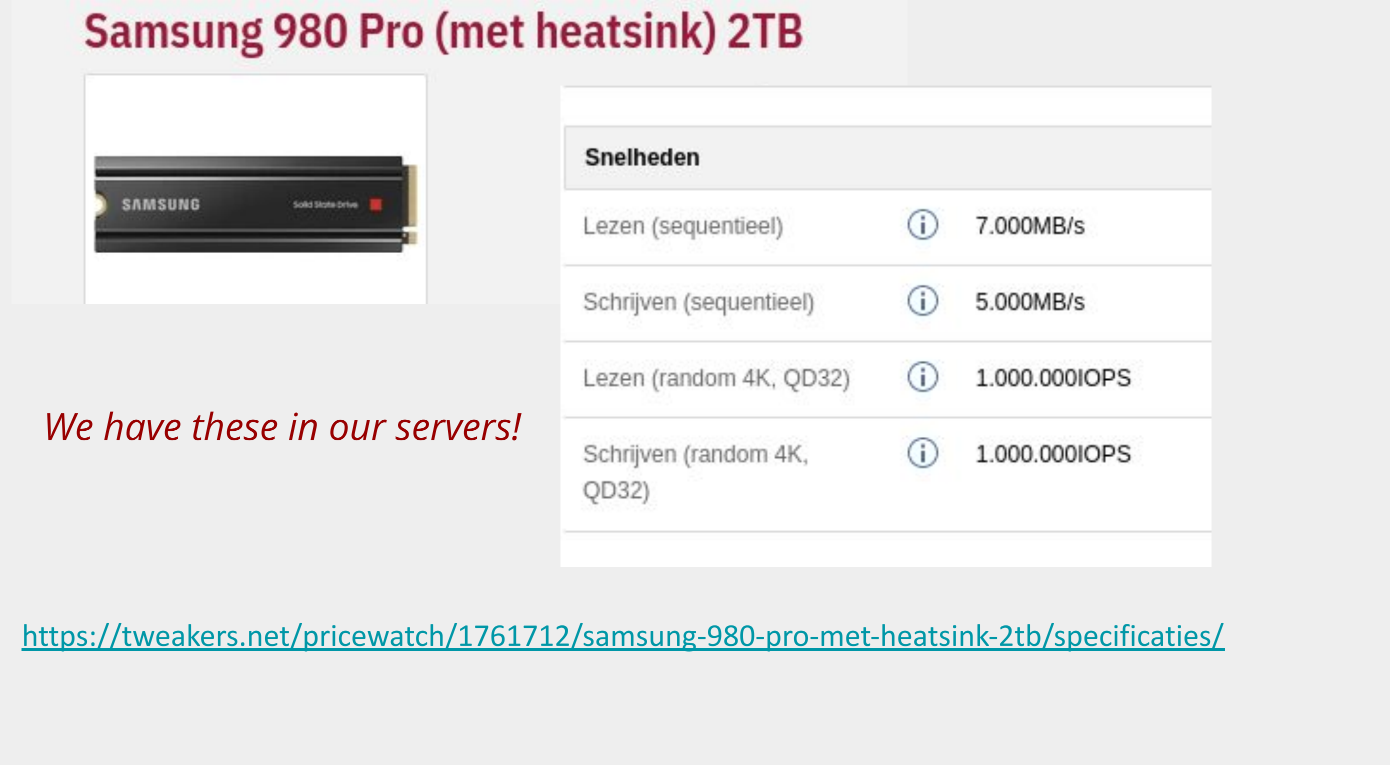
These components can work in parallel, giving a large amount of bandwidth

A single server can host multiple PCIe connected flash devices

https://www.snia.org/sites/default/education/tutorials/2009/spring/solid/JonathanThatcher_NandFlash_SSS_PerformanceV10-nc.pdf

Recall: Flash Internal Structure





Samsung 980 Pro (met heatsink) 2TB

Snelheden	
Lezen (sequentieel)	7.000MB/s
Schrijven (sequentieel)	5.000MB/s
Lezen (random 4K, QD32)	1.000.000IOPS
Schrijven (random 4K, QD32)	1.000.000IOPS

We have these in our servers!

https://tweakers.net/pricewatch/1761712/samsung-980-pro-met-heatsink-2tb/specificaties/

, MLC)

0/100, 50/50, 75/25) MB/s

(50/50) MB/s

(25/75) MB/s

(0/100) MB/s

umes 8 die per bus

CS per bus

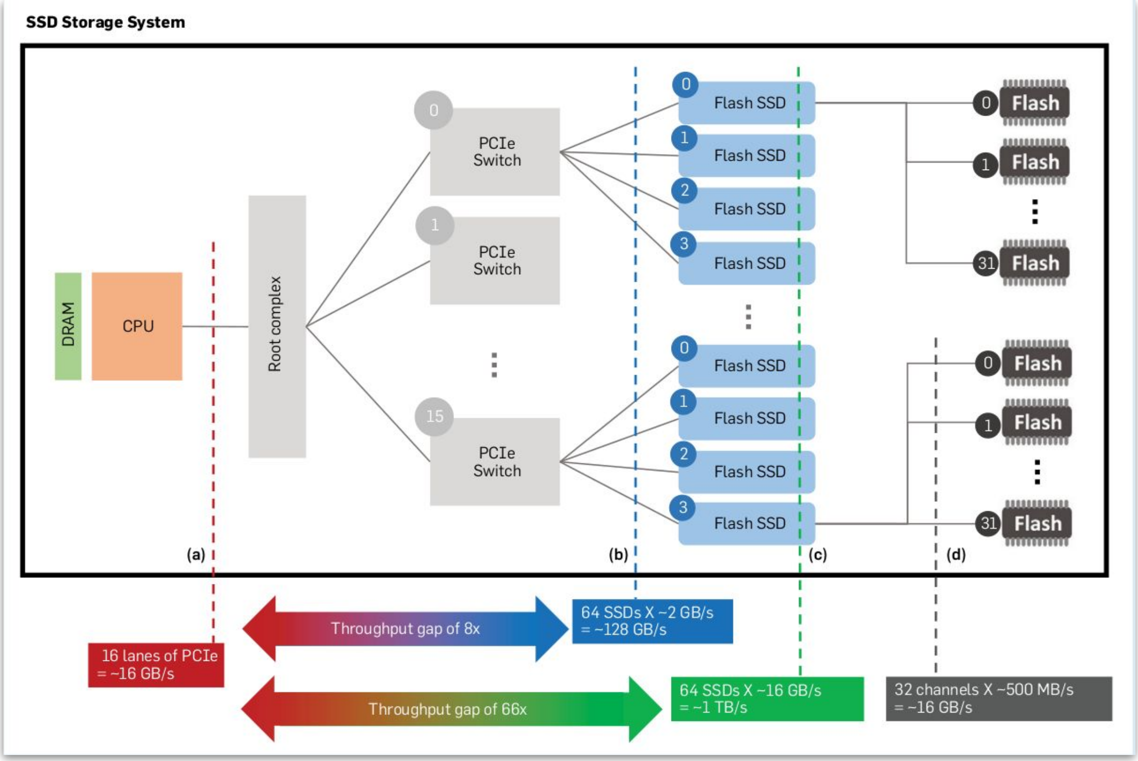
Flash devi

These con

A single se

https://www.snia.org/sites/default/education/tutorials/2009/spring/solid/JonathanThatcher_NandFlash_SSS_PerformanceV10-nc.pdf

Data Movement Bottlenecks Inside a Single System



A rack-level SSDs deployment

64 SSDs connected in a system

Internally each SSD can have 32 flash packages in parallel

At the green line you have 1TB/s

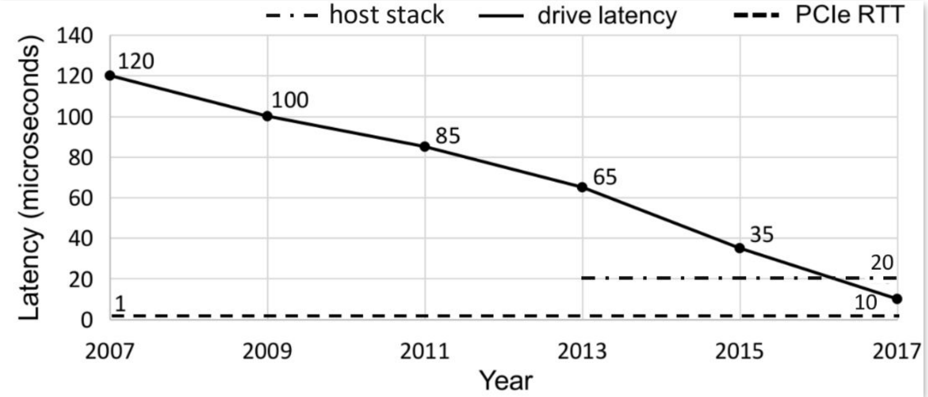
It drops to 128GB/s at the PCIe switches

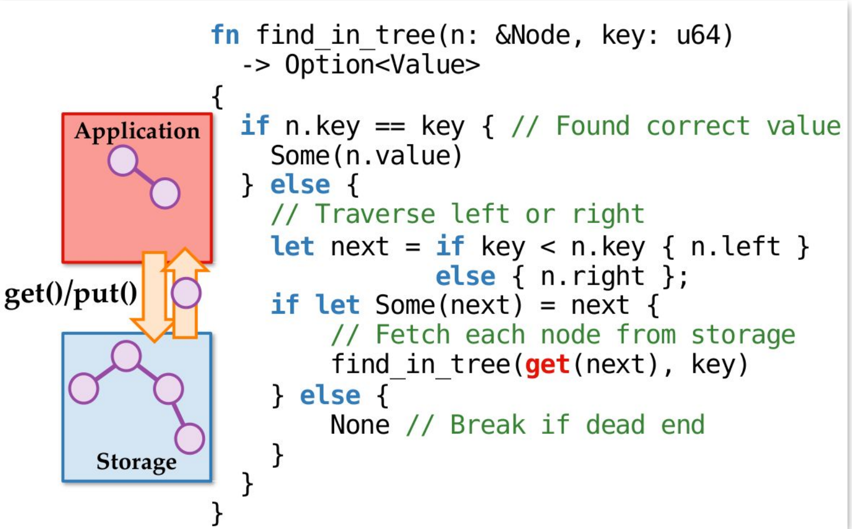
It further drops to 16 GB/s at the CPU

Yes, PCIe is improving, but not as fast!

Jaeyoung Do, Sudipta Sengupta, and Steven Swanson. 2019. Programmable solid-state storage in future cloud datacenters. *Commun. ACM* 62, 6 (June 2019), 54–62.

Latency Pressure





```
fn find_in_tree(n: &Node, key: u64)
  -> Option<Value>
{
  if n.key == key { // Found correct value
    Some(n.value)
  } else {
    // Traverse left or right
    let next = if key < n.key { n.left }
               else { n.right };
    if let Some(next) = next {
      // Fetch each node from storage
      find_in_tree(get(next), key)
    } else {
      None // Break if dead end
    }
  }
}
```

Kulkarni, Splinter: bare-metal extensions for multi-tenant low-latency storage, OSDI 2018.

Crossing PCIe bus (v3.0, v4.0) can take time ~1 useconds

Over the years the drive latencies have been improving

- See ULL and 3D-NAND flash
- Can do ~5 usec latencies

PCIe latency has become a bottleneck for pointer chasing, latency-sensitive applications

What about over an external network?

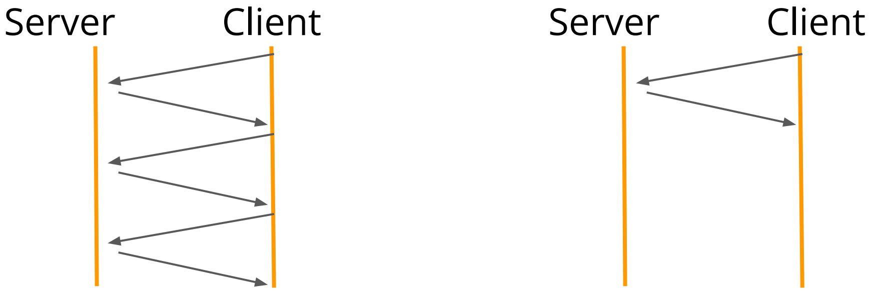
Over the Network?

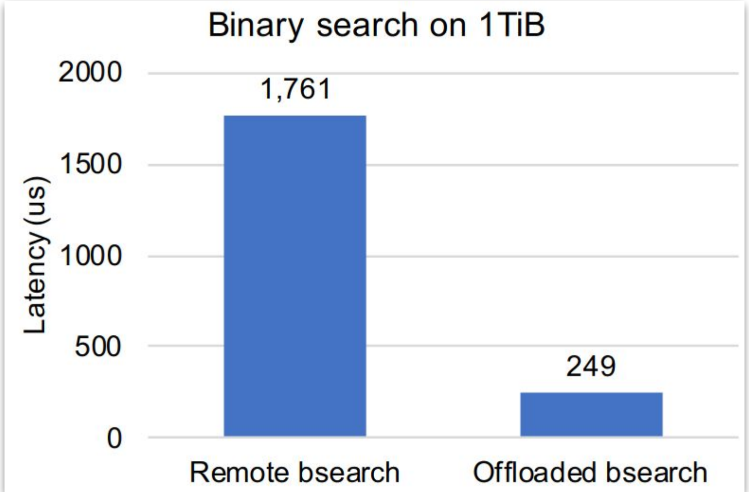
1 TB data with 8 bytes keys (2^{37} values), RTT of 40 usec (on 10 Gbps)

Remote bsearch: fetch each node on demand and pointer chasing left/right, ~37 round trips

Offloaded bsearch: send code to the remote, disaggregated storage server for execution, get the result, 1 round trip

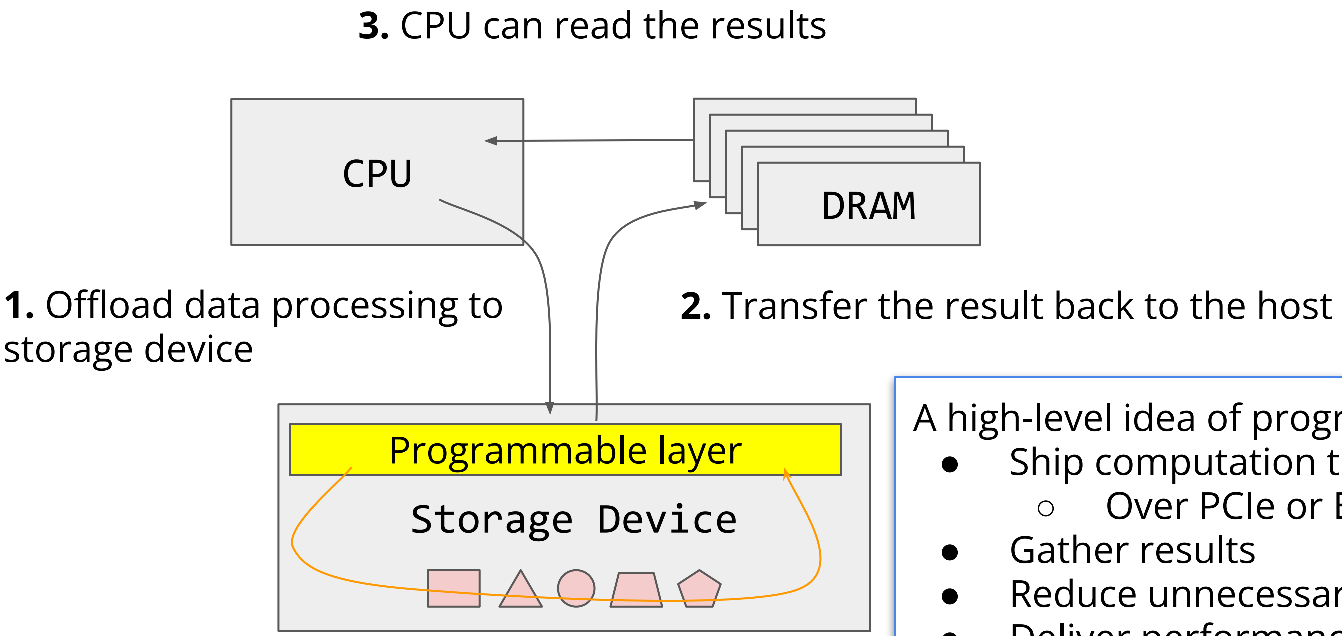
Shows up in performance difference





Kornilios Kourtis, Animesh Trivedi, Nikolas Ioannou, Safe and Efficient Remote Application Code Execution on Disaggregated NVM Storage with eBPF, https://arxiv.org/abs/2002.11528 (2020).

Enter: Programmable Storage



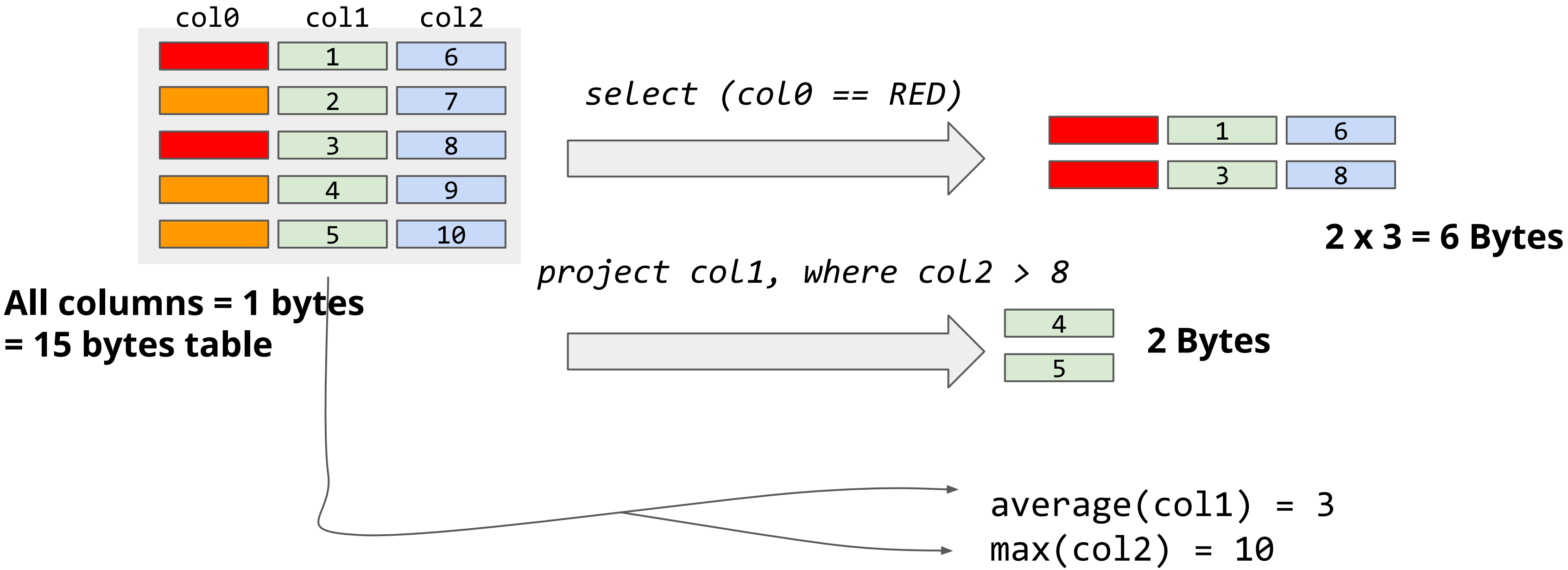
3. CPU can read the results
CPU
DRAM
1. Offload data processing to storage device
2. Transfer the result back to the host
Programmable layer
Storage Device

A high-level idea of programmable storage

- Ship computation to the storage device
    - Over PCIe or Ethernet
- Gather results
- Reduce unnecessary data movement
- Deliver performance, low latency operations
- Saves energy!

Why is Programmable Storage Useful?

1. Data processing is often **reductive** (not always!)
   a. grep, filter, aggregate → results are often smaller than the original data



col0	col1	col2
🟥	1	6
🟧	2	7
🟥	3	8
🟧	4	9
🟧	5	10

All columns = 1 bytes
= 15 bytes table

select (col0 == RED)

🟥	1	6
🟥	3	8

2 x 3 = 6 Bytes

project col1, where col2 > 8

4
5

2 Bytes

```
average(col1) = 3
max(col2) = 10
```

Why is Programmable Storage Useful?

1. Data processing is often reductive (not always!)
    a. grep, filter, aggregate → results are often smaller than the original data

2. SSDs already are complex
    a. FTL implementation, GC logic
    b. SSDs already have some “logic” implementation capabilities and spare cycles

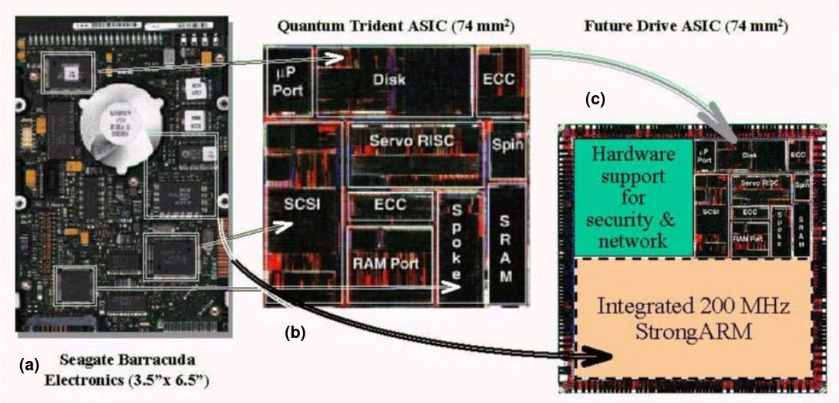
3. Additional support from the devices have been helpful
    a. Expose SSD internals to optimize for applications (SDF, OCSSDs, ZNS)
    b. Flash virtualization (DFS file system)
    c. Further capabilities: caching, atomic updates and appends, transactions, KV-SSDs

Why not make programmable SSDs a standard feature where a user can offload computation to the SSD? (if yes, then how can we do it?)

The Idea Itself is Not New ...

The idea itself is not now (as with many ideas in Computer Science)

- Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. 1998. A case for intelligent disks (IDISKs). SIGMOD Rec. 27, 3 (Sept. 1, **1998**), 42–52.
- Erik Riedel, Garth A. Gibson, and Christos Faloutsos. 1998. Active Storage for Large-Scale Data Mining and Multimedia. In Proceedings of the 24th International Conference on Very Large Data Bases (VLDB '**1998**).
- And many more ...


Quantum Trident ASIC (74 mm²)
Future Drive ASIC (74 mm²)
µP Port
Disk
ECC
Servo RISC
Spin
SCSI
ECC
Spoke
SRAM
RAM Port
(a)
Seagate Barracuda Electronics (3.5"x 6.5")
(b)
(c)
Hardware support for security & network
Integrated 200 MHz StrongARM

Corne Lukken, Animesh Trivedi:
Past, Present and Future of Computational Storage: A Survey. CoRR abs/2112.09691 (2021), https://arxiv.org/abs/2112.09691

The Idea Itself is Not New ...

The idea itself is not now (as with many ideas in Computer Science)

- Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. 1998. A case for intelligent disks (IDISKs). SIGMOD Rec. 27, 3 (Sept. 1, **1998**), 42–52.
- Erik Riedel, Garth A. Gibson, and Christos Faloutsos. 1998. Active Storage for Large-Scale Data Mining and Multimedia. In Proceedings of the 24th International Conference on Very Large Data Bases (VLDB '**1998**).
- And many more ...

However, they did not become popular because

1. Too expensive technology
2. Gains from such disk-based setup were low. Disk performance was bottleneck, and
   a. host/drive/link speeds were improving
   b. DRAM caching size was getting bigger too

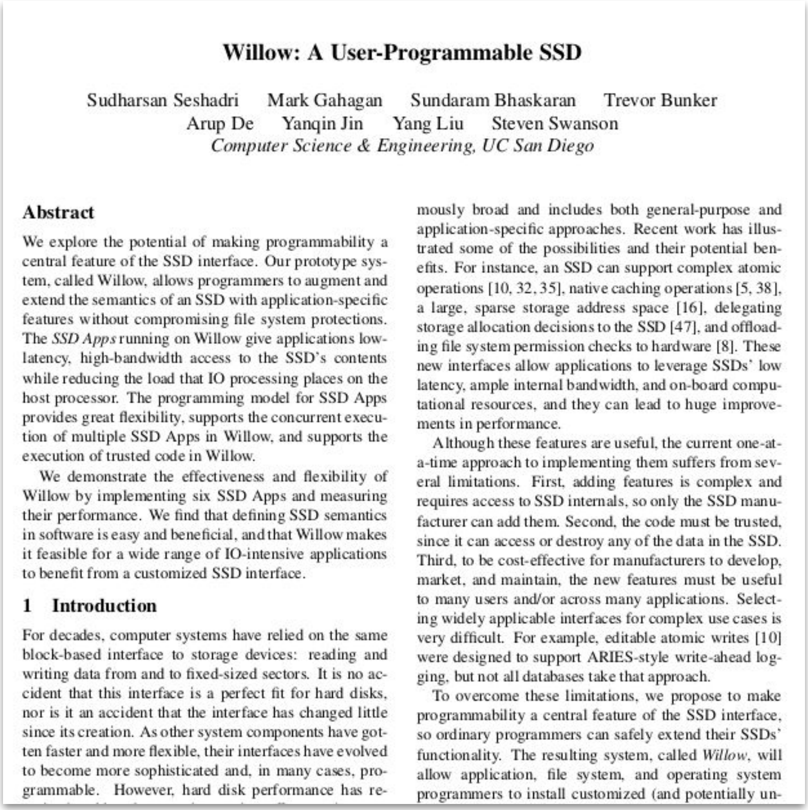
What are the Challenges in Programmable Storage?

1. **How to provide programmability?**
   a. In the hardware or software, or some combination of these?
   b. ASIC, embedded CPUs, FGPA, languages, toolchain

2. **What is the programming API?**
   a. What is a useful programing abstraction to perform any computation
   b. How do you transfer computation logic to a remote end point (storage)
   c. Integrate other known storage abstractions: files, key-value stores, etc.

3. **How do you provide?**
   a. Multi-tenancy
   b. Quality of service, isolation
   c. Security and privacy

Willow: A User-Programmable SSD (2014)



Willow: A User-Programmable SSD

Sudharsan Seshadri Mark Gahagan Sundaram Bhaskaran Trevor Bunker
Arup De Yanqin Jin Yang Liu Steven Swanson
Computer Science & Engineering, UC San Diego

Abstract

We explore the potential of making programmability a central feature of the SSD interface. Our prototype system, called Willow, allows programmers to augment and extend the semantics of an SSD with application-specific features without compromising file system protections. The *SSD Apps* running on Willow give applications low-latency, high-bandwidth access to the SSD's contents while reducing the load that IO processing places on the host processor. The programming model for SSD Apps provides great flexibility, supports the concurrent execution of multiple SSD Apps in Willow, and supports the execution of trusted code in Willow.

We demonstrate the effectiveness and flexibility of Willow by implementing six SSD Apps and measuring their performance. We find that defining SSD semantics in software is easy and beneficial, and that Willow makes it feasible for a wide range of IO-intensive applications to benefit from a customized SSD interface.

1 Introduction

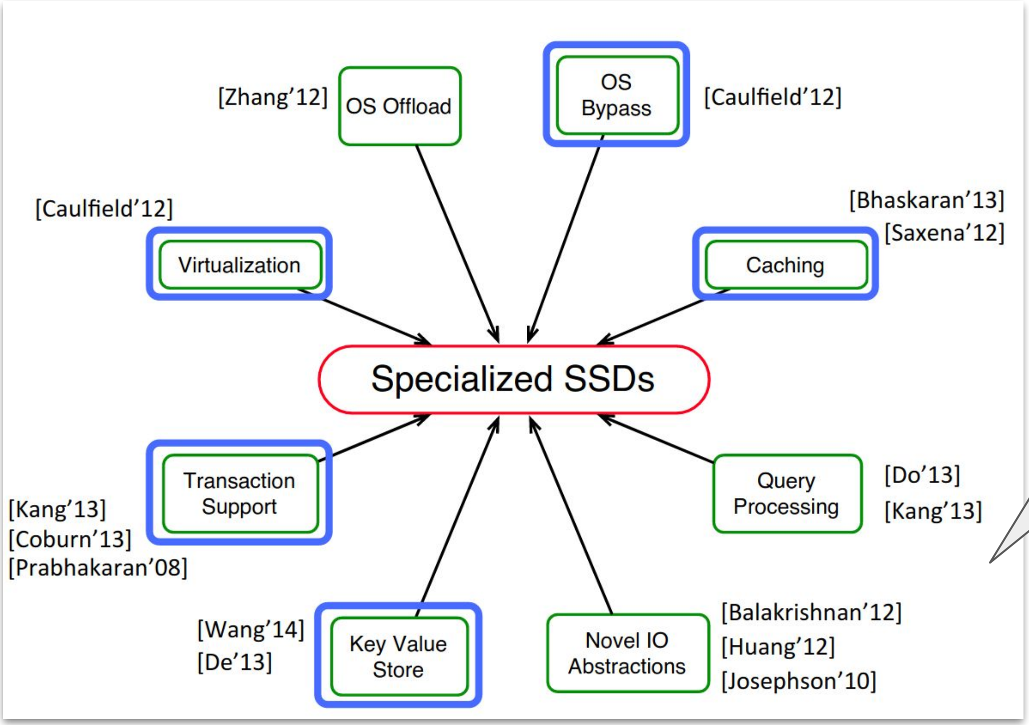
For decades, computer systems have relied on the same block-based interface to storage devices: reading and writing data from and to fixed-sized sectors. It is no accident that this interface is a perfect fit for hard disks, nor is it an accident that the interface has changed little since its creation. As other system components have gotten faster and more flexible, their interfaces have evolved to become more sophisticated and, in many cases, programmable. However, hard disk performance has re-

mously broad and includes both general-purpose and application-specific approaches. Recent work has illustrated some of the possibilities and their potential benefits. For instance, an SSD can support complex atomic operations [10, 32, 35], native caching operations [5, 38], a large, sparse storage address space [16], delegating storage allocation decisions to the SSD [47], and offloading file system permission checks to hardware [8]. These new interfaces allow applications to leverage SSDs' low latency, ample internal bandwidth, and on-board computational resources, and they can lead to huge improvements in performance.

Although these features are useful, the current one-at-a-time approach to implementing them suffers from several limitations. First, adding features is complex and requires access to SSD internals, so only the SSD manufacturer can add them. Second, the code must be trusted, since it can access or destroy any of the data in the SSD. Third, to be cost-effective for manufacturers to develop, market, and maintain, the new features must be useful to many users and/or across many applications. Selecting widely applicable interfaces for complex use cases is very difficult. For example, editable atomic writes [10] were designed to support ARIES-style write-ahead logging, but not all databases take that approach.

To overcome these limitations, we propose to make programmability a central feature of the SSD interface, so ordinary programmers can safely extend their SSDs' functionality. The resulting system, called *Willow*, will allow application, file system, and operating system programmers to install customized (and potentially un-

Key Challenge



How to build a usable abstraction to build these multiple ***specialized*** *applications?*

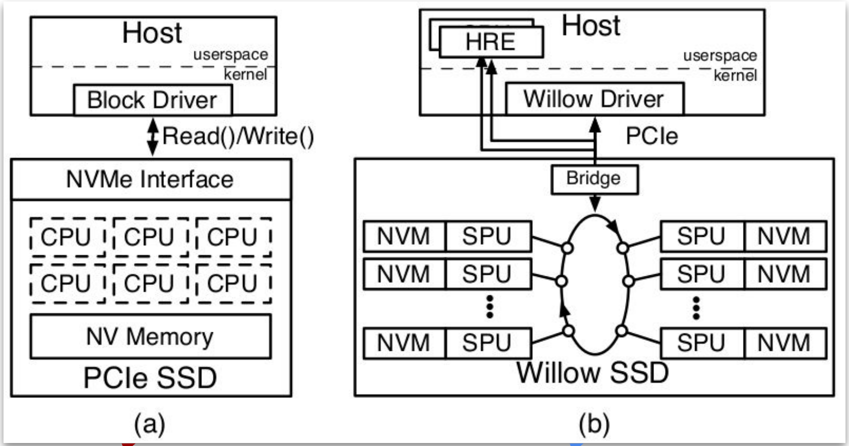
Why specalization?

https://www.usenix.org/sites/default/files/conference/protected-files/osdi14_slides_seshadri.pdf

Willow Architecture

Conventional SSDs (figure (a)), Willow (figure (b))

- Contains **Storage Processor Unit (SPUs)**
  - that process requests for their attached NVM storage



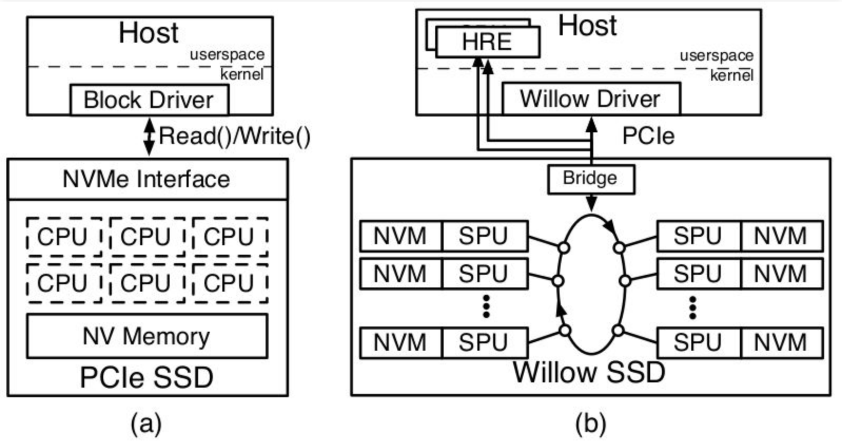
Normal NVMe SSD

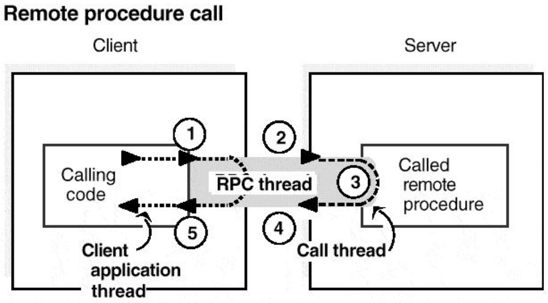
Willow SSD (uses own PCIe protocol)

Willow Architecture

Conventional SSDs (figure (a)), Willow (figure (b))

- Contains **Storage Processor Unit (SPUs)**
  - that process requests for their attached NVM storage
- The host does not do conventional r/w but uses **Host RPC Endpoints (HREs)**
  - **Why RPCs?** The most flexible way of establishing a command/response protocol
  - HREs communicate with SPUs
  - What to communicate, how to communicate - the application/user decide





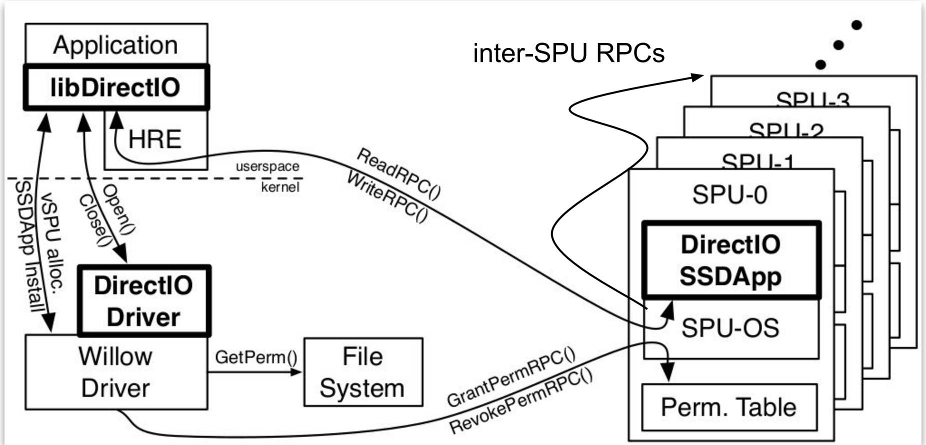
So how do these HREs, SPUs work together to offer a programmable SSD?

https://networkencyclopedia.com/remote-procedure-call-rpc/

Willow: An SSD-Application View

Each **SSD application**

1. Provides RPC handlers to the Willow driver to be installed in SSD
2. A user-space library to access SSD directly
3. [optional] Kernel module to get support for kernel routines - filesystem



Here in the figure (example): Design for a Direct-Access Storage

1. Ask the Willow driver to install direct-IO RPC handlers and request an Host RPC Endpoint (**HRE**)
2. At the open of a file for direct I/O, the application asks the kernel driver to check file permissions and install them in the SSD
3. Do a direct read/write using RPCs from HREs to SPUs

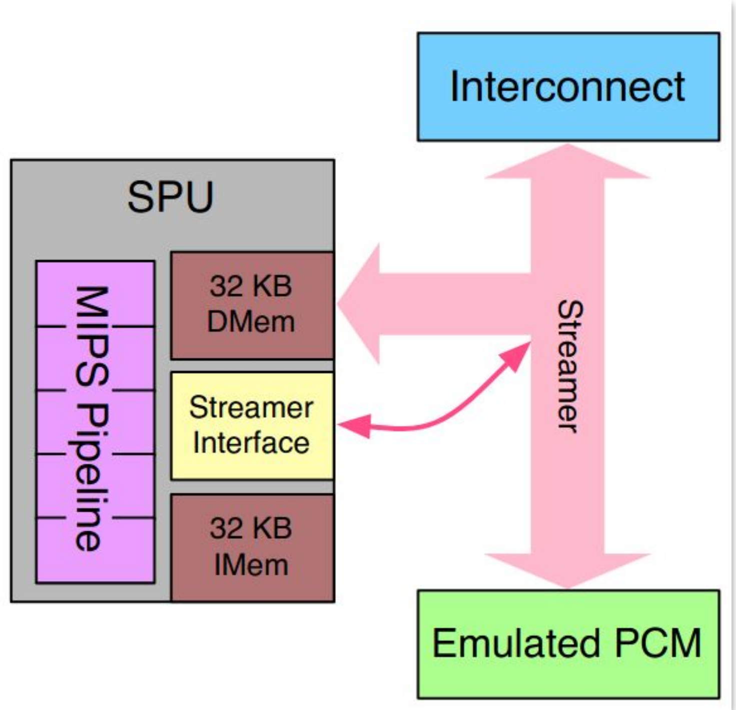
The customization with kernel module, user library and SPU RPC handler → Programming !

What is Inside SPUs?

- 125 MHz MIPS processor
- 32 KB of Data and Instruction Memory
- Connected to a bank of NVM (here: PCM)
- Network interface (PCIe)

The SPU runs a simple operating system (SPU-OS)

- Gives simple multi-threading
- Memory is managed by the host driver
    - Statically allocated



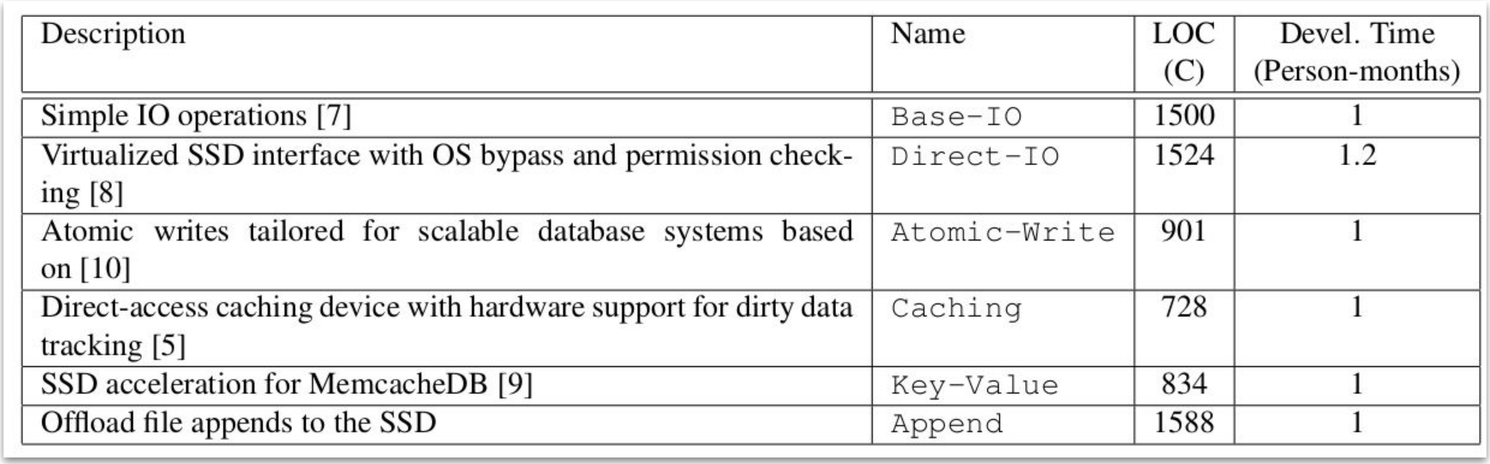
Protection and Sharing Features

1. How to track which user application is executing code on a shared SSDs?
   a. Each HRE has an id which is always propagated with all RPC request and responses to keep track of which process is responsible for computation

2. How to check if an SSD-Application has rights to modify and update data?
   a. Each application has permissions associated with the HRE and data touched
   b. In case not all permissions can be stored inside the SSD, a permission miss will happen and the SPU will contact the kernel model to get updated permissions

3. Code and data protection inside SPU
   a. Use SPU’s memory segmentation support (segmentation registers)

Code Complexity

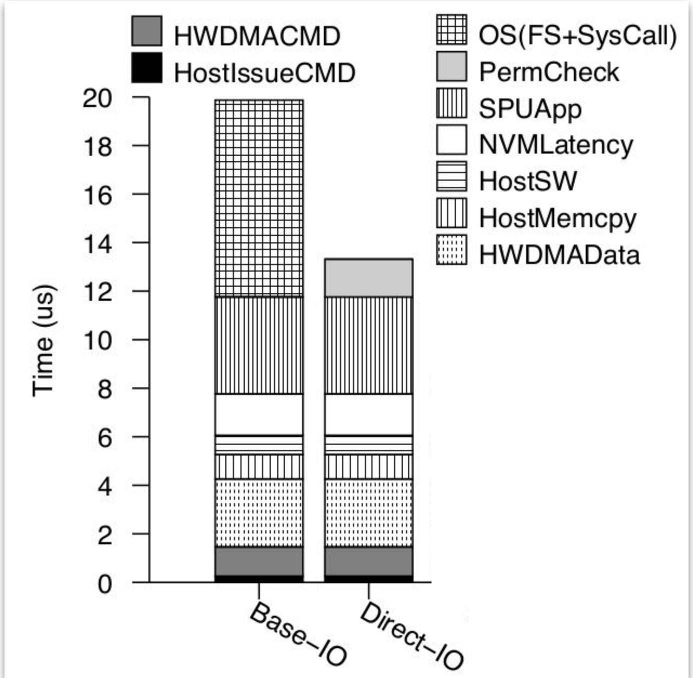


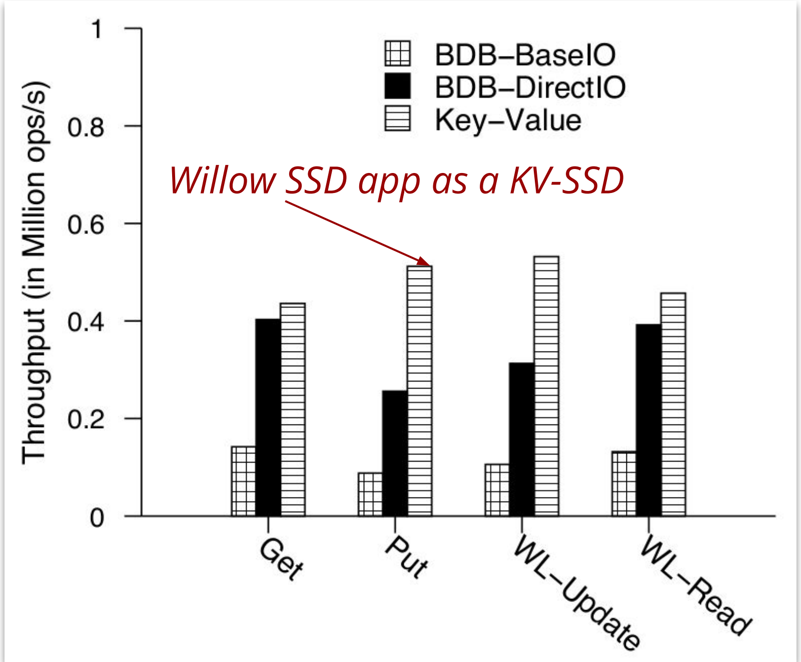
Description	Name	LOC (C)	Devel. Time (Person-months)
Simple IO operations [7]	`Base-IO`	1500	1
Virtualized SSD interface with OS bypass and permission checking [8]	`Direct-IO`	1524	1.2
Atomic writes tailored for scalable database systems based on [10]	`Atomic-Write`	901	1
Direct-access caching device with hardware support for dirty data tracking [5]	`Caching`	728	1
SSD acceleration for MemcacheDB [9]	`Key-Value`	834	1
Offload file appends to the SSD	`Append`	1588	1

Many ideas only take a 100s of lines of code to implement in Willow

4-6 weeks of development time (reasonable)

Performance

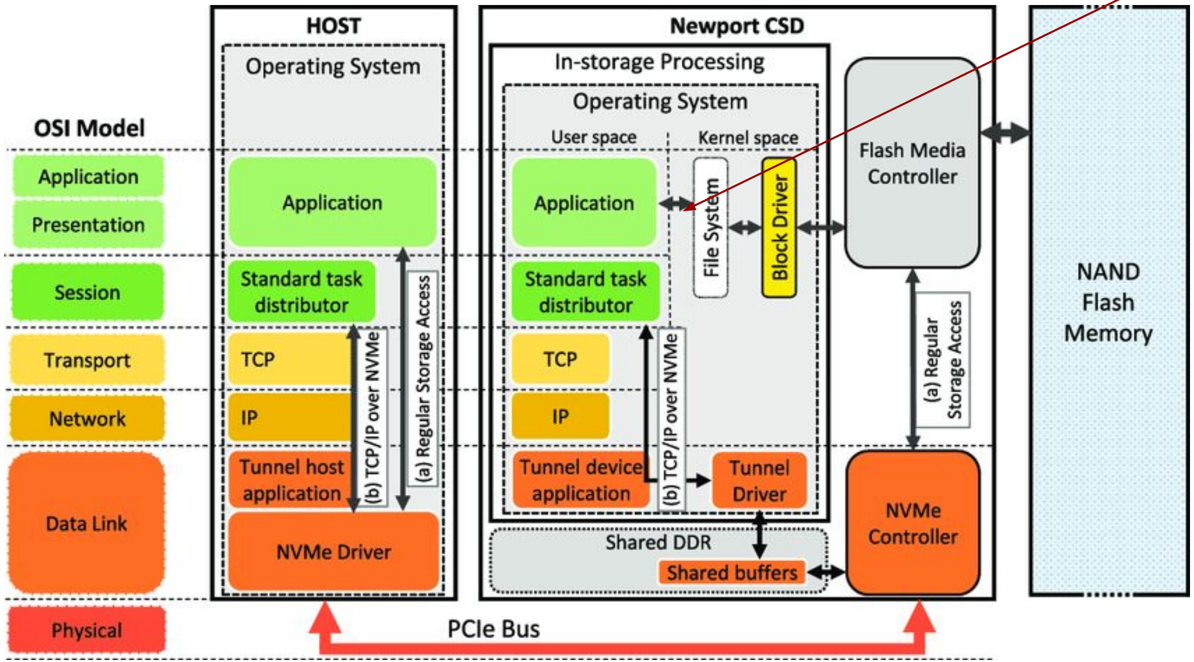




Willow SSD app as a KV-SSD

- Direct I/O helps to reduce FS + syscall overheads
- Key-value on Willow (RPC) can improve performance from 8% - 4.8x

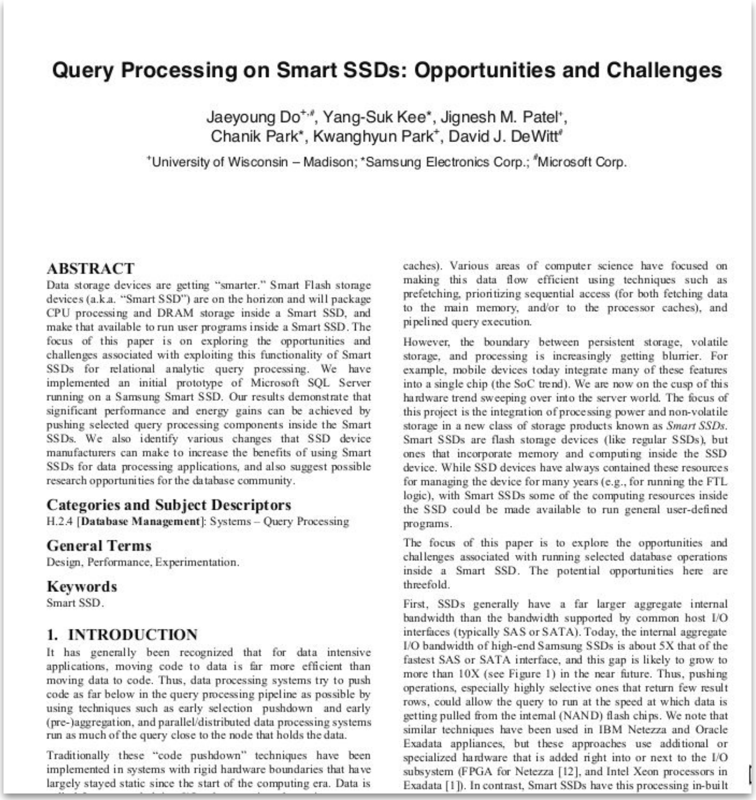
This RPC-based Design - Flexibility



Local SSD as an Network-attached Server with RPC (over PCIe)

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 (November 2020), 37 pages. https://doi.org/10.1145/3415580

Relational Data Processing Frameworks (why?)



Query Processing on Smart SSDs: Opportunities and Challenges

Jaeyoung Do[+,#], Yang-Suk Kee[*], Jignesh M. Patel[+],
Chanik Park[*], Kwanghyun Park[+], David J. DeWitt[#]
[+]University of Wisconsin – Madison; [*]Samsung Electronics Corp.; [#]Microsoft Corp.

ABSTRACT

Data storage devices are getting "smarter." Smart Flash storage devices (a.k.a. "Smart SSD") are on the horizon and will package CPU processing and DRAM storage inside a Smart SSD, and make that available to run user programs inside a Smart SSD. The focus of this paper is on exploring the opportunities and challenges associated with exploiting this functionality of Smart SSDs for relational analytic query processing. We have implemented an initial prototype of Microsoft SQL Server running on a Samsung Smart SSD. Our results demonstrate that significant performance and energy gains can be achieved by pushing selected query processing components inside the Smart SSDs. We also identify various changes that SSD device manufacturers can make to increase the benefits of using Smart SSDs for data processing applications, and also suggest possible research opportunities for the database community.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems – Query Processing

General Terms

Design, Performance, Experimentation.

Keywords

Smart SSD.

1. INTRODUCTION

It has generally been recognized that for data intensive applications, moving code to data is far more efficient than moving data to code. Thus, data processing systems try to push code as far below in the query processing pipeline as possible by using techniques such as early selection pushdown and early (pre-)aggregation, and parallel/distributed data processing systems run as much of the query close to the node that holds the data.

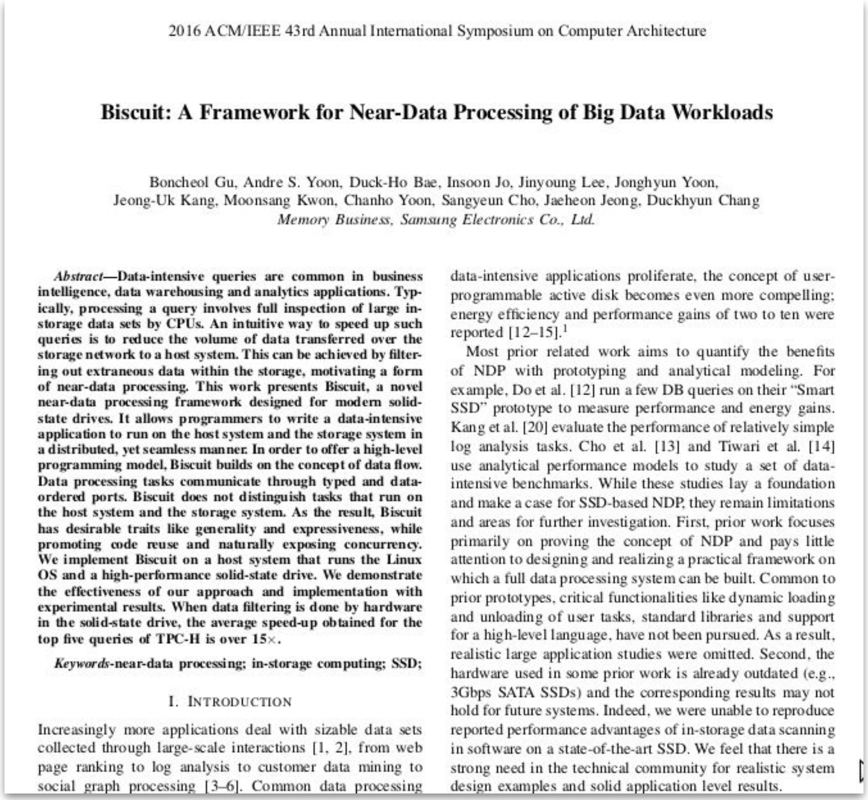
Traditionally these "code pushdown" techniques have been implemented in systems with rigid hardware boundaries that have largely stayed static since the start of the computing era. Data is

caches). Various areas of computer science have focused on making this data flow efficient using techniques such as prefetching, prioritizing sequential access (for both fetching data to the main memory, and/or to the processor caches), and pipelined query execution.

However, the boundary between persistent storage, volatile storage, and processing is increasingly getting blurrier. For example, mobile devices today integrate many of these features into a single chip (the SoC trend). We are now on the cusp of this hardware trend sweeping over into the server world. The focus of this project is the integration of processing power and non-volatile storage in a new class of storage products known as *Smart SSDs*. Smart SSDs are flash storage devices (like regular SSDs), but ones that incorporate memory and computing inside the SSD device. While SSD devices have always contained these resources for managing the device for many years (e.g., for running the FTL logic), with Smart SSDs some of the computing resources inside the SSD could be made available to run general user-defined programs.

The focus of this paper is to explore the opportunities and challenges associated with running selected database operations inside a Smart SSD. The potential opportunities here are threefold.

First, SSDs generally have a far larger aggregate internal bandwidth than the bandwidth supported by common host I/O interfaces (typically SAS or SATA). Today, the internal aggregate I/O bandwidth of high-end Samsung SSDs is about 5X that of the fastest SAS or SATA interface, and this gap is likely to grow to more than 10X (see Figure 1) in the near future. Thus, pushing operations, especially highly selective ones that return few result rows, could allow the query to run at the speed at which data is getting pulled from the internal (NAND) flash chips. We note that similar techniques have been used in IBM Netezza and Oracle Exadata appliances, but these approaches use additional or specialized hardware that is added right into or next to the I/O subsystem (FPGA for Netezza [12], and Intel Xeon processors in Exadata [1]). In contrast, Smart SSDs have this processing in-built



2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture

Biscuit: A Framework for Near-Data Processing of Big Data Workloads

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, Duckhyun Chang
Memory Business, Samsung Electronics Co., Ltd.

***Abstract*—Data-intensive queries are common in business intelligence, data warehousing and analytics applications. Typically, processing a query involves full inspection of large in-storage data sets by CPUs. An intuitive way to speed up such queries is to reduce the volume of data transferred over the storage network to a host system. This can be achieved by filtering out extraneous data within the storage, motivating a form of near-data processing. This work presents Biscuit, a novel near-data processing framework designed for modern solid-state drives. It allows programmers to write a data-intensive application to run on the host system and the storage system in a distributed, yet seamless manner. In order to offer a high-level programming model, Biscuit builds on the concept of data flow. Data processing tasks communicate through typed and data-ordered ports. Biscuit does not distinguish tasks that run on the host system and the storage system. As the result, Biscuit has desirable traits like generality and expressiveness, while promoting code reuse and naturally exposing concurrency. We implement Biscuit on a host system that runs the Linux OS and a high-performance solid-state drive. We demonstrate the effectiveness of our approach and implementation with experimental results. When data filtering is done by hardware in the solid-state drive, the average speed-up obtained for the top five queries of TPC-H is over 15×.**

***Keywords*-near-data processing; in-storage computing; SSD;**

I. Introduction

Increasingly more applications deal with sizable data sets collected through large-scale interactions [1, 2], from web page ranking to log analysis to customer data mining to social graph processing [3–6]. Common data processing

data-intensive applications proliferate, the concept of user-programmable active disk becomes even more compelling; energy efficiency and performance gains of two to ten were reported [12–15].[1]

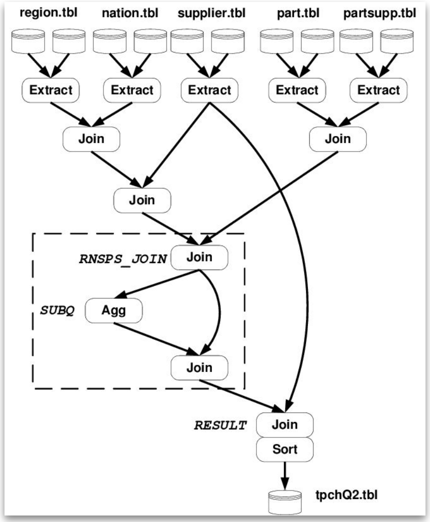
Most prior related work aims to quantify the benefits of NDP with prototyping and analytical modeling. For example, Do et al. [12] run a few DB queries on their "Smart SSD" prototype to measure performance and energy gains. Kang et al. [20] evaluate the performance of relatively simple log analysis tasks. Cho et al. [13] and Tiwari et al. [14] use analytical performance models to study a set of data-intensive benchmarks. While these studies lay a foundation and make a case for SSD-based NDP, they remain limitations and areas for further investigation. First, prior work focuses primarily on proving the concept of NDP and pays little attention to designing and realizing a practical framework on which a full data processing system can be built. Common to prior prototypes, critical functionalities like dynamic loading and unloading of user tasks, standard libraries and support for a high-level language, have not been pursued. As a result, realistic large application studies were omitted. Second, the hardware used in some prior work is already outdated (e.g., 3Gbps SATA SSDs) and the corresponding results may not hold for future systems. Indeed, we were unable to reproduce reported performance advantages of in-storage data scanning in software on a state-of-the-art SSD. We feel that there is a strong need in the technical community for realistic system design examples and solid application level results.

Query Processing on Smart SSDs

One of the earliest attempt to revisit the idea of programmable storage for relational query processing

Advantages with relational query processing

- Structured operators and query plans
- Defined I/O access patterns
- Opportunities for “code-pushdown”, early filtering, selection, and aggregation



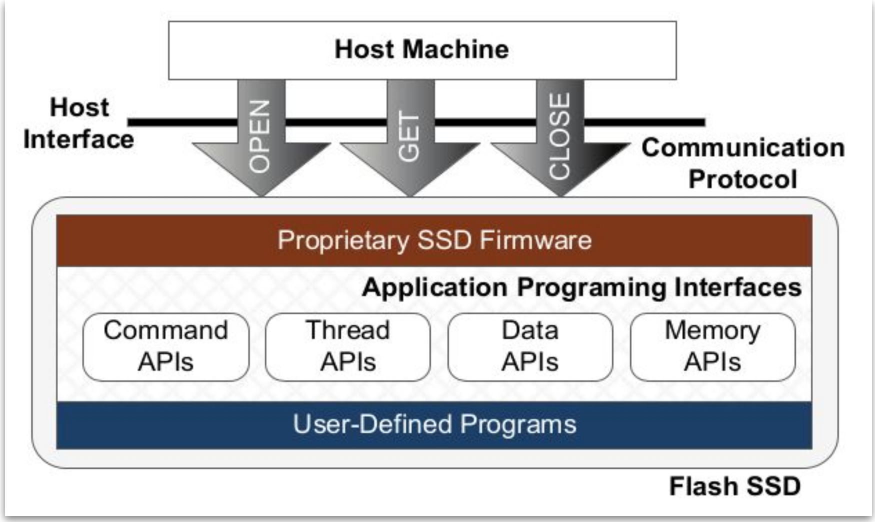
Proposed: implemented the simple selection and aggregation operators into the device FTL and integrated with SQL Server query plans

Architecture

1. Open and close to maintain session
2. Get to get results

User defined program is executed on an event (open, close) or arrival of a data page from flash

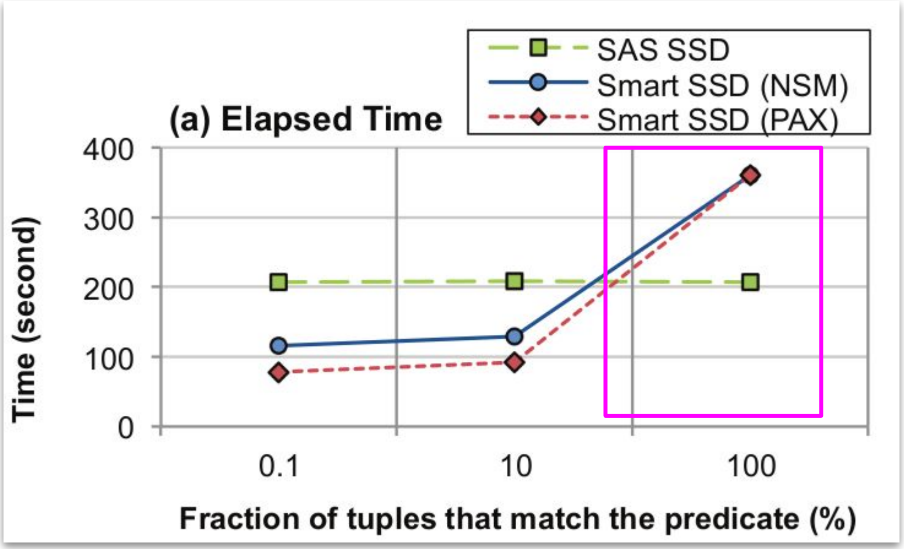
- Data pages can be staged in parallel



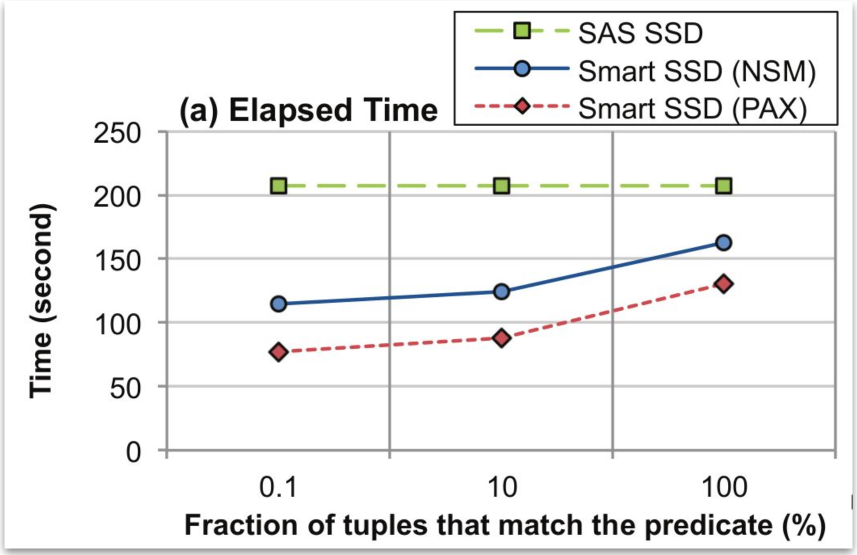
Basic thread scheduling (a master and worker threads), and memory management (static, per-thread)

→ **Focus on a single workload, no multi tenancy, no file system here!**

Performance

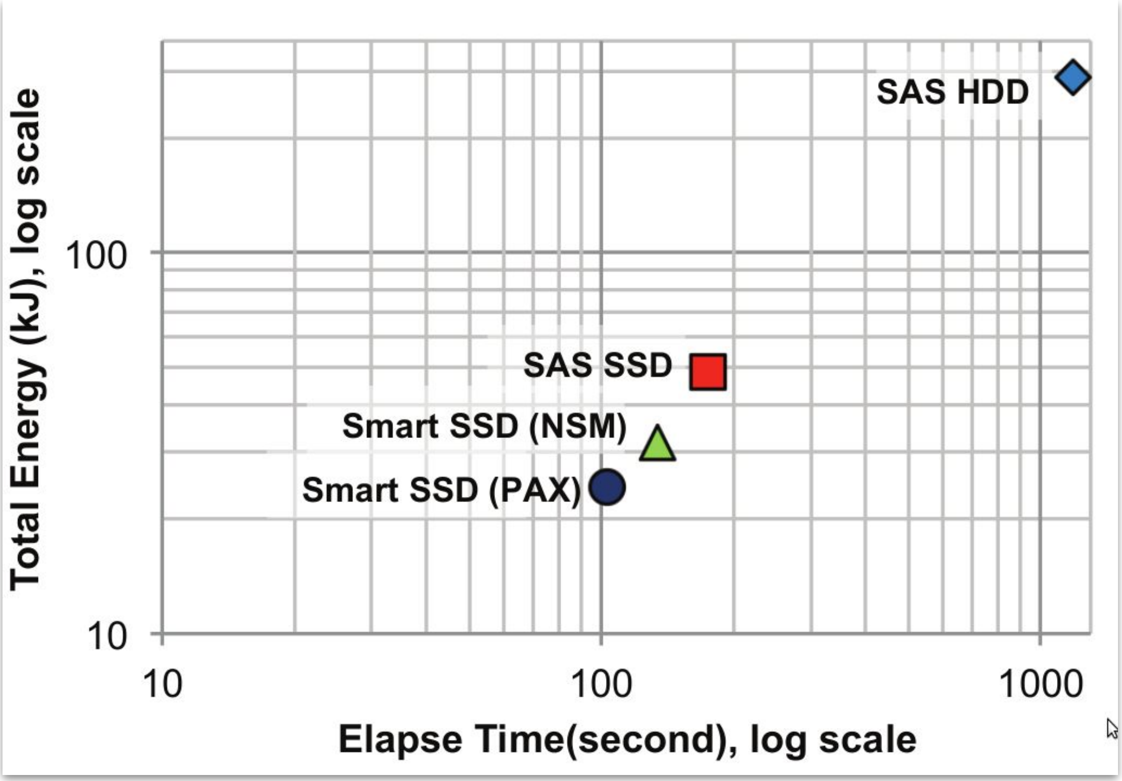


```
SELECT SecondColumn
FROM SyntheticTable
WHERE FirstColumn < [VALUE]
```



```
SELECT AVG (SecondColumn)
FROM SyntheticTable
WHERE FirstColumn < [VALUE]
```

Energy Efficiency



Compared to HDDs, SSDs are more energy efficient

Smart SSDs further allow faster, more energy efficient execution

N-ary Storage Model (NSM) and Partition Attributes Across (PAX) data layouts - how data is stored on the device

Biscuit: A Framework for Near-Data Processing of Big Data Workloads

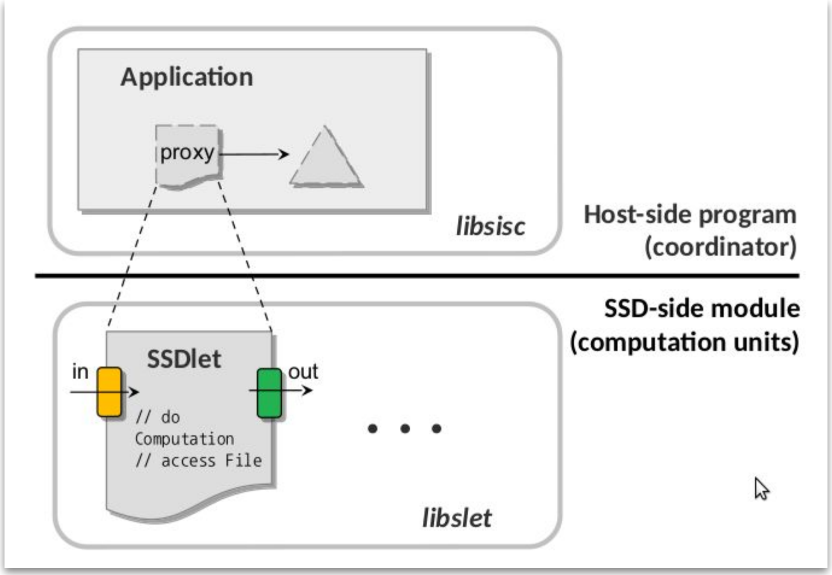
Flow-based programming model : build a graph of computation steps (very much like SQL DAGs)

Support (almost) full C++ 11/14 semantics

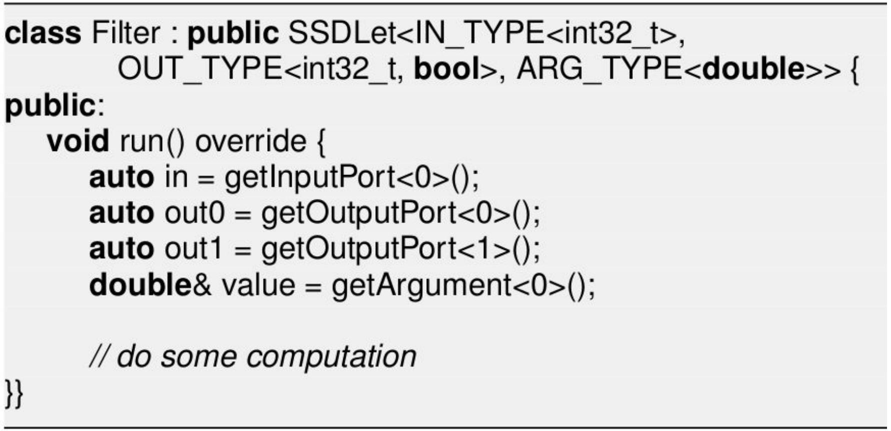
Split coordination and computation models

A typical application

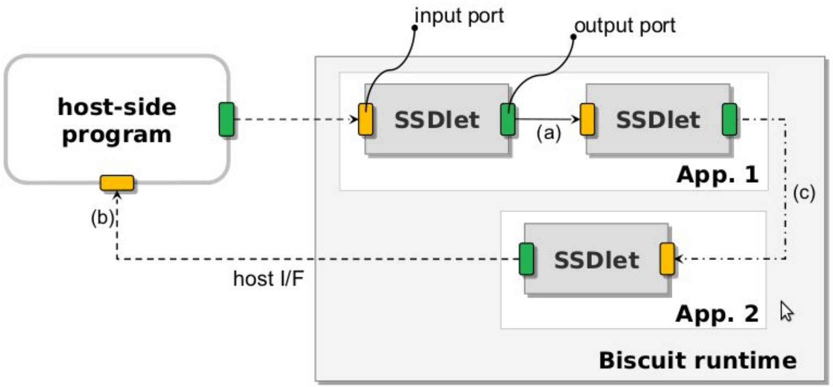
- Host side : libsisc
- SSD side : libslet,
  with IN/OUT coordination



SSDlets and Applications



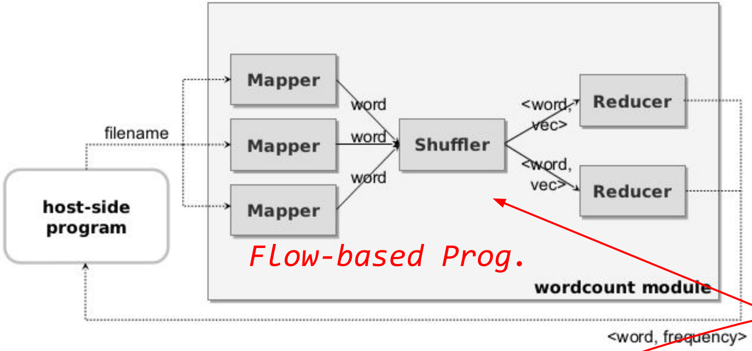
```
class Filter : public SSDLet<IN_TYPE<int32_t>,
           OUT_TYPE<int32_t, bool>, ARG_TYPE<double>> {
public:
    void run() override {
        auto in = getInputPort<0>();
        auto out0 = getOutputPort<0>();
        auto out1 = getOutputPort<1>();
        double& value = getArgument<0>();

        // do some computation
}}
```

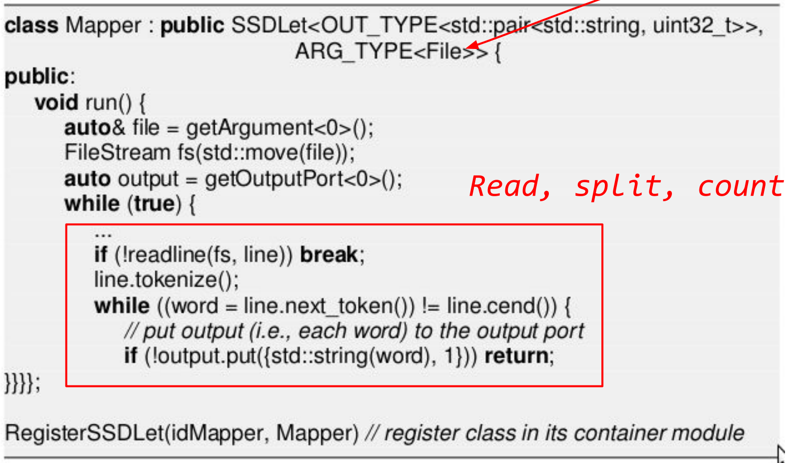


a. Inter-SSDlet (same application)
b. Host-device ports
c. Inter application ports

Important for coordination and staging of data

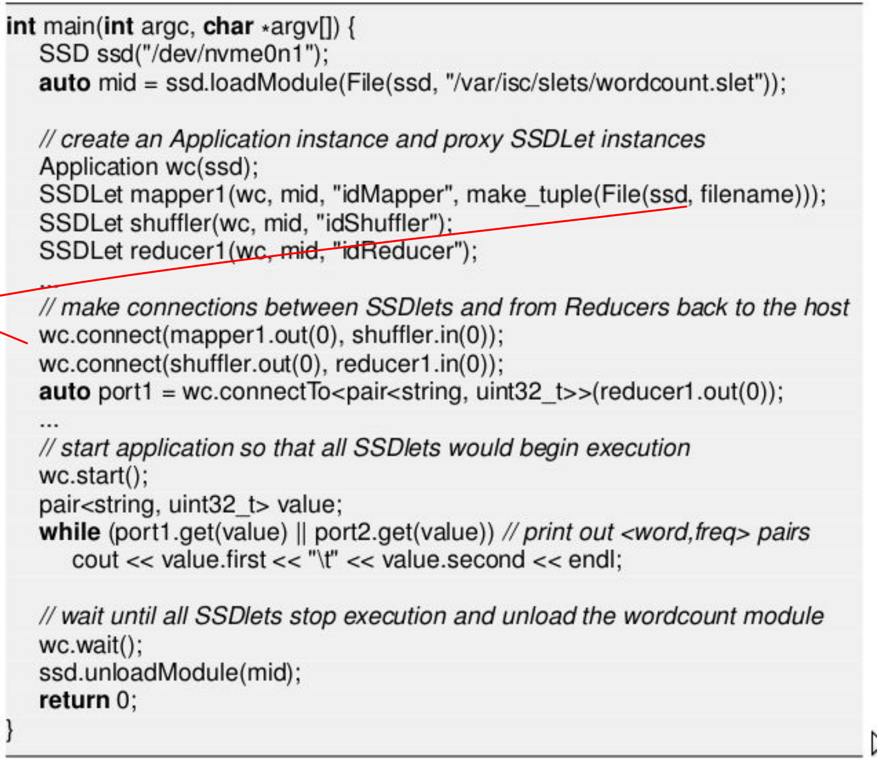
Word Count Application





```
class Mapper : public SSDLet<OUT_TYPE<std::pair<std::string, uint32_t>>,
                             ARG_TYPE<File>> {
public:
    void run() {
        auto& file = getArgument<0>();
        FileStream fs(std::move(file));
        auto output = getOutputPort<0>();
        while (true) {
            ...
            if (!readline(fs, line)) break;
            line.tokenize();
            while ((word = line.next_token()) != line.cend()) {
                // put output (i.e., each word) to the output port
                if (!output.put({std::string(word), 1})) return;
}}}};

RegisterSSDLet(idMapper, Mapper) // register class in its container module
```

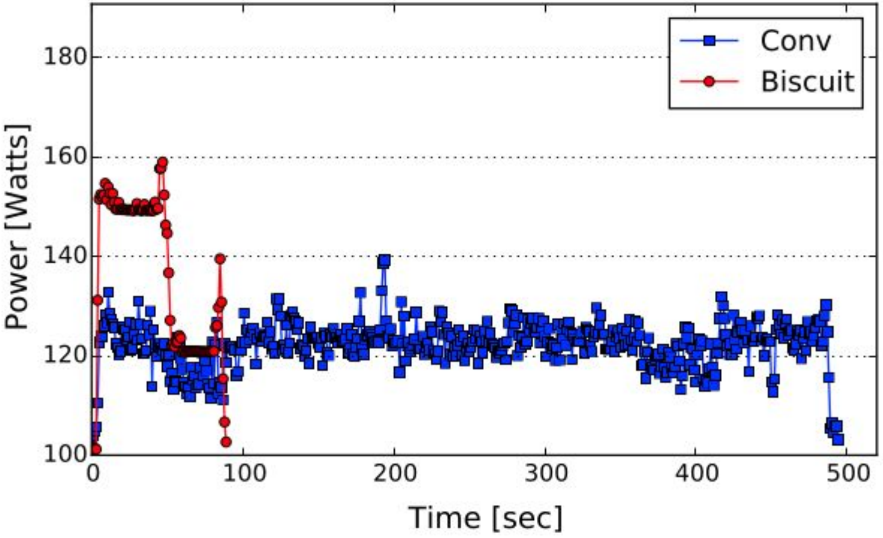
Read, split, count



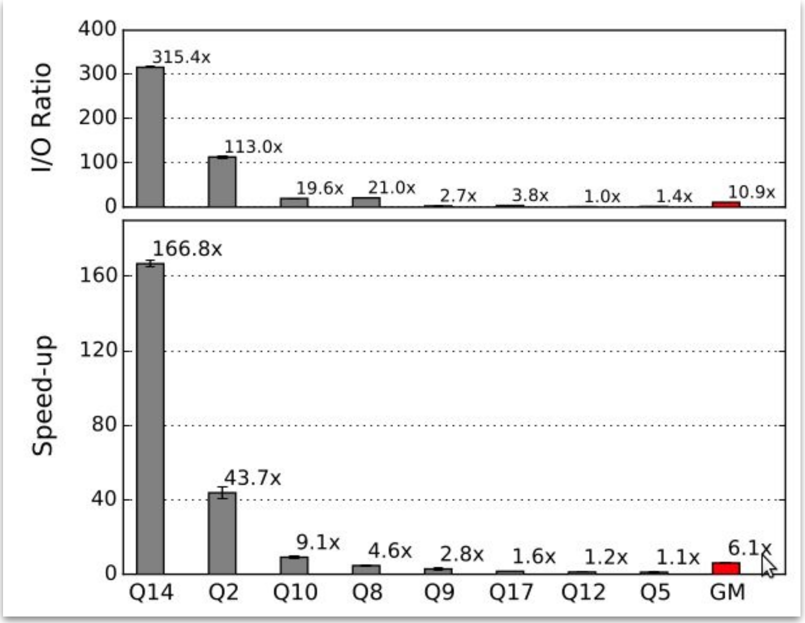
```
int main(int argc, char *argv[]) {
    SSD ssd("/dev/nvme0n1");
    auto mid = ssd.loadModule(File(ssd, "/var/isc/slets/wordcount.slet"));

    // create an Application instance and proxy SSDLet instances
    Application wc(ssd);
    SSDLet mapper1(wc, mid, "idMapper", make_tuple(File(ssd, filename)));
    SSDLet shuffler(wc, mid, "idShuffler");
    SSDLet reducer1(wc, mid, "idReducer");
    ...
    // make connections between SSDlets and from Reducers back to the host
    wc.connect(mapper1.out(0), shuffler.in(0));
    wc.connect(shuffler.out(0), reducer1.in(0));
    auto port1 = wc.connectTo<pair<string, uint32_t>>(reducer1.out(0));
    ...
    // start application so that all SSDlets would begin execution
    wc.start();
    pair<string, uint32_t> value;
    while (port1.get(value) || port2.get(value)) // print out <word,freq> pairs
        cout << value.first << "\t" << value.second << endl;

    // wait until all SSDlets stop execution and unload the wordcount module
    wc.wait();
    ssd.unloadModule(mid);
    return 0;
}
```

Performance

SSD Prototype has: Two ARM Cortex R7 cores @750MHz, L1$, no cache coherence, and Key-based pattern matcher per channel (filtering)



TPC-H Q1, base system energy 103 Watts



In Summary

Fast NVMs put pressure on network/link and performance demands

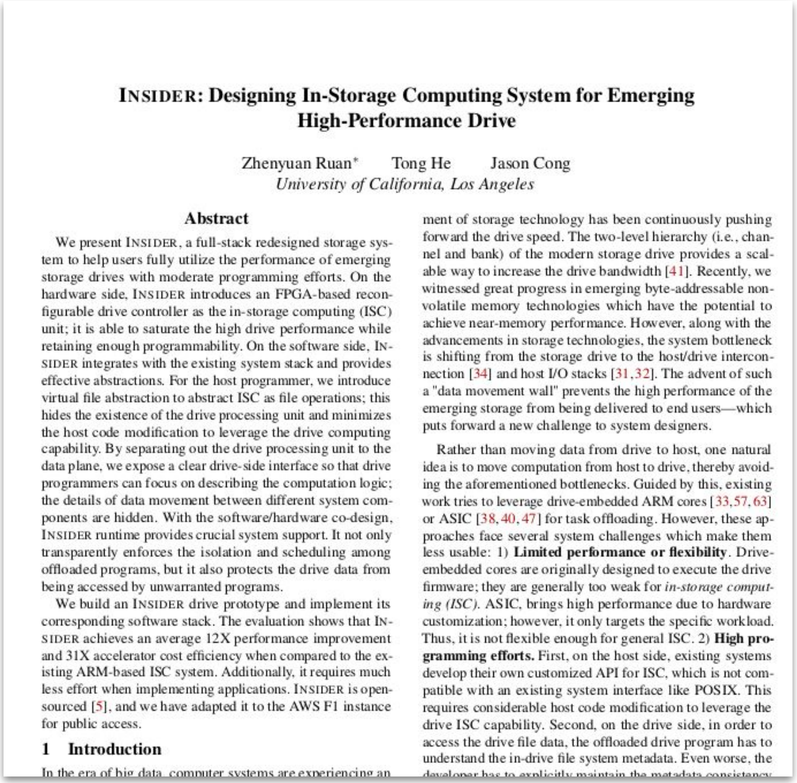
Modern SSDs are already software-defined, why restrict their use to a block-storage protocol like NVMe

Willow : a user programmable RPC-based SSD design (with limited memory and multi-tenancy management) - uses SPUs
Smart Query and Biscuit: query processing designs, with operator offloading and flow based programming - uses ARM

- Clean, flexible, and powerful
- Block I/O, direct I/O, Append, Transactions, Caching, and KV Store

Is running a general purpose MIPS/ARM processor a right choice? Are there alternative hardware options for programmability?

Insider : Designing In-Storage Computing System for Emerging High-Performance Drive (2019)



INSIDER: Designing In-Storage Computing System for Emerging High-Performance Drive

Zhenyuan Ruan* Tong He Jason Cong
University of California, Los Angeles

Abstract

We present INSIDER, a full-stack redesigned storage system to help users fully utilize the performance of emerging storage drives with moderate programming efforts. On the hardware side, INSIDER introduces an FPGA-based reconfigurable drive controller as the in-storage computing (ISC) unit; it is able to saturate the high drive performance while retaining enough programmability. On the software side, INSIDER integrates with the existing system stack and provides effective abstractions. For the host programmer, we introduce virtual file abstraction to abstract ISC as file operations; this hides the existence of the drive processing unit and minimizes the host code modification to leverage the drive computing capability. By separating out the drive processing unit to the data plane, we expose a clear drive-side interface so that drive programmers can focus on describing the computation logic; the details of data movement between different system components are hidden. With the software/hardware co-design, INSIDER runtime provides crucial system support. It not only transparently enforces the isolation and scheduling among offloaded programs, but it also protects the drive data from being accessed by unwarranted programs.

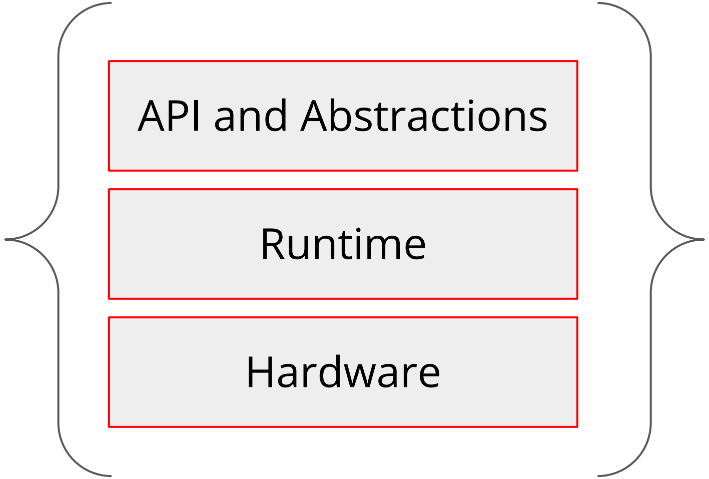
We build an INSIDER drive prototype and implement its corresponding software stack. The evaluation shows that INSIDER achieves an average 12X performance improvement and 31X accelerator cost efficiency when compared to the existing ARM-based ISC system. Additionally, it requires much less effort when implementing applications. INSIDER is open-sourced [5], and we have adapted it to the AWS F1 instance for public access.

1 Introduction

In the era of big data, computer systems are experiencing an

ment of storage technology has been continuously pushing forward the drive speed. The two-level hierarchy (i.e., channel and bank) of the modern storage drive provides a scalable way to increase the drive bandwidth [41]. Recently, we witnessed great progress in emerging byte-addressable non-volatile memory technologies which have the potential to achieve near-memory performance. However, along with the advancements in storage technologies, the system bottleneck is shifting from the storage drive to the host/drive interconnection [34] and host I/O stacks [31,32]. The advent of such a "data movement wall" prevents the high performance of the emerging storage from being delivered to end users—which puts forward a new challenge to system designers.

Rather than moving data from drive to host, one natural idea is to move computation from host to drive, thereby avoiding the aforementioned bottlenecks. Guided by this, existing work tries to leverage drive-embedded ARM cores [33,57,63] or ASIC [38,40,47] for task offloading. However, these approaches face several system challenges which make them less usable: 1) **Limited performance or flexibility**. Drive-embedded cores are originally designed to execute the drive firmware; they are generally too weak for *in-storage computing (ISC)*. ASIC, brings high performance due to hardware customization; however, it only targets the specific workload. Thus, it is not flexible enough for general ISC. 2) **High programming efforts.** First, on the host side, existing systems develop their own customized API for ISC, which is not compatible with an existing system interface like POSIX. This requires considerable host code modification to leverage the drive ISC capability. Second, on the drive side, in order to access the drive file data, the offloaded drive program has to understand the in-drive file system metadata. Even worse, the developer has to explicitly maintain the metadata consistency



API and Abstractions

Runtime

Hardware

Programmability needs Support from the Whole Stack

Hardware

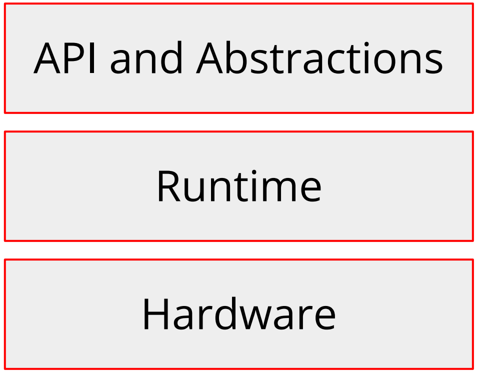
1. ASIC: fast but not-programmable
2. CPU: programmable but not fast

Runtime

1. How to ensure correct access from a code
2. How to ensure multi-tenancy with codes

API and Abstractions

1. New APIs leads to less familiarity with developers
2. Might lead to significant code modifications



API and Abstractions

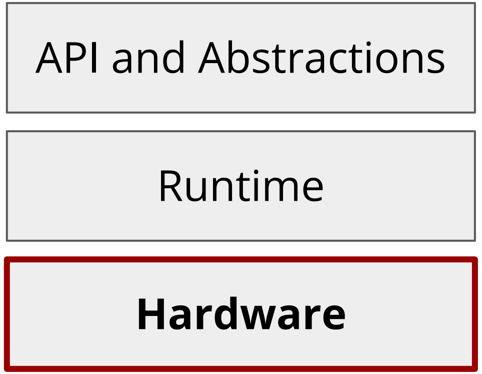
Runtime

Hardware

How to make Programmable Hardware?

Hardware?

- Candidates: ASIC, FPGA, GPU, ARM, x86
- Need to support
  - General programmability
  - Massive parallelism (all flash chips)
  - High energy efficiency



API and Abstractions

Runtime

Hardware

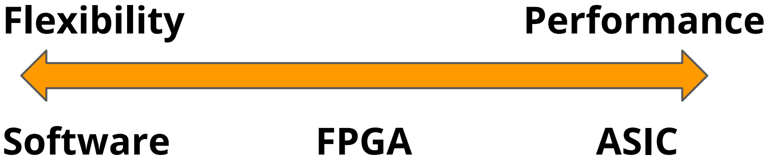
Do you know what FPGA is?

Field Programmable Gate Array (FPGA)

DIY hardware, programs can be compiled to be synthesized for FPGA

Very active area of research

- Performance
- Energy efficiency
- Domain-specific architectures



Flexibility **Performance**

Software **FPGA** **ASIC**

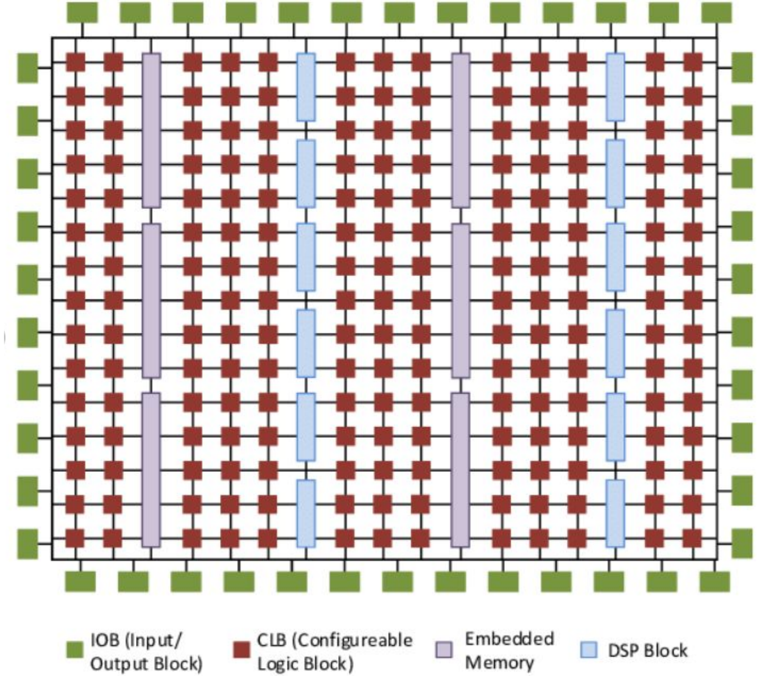
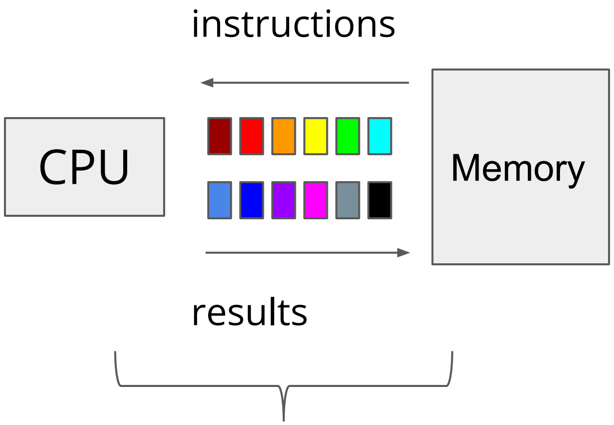

🟩 IOB (Input/ Output Block)
🟥 CLB (Configureable Logic Block)
▨ Embedded Memory
□ DSP Block

Image credit: B. Ronak et al, *Mapping for Maximum Performance on FPGA DSP Blocks*, https://ieeexplore.ieee.org/document/7229289

What is special about FPGA?



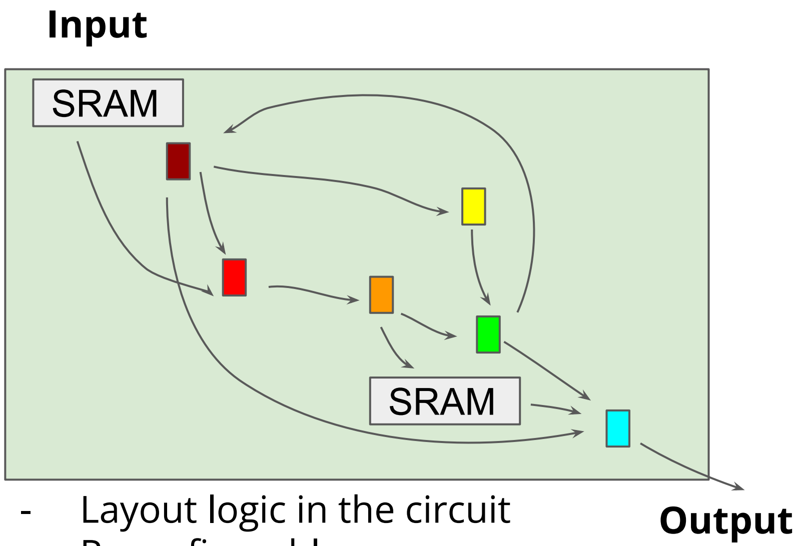
instructions

CPU

Memory

results

- Distance to memory
- Instruction dependencies
- Programming control units in CPUs



Input

SRAM

SRAM

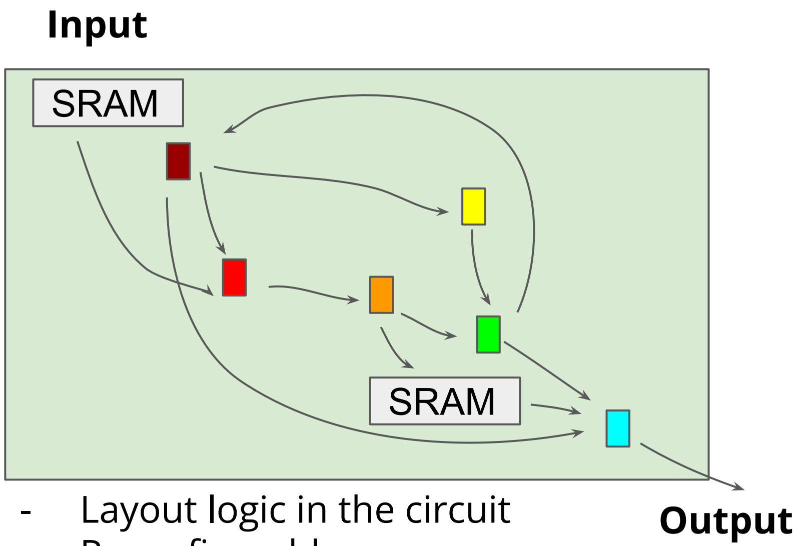
Output

- Layout logic in the circuit
- Reconfigurable
- Close and fast memory access
- Heavily pipelined

- Further reading: https://blog.esciencecenter.nl/why-use-an-fpga-instead-of-a-cpu-or-gpu-b234cd4f309c
- Zsolt Istvan, Building Distributed Storage with Specialized Hardware, https://www.research-collection.ethz.ch/bitstream/handle/20.500.11850/266096/1/zistvan-phd-dissert-rev.pdf

Sources of Performance Gains

1. Hardware-software co-design
   a. Trade easy operations in hardware with difficult ones
2. Specialized operations
   a. Use FPGA and specialized operations
3. Leverage parallelism
   a. Processing elements (PEs) and space
4. Local memories
   a. Leverage SRAM
5. Maximize off-chip DRAM access
   a. Large sequential accesses
6. Reduce programming overheads
   a. Heavy pipelining



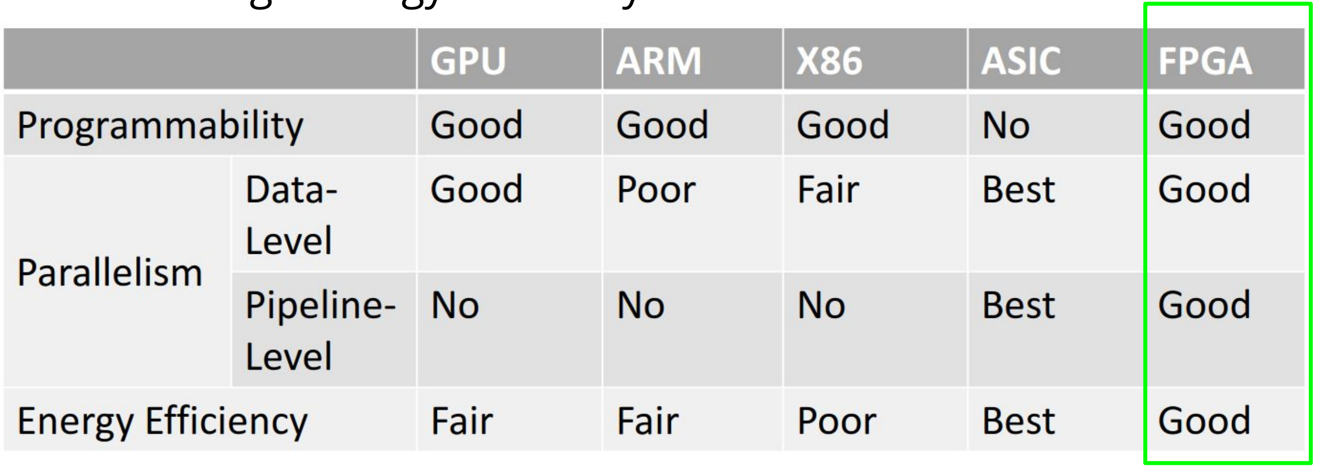
- Layout logic in the circuit
- Reconfigurable
- Close and fast memory access
- Heavily pipelined

See *Darwin: A Genomics Co-processor Provides up to 15,000X Acceleration on Long Read Assembly.* ASPLOS 2018.

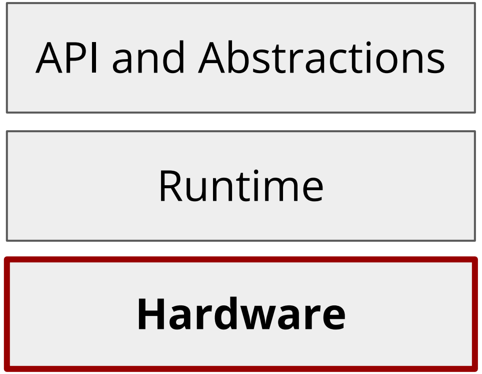
How to make Programmable Hardware?

Hardware?

- Candidates: ASIC, FPGA, GPU, ARM, X86
- Need to support
  - General programmability
  - Massive parallelism (all flash chips)
  - High energy efficiency



		GPU	ARM	X86	ASIC	FPGA
Programmability		Good	Good	Good	No	Good
Parallelism	Data-Level	Good	Poor	Fair	Best	Good
	Pipeline-Level	No	No	No	Best	Good
Energy Efficiency		Fair	Fair	Poor	Best	Good



Programmability needs Support from the Whole Stack

Hardware

1. ASIC: fast but not-programmable
2. CPU: programmable but not fast

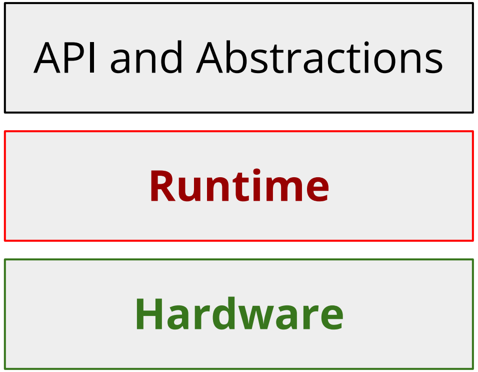
→ ***Use FPGA***

Runtime

1. How to ensure correct access from a code
2. How to ensure multi-tenancy with codes

API and Abstractions

1. New APIs leads to less familiarity with developers
2. Might lead to significant code modifications

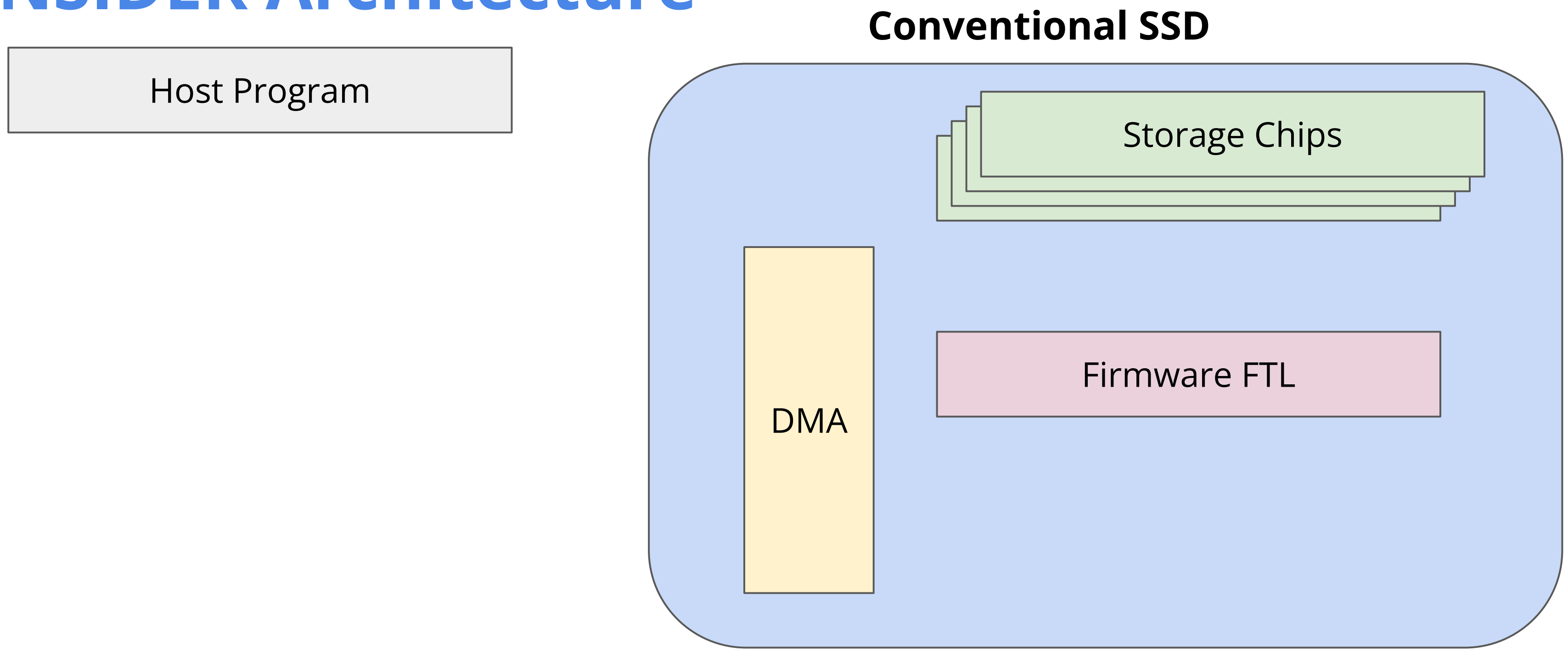


API and Abstractions

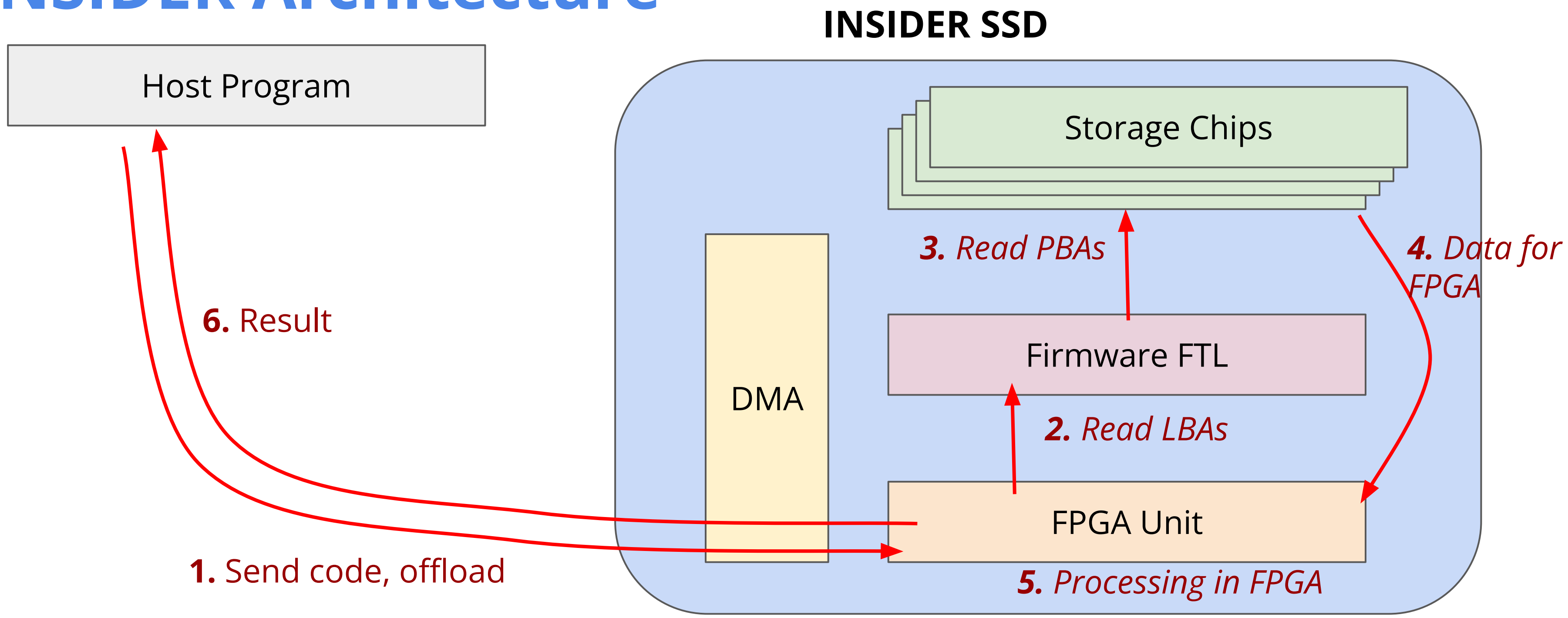
Runtime

Hardware

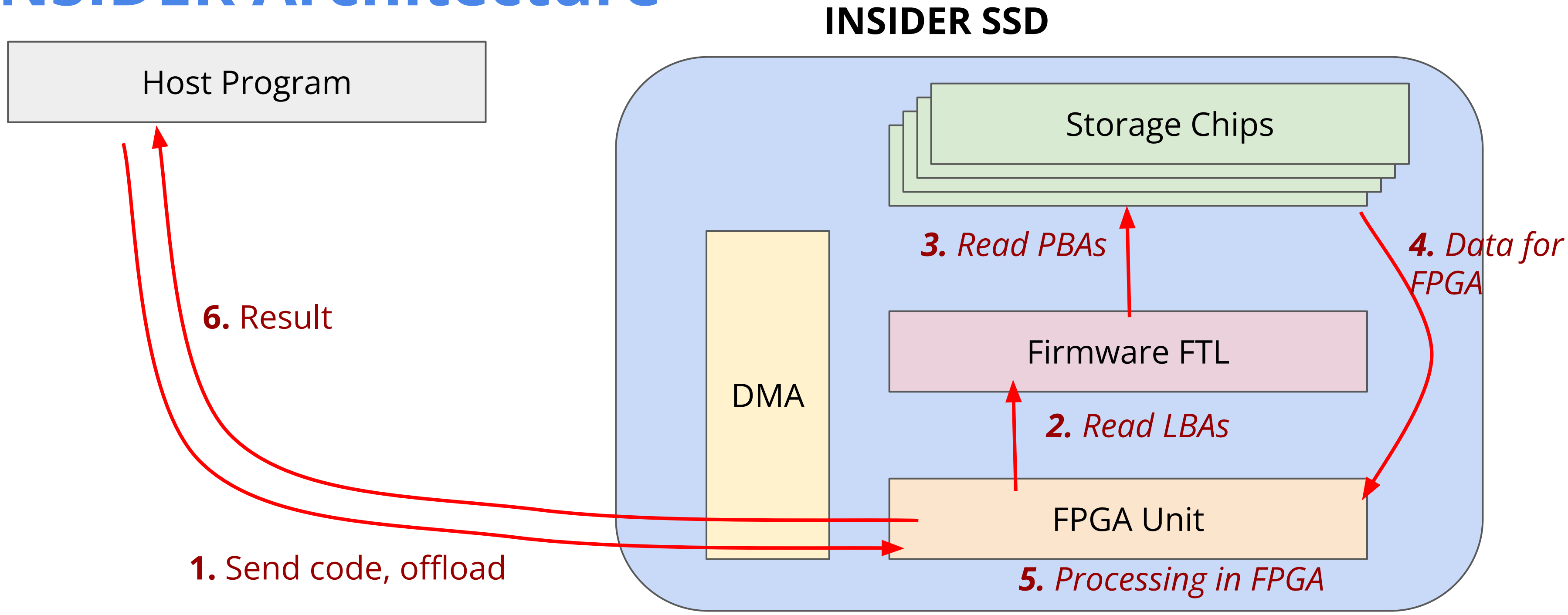
INSIDER Architecture



INSIDER Architecture

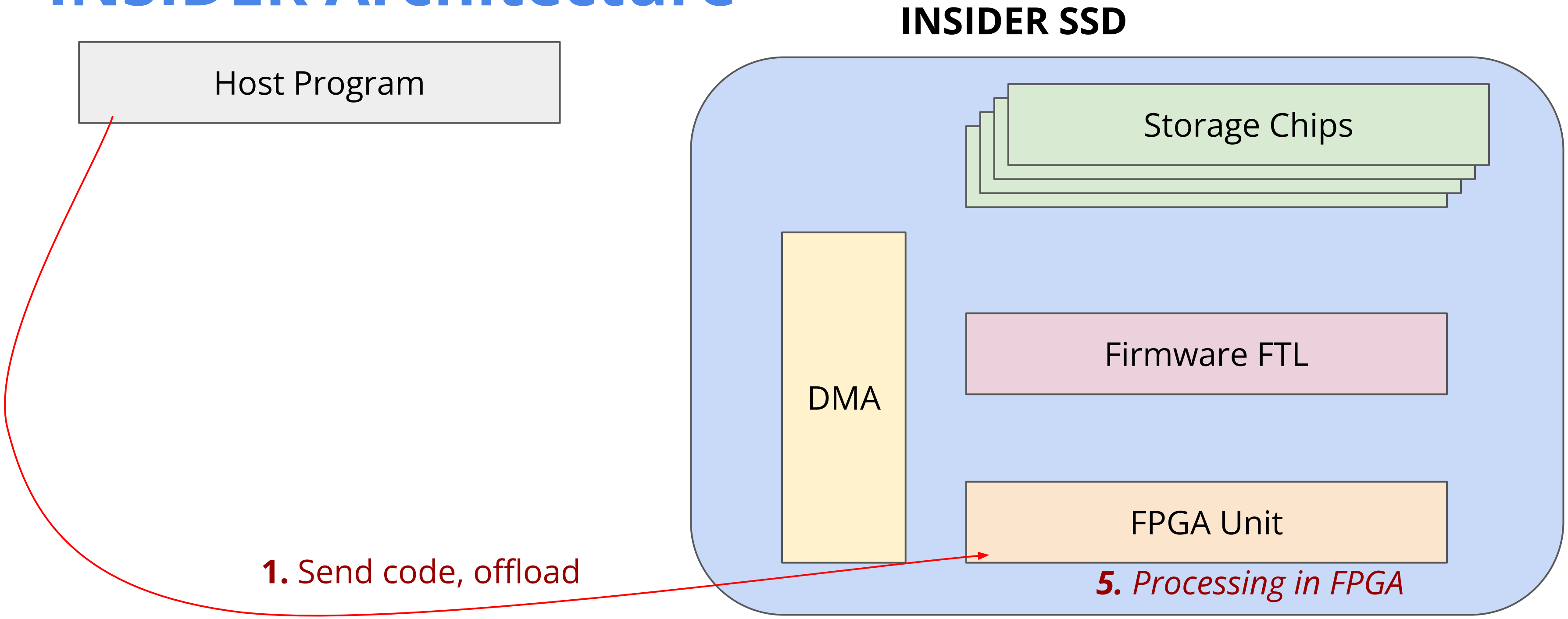


INSIDER Architecture



How to make sure a rogue FPGA program is not able to read any arbitrary storage location or write to any location?

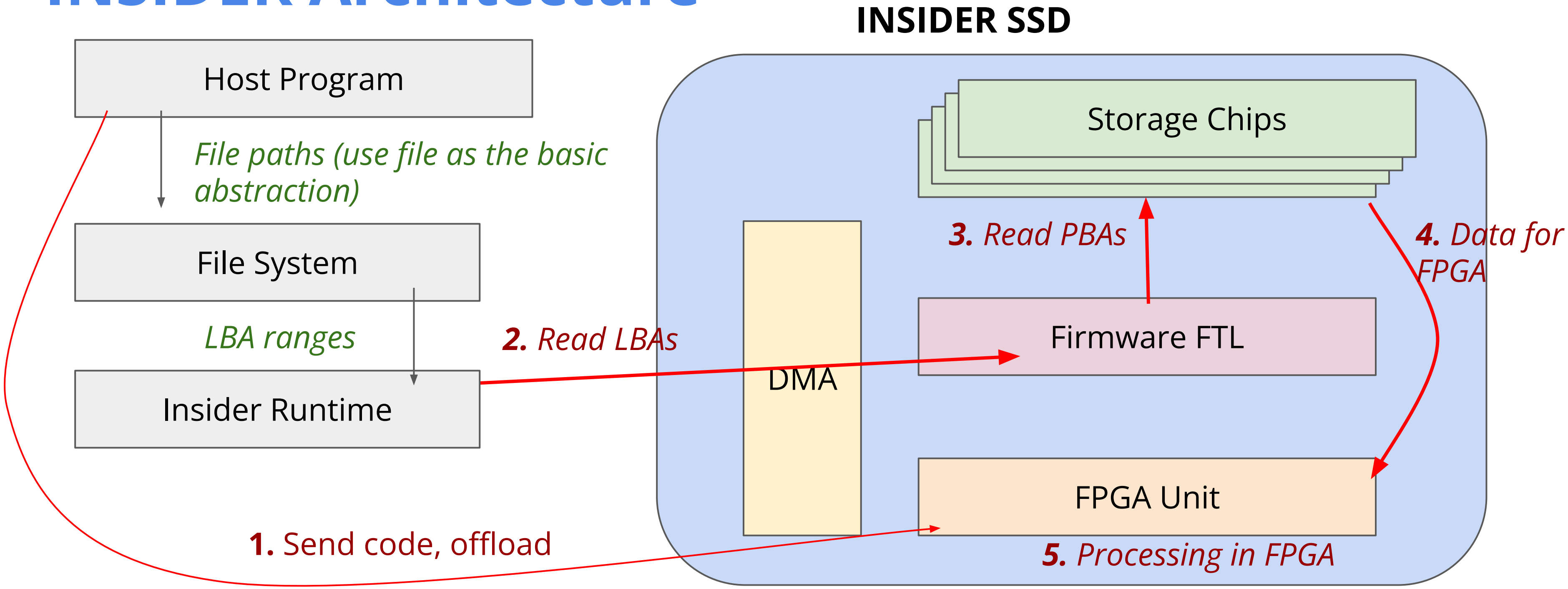
INSIDER Architecture



Idea 1: Make FPGA program “**Compute-Only**”, hence the program itself cannot issue any r/w ops.

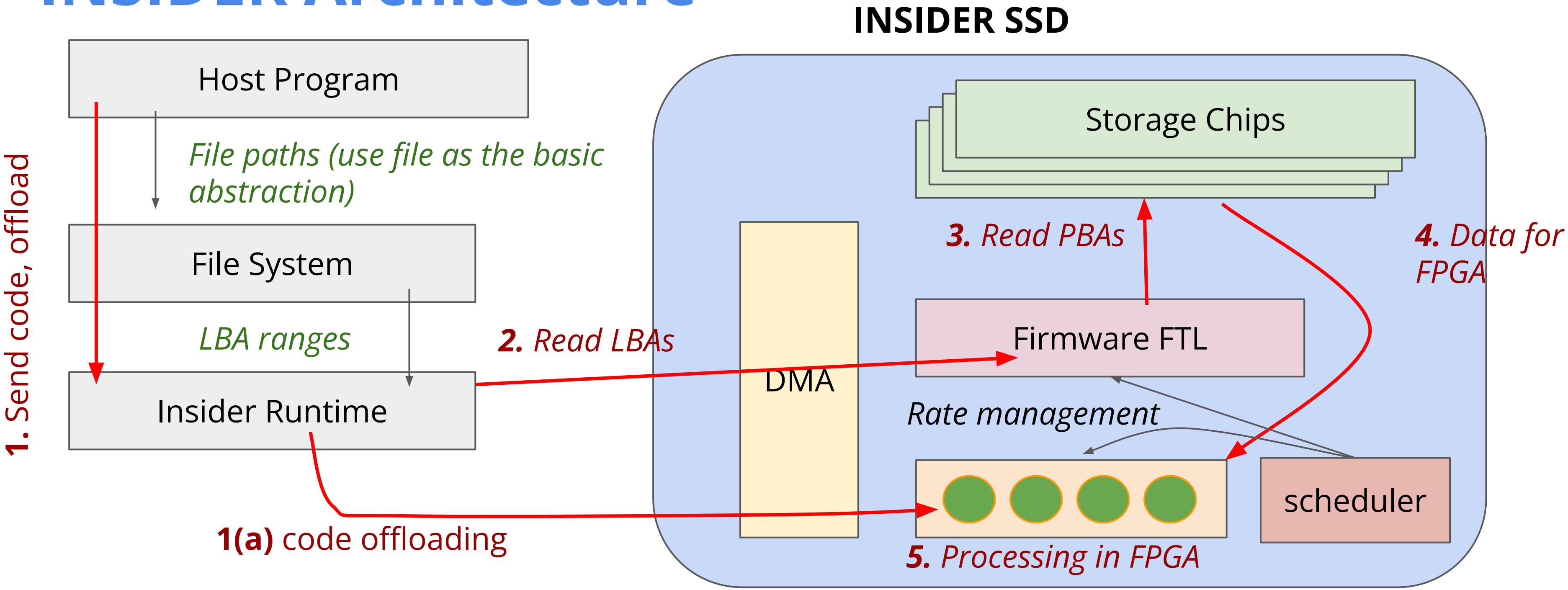
Then how to get data from flash chips for processing?

INSIDER Architecture



Idea 1: Make FPGA program "**Compute-Only**", hence the program itself cannot issue any r/w ops.
Idea 2: Make a separate "**control plane**" which issues read operations for data which FPGA processes

INSIDER Architecture



Idea 1: Make FPGA program "**Compute-Only**", hence the program itself cannot issue any r/w ops.
Idea 2: Make a separate "**control plane**" which issues read operations for data which FPGA processes
Idea 3: Partition the FPGA into independent processing spaces for parallelism + scheduler

Programmability needs Support from the Whole Stack

Hardware

Use FPGA

1. ASIC: fast but not-programmable
2. CPU: programmable but not fast

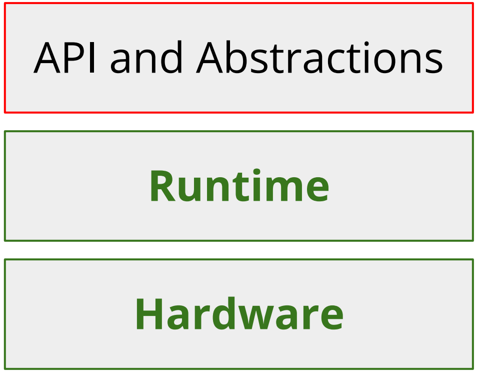
Runtime

Compute-only programs with FPGA partitioning

1. How to ensure correct access from a code
2. How to ensure multi-tenancy with codes

API and Abstractions

1. New APIs leads to less familiarity with developers
2. Might lead to significant code modifications



Files, File, and Files everywhere!

Everything is a file - The UNIX philosophy :)

```
// get a virtual file
vfile = reg_virt_file (real_file, accelerator id);

int fd = vopen(vfile, flags);

send_params(fd, void * argc, int argv);

int sz = vread  (fd, buf, buf_size);
int sz = vwrite (fd, buf, buf_size);

// vsync - if written

vclose(fd);
```

Tells INSIDER which files to prep for reading, reserve id

Check file systems permissions, and hold the file for processing

Send FPGA program parameters

These reads and writes move data from flash to FPGA for processing. Hence, the virtual. Only the final result is returned!

Synchronize and close the file to release resources

Files, File, and Files everywhere!

Everything is a file - The UNIX philosophy :)

```
// get a virtual file
vfile =

int fd =

send_par

int sz =
int sz =

// vsync - if written

vclose(fd);
```

Tells INSIDER which files to prep for reading, reserve id

s permissions, and hold ssing

am parameters

vrites move data from processing. Hence, the virtual. Only the final result is returned!

Synchronize and close the file to release resources

You can see the basic compute-only idea here that the user program needs to issue `vread/vwrites` to trigger data movements from the flash chips to FPGA.

FPGA itself cannot issue a read or write request!

How Does FPGA Code Look Like?

Like a simple C++ code … (INSIDER provides a compiler)

```
(simplified)

struct app_data {
char bytes[64];
int length;
bool eop;
}
```

```
(simplified)

void filter(Queue<app_data> input, Queue<app_data>
output, void *argv, int argc)
{
  // use argv, argv to setup the environment
  item_to_process = input.read();
  result = process(item_to_process);
  output.append(result);
}

// Essentially a record-by-record processing
```

Programmability needs Support from the Whole Stack

Hardware

Use FPGA

1. ASIC: fast but not-programmable
2. CPU: programmable but not fast

Runtime

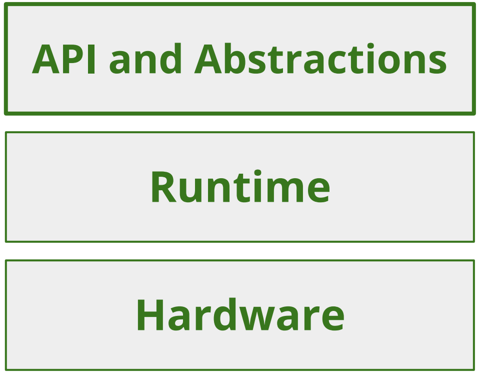
Compute-only programs with FPGA partitioning

1. How to ensure correct access from a code
2. How to ensure multi-tenancy with codes

API and Abstractions

Virtual files

1. New APIs leads to less familiarity with developers
2. Might lead to significant code modifications

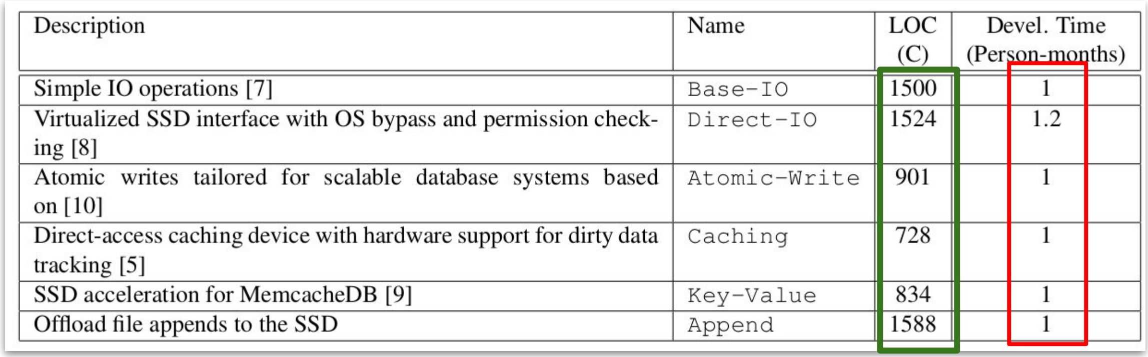


API and Abstractions

Runtime

Hardware

Is It Simple? Compared to Moneta

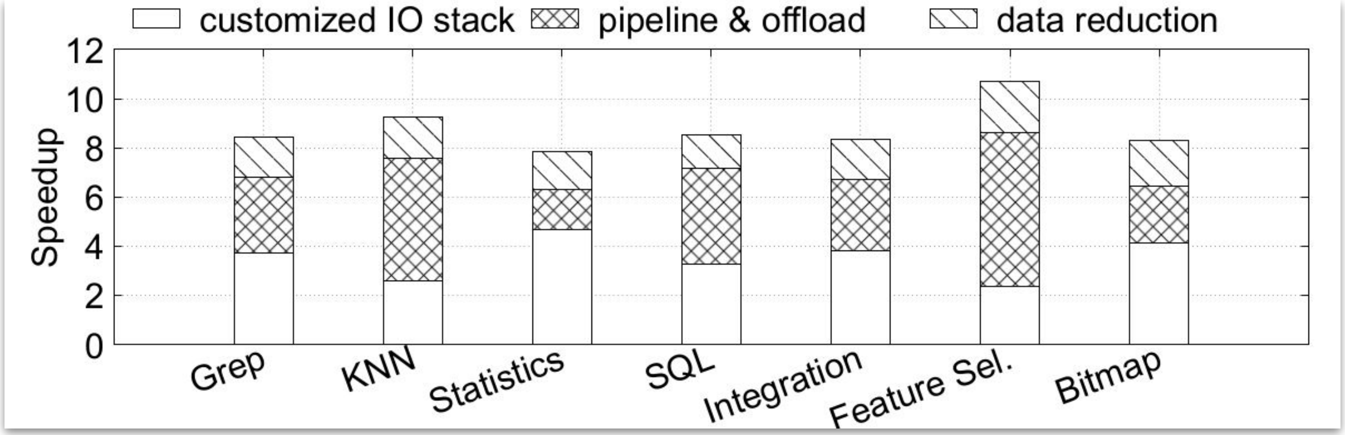


Description	Name	LOC (C)	Devel. Time (Person-months)
Simple IO operations [7]	`Base-IO`	1500	1
Virtualized SSD interface with OS bypass and permission checking [8]	`Direct-IO`	1524	1.2
Atomic writes tailored for scalable database systems based on [10]	`Atomic-Write`	901	1
Direct-access caching device with hardware support for dirty data tracking [5]	`Caching`	728	1
SSD acceleration for MemcacheDB [9]	`Key-Value`	834	1
Offload file appends to the SSD	`Append`	1588	1

Application	Devel.Time (Person-Day)	LOC	
		Host	Drive
Grep	3	51	193
KNN	2	77	72
Statistics	3	65	170
SQL Query	5	97	256
Data Integration	5	41	307
Feature Selection	9	50	632
Bitmap file decompression	5	94	213

File based interface does offer tangible benefits in terms of developer's familiarity

INSIDER: Performance



- Baseline : implementation on POSIX files on host
- Customized I/O Stack: Host-bypass, and use vread of INSIDER to bypass the host fs/block overheads
- Pipeline and offload : Overlap compute and data movement, and offload code to INSIDER drive
- Data Reduction: Gains from reducing the amount of data movement from the drive to the host

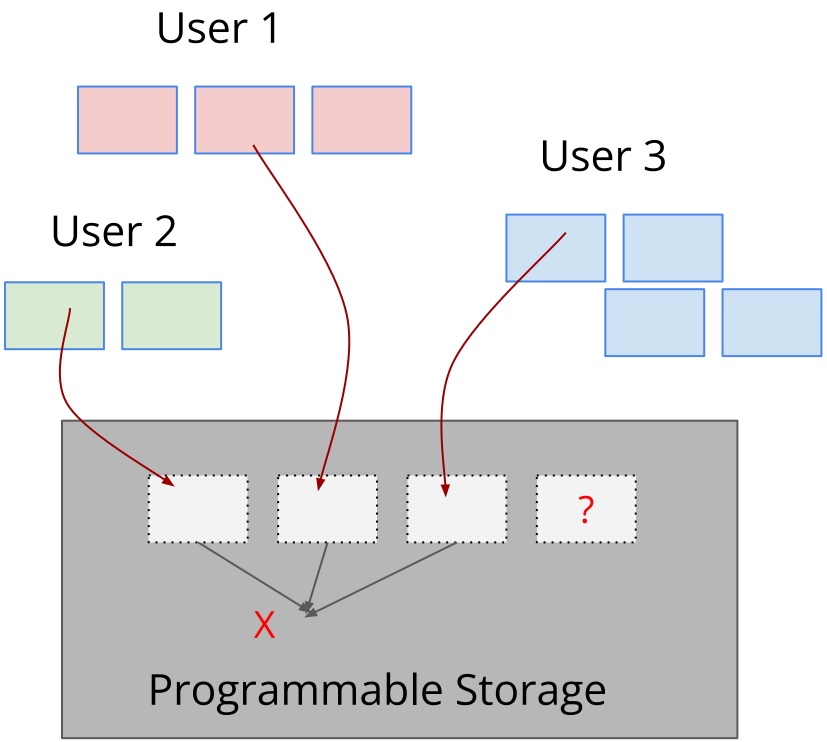
Almost an order of magnitude performance gains

Is FPGA the only way to provide Programmability?

No - programmability is a large concept with multiple independent ideas

- Programmability in storage device
    - Integrated : ASIC, FPGA, or embedded CPU
    - Side-by-side: FGPA, GPUs, ASICs, co-processor (DSPs) etc.
- How to ensure multi tenancy and isolation?

Scheduling, Multi-Tenancy and Isolation



Installing and running user-provided extensions safely

Scheduling, which extension to pick next

Would it yield? Preemption?

How to ensure isolation : security and performance for multi-tenancy

Parallel themes in the OS/Kernel development, fault isolation, static and dynamic verifications, etc.

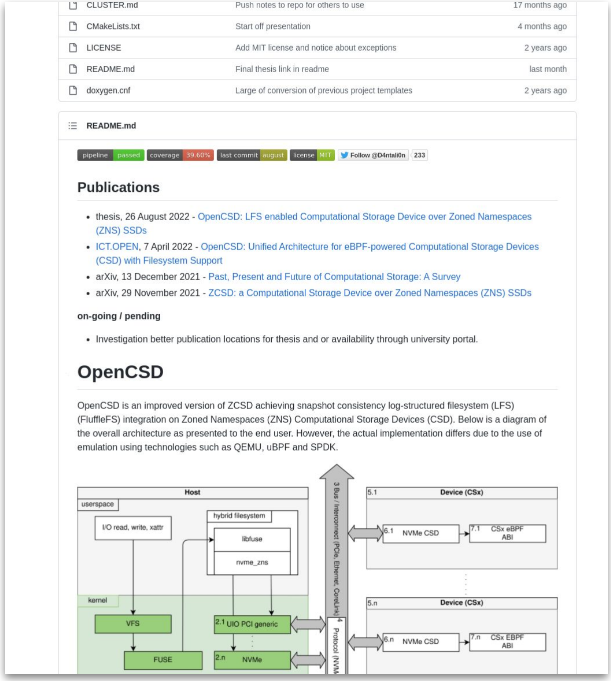
- Architecture
- Systems software
- Language and runtimes

Is FPGA the only way to provide Programmability?

No - programmability is a large concept with multiple independent ideas

- Programmability in storage device
  - Integrated : ASIC, FPGA, or embedded CPU
  - Side-by-side: FGPA, GPUs, ASICs, co-processor (DSPs) etc.
- How to ensure multi tenancy and isolation?
  - Hardware
    - (INSIDER) FPGA: partition the FPGA
    - (Willow/Biscuit): Use SPU-OS/ARM process scheduling
  - **Software, use programming languages to provide isolation and correctness**
      - Rust, Java script, eBPF (⇐ *we are working on it, see further reading*)
- What is the new programming abstraction?
  - RPCs, Virtual Files, ???

MSc thesis also:



CLUSTER.md	Push notes to repo for others to use	17 months ago
CMakeLists.txt	Start off presentation	4 months ago
LICENSE	Add MIT license and notice about exceptions	2 years ago
README.md	Final thesis link in readme	last month
doxygen.cnf	Large of conversion of previous project templates	2 years ago

README.md

pipeline passed | coverage 39.60% | last commit august | license MIT | Follow @D4ntali0n 233

Publications

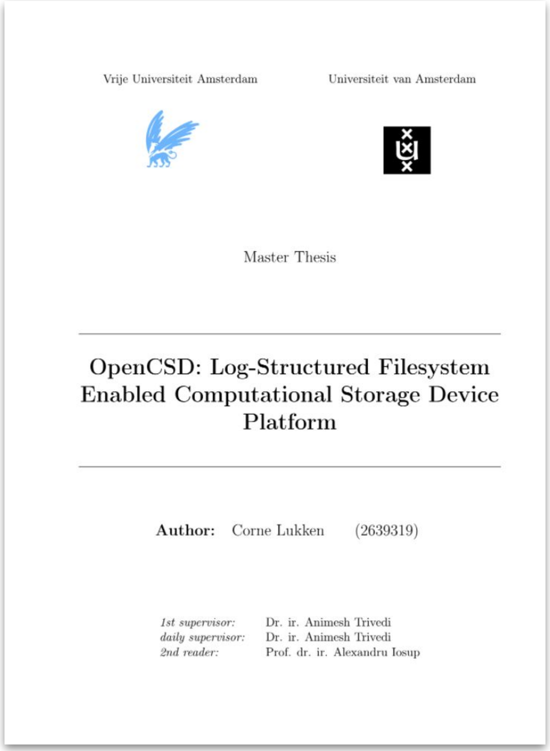
- thesis, 26 August 2022 - OpenCSD: LFS enabled Computational Storage Device over Zoned Namespaces (ZNS) SSDs
- ICT.OPEN, 7 April 2022 - OpenCSD: Unified Architecture for eBPF-powered Computational Storage Devices (CSD) with Filesystem Support
- arXiv, 13 December 2021 - Past, Present and Future of Computational Storage: A Survey
- arXiv, 29 November 2021 - ZCSD: a Computational Storage Device over Zoned Namespaces (ZNS) SSDs

on-going / pending

- Investigation better publication locations for thesis and or availability through university portal.

OpenCSD

OpenCSD is an improved version of ZCSD achieving snapshot consistency log-structured filesystem (LFS) (FluffleFS) integration on Zoned Namespaces (ZNS) Computational Storage Devices (CSD). Below is a diagram of the overall architecture as presented to the end user. However, the actual implementation differs due to the use of emulation using technologies such as QEMU, uBPF and SPDK.



Vrije Universiteit Amsterdam

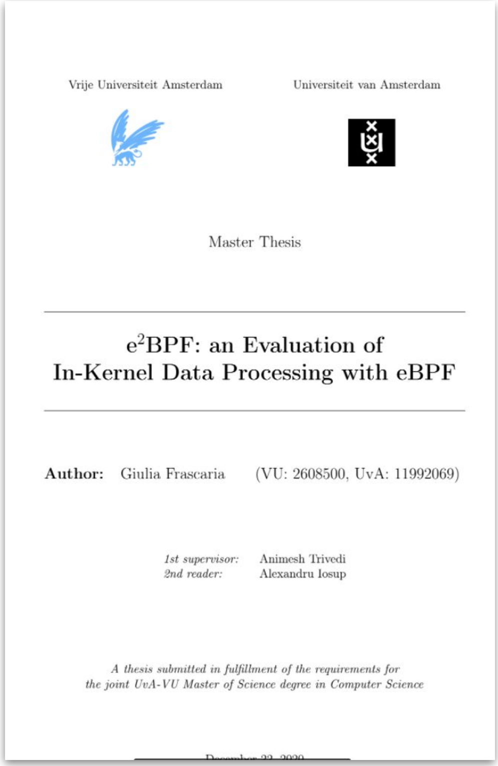
Universiteit van Amsterdam

Master Thesis

OpenCSD: Log-Structured Filesystem Enabled Computational Storage Device Platform

Author: Corne Lukken (2639319)

1st supervisor: Dr. ir. Animesh Trivedi
daily supervisor: Dr. ir. Animesh Trivedi
2nd reader: Prof. dr. ir. Alexandru Iosup



Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

e^{2}BPF: an Evaluation of In-Kernel Data Processing with eBPF

Author: Giulia Frascaria (VU: 2608500, UvA: 11992069)

1st supervisor: Animesh Trivedi
2nd reader: Alexandru Iosup

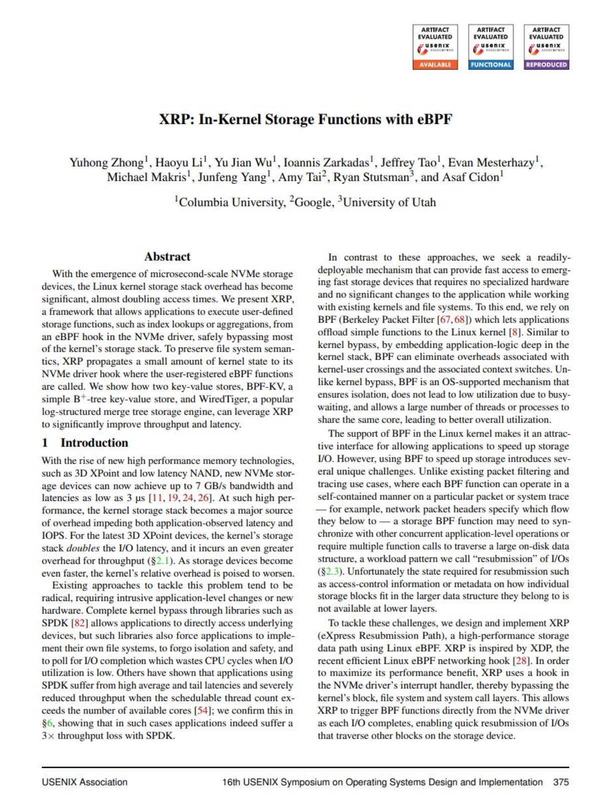
A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

December 22, 2020

OpenCSD Platform:
https://github.com/Dantali0n/OpenCSD

https://animeshtrivedi.github.io/team/

eBPF-based Kernel Programming



ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE | ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL | ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

XRP: In-Kernel Storage Functions with eBPF

Yuhong Zhong[1], Haoyu Li[1], Yu Jian Wu[1], Ioannis Zarkadas[1], Jeffrey Tao[1], Evan Mesterhazy[1],
Michael Makris[1], Junfeng Yang[1], Amy Tai[2], Ryan Stutsman[3], and Asaf Cidon[1]

[1]Columbia University, [2]Google, [3]University of Utah

Abstract

With the emergence of microsecond-scale NVMe storage devices, the Linux kernel storage stack overhead has become significant, almost doubling access times. We present XRP, a framework that allows applications to execute user-defined storage functions, such as index lookups or aggregations, from an eBPF hook in the NVMe driver, safely bypassing most of the kernel's storage stack. To preserve file system semantics, XRP propagates a small amount of kernel state to its NVMe driver hook where the user-registered eBPF functions are called. We show how two key-value stores, BPF-KV, a simple B^+-tree key-value store, and WiredTiger, a popular log-structured merge tree storage engine, can leverage XRP to significantly improve throughput and latency.

1 Introduction

With the rise of new high performance memory technologies, such as 3D XPoint and low latency NAND, new NVMe storage devices can now achieve up to 7 GB/s bandwidth and latencies as low as 3 μs [11, 19, 24, 26]. At such high performance, the kernel storage stack becomes a major source of overhead impeding both application-observed latency and IOPS. For the latest 3D XPoint devices, the kernel's storage stack *doubles* the I/O latency, and it incurs an even greater overhead for throughput (§2.1). As storage devices become even faster, the kernel's relative overhead is poised to worsen.

Existing approaches to tackle this problem tend to be radical, requiring intrusive application-level changes or new hardware. Complete kernel bypass through libraries such as SPDK [82] allows applications to directly access underlying devices, but such libraries also force applications to implement their own file systems, to forgo isolation and safety, and to poll for I/O completion which wastes CPU cycles when I/O utilization is low. Others have shown that applications using SPDK suffer from high average and tail latencies and severely reduced throughput when the schedulable thread count exceeds the number of available cores [54]; we confirm this in §6, showing that in such cases applications indeed suffer a 3× throughput loss with SPDK.

In contrast to these approaches, we seek a readily-deployable mechanism that can provide fast access to emerging fast storage devices that requires no specialized hardware and no significant changes to the application while working with existing kernels and file systems. To this end, we rely on BPF (Berkeley Packet Filter [67, 68]) which lets applications offload simple functions to the Linux kernel [8]. Similar to kernel bypass, by embedding application-logic deep in the kernel stack, BPF can eliminate overheads associated with kernel-user crossings and the associated context switches. Unlike kernel bypass, BPF is an OS-supported mechanism that ensures isolation, does not lead to low utilization due to busy-waiting, and allows a large number of threads or processes to share the same core, leading to better overall utilization.

The support of BPF in the Linux kernel makes it an attractive interface for allowing applications to speed up storage I/O. However, using BPF to speed up storage introduces several unique challenges. Unlike existing packet filtering and tracing use cases, where each BPF function can operate in a self-contained manner on a particular packet or system trace — for example, network packet headers specify which flow they below to — a storage BPF function may need to synchronize with other concurrent application-level operations or require multiple function calls to traverse a large on-disk data structure, a workload pattern we call "resubmission" of I/Os (§2.3). Unfortunately the state required for resubmission such as access-control information or metadata on how individual storage blocks fit in the larger data structure they belong to is not available at lower layers.

To tackle these challenges, we design and implement XRP (eXpress Resubmission Path), a high-performance storage data path using Linux eBPF. XRP is inspired by XDP, the recent efficient Linux eBPF networking hook [28]. In order to maximize its performance benefit, XRP uses a hook in the NVMe driver's interrupt handler, thereby bypassing the kernel's block, file system and system call layers. This allows XRP to trigger BPF functions directly from the NVMe driver as each I/O completes, enabling quick resubmission of I/Os that traverse other blocks on the storage device.

USENIX Association 16th USENIX Symposium on Operating Systems Design and Implementation 375

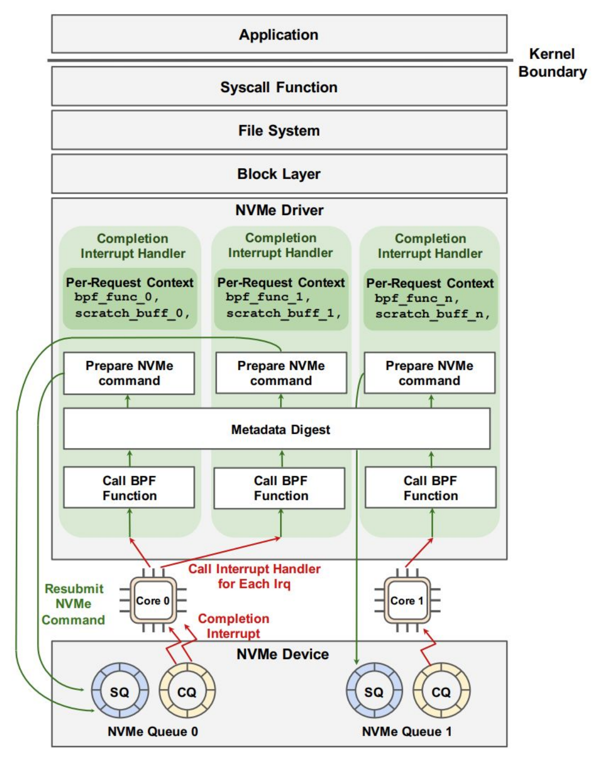
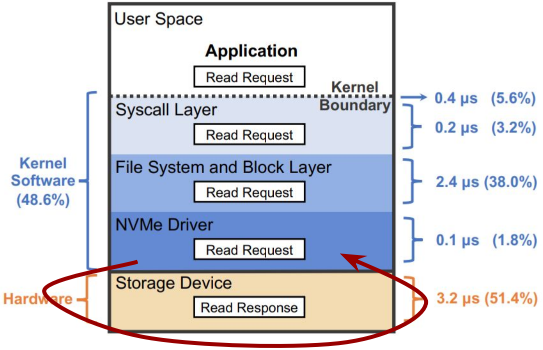
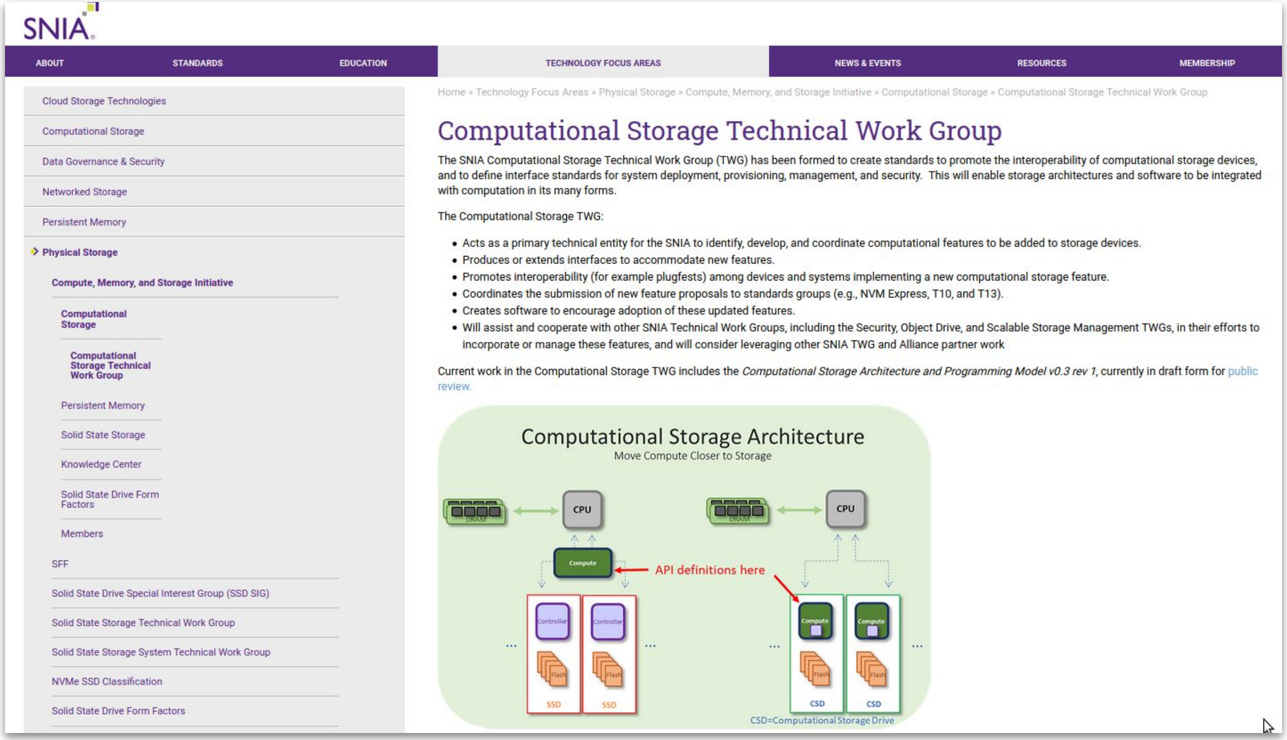


Figure 4: XRP architecture.



Computation Storage: New Emerging Standard


SNIA
ABOUT
STANDARDS
EDUCATION
TECHNOLOGY FOCUS AREAS
NEWS & EVENTS
RESOURCES
MEMBERSHIP
Cloud Storage Technologies
Computational Storage
Data Governance & Security
Networked Storage
Persistent Memory
Physical Storage
Compute, Memory, and Storage Initiative
Computational Storage
Computational Storage Technical Work Group
Persistent Memory
Solid State Storage
Knowledge Center
Solid State Drive Form Factors
Members
SFF
Solid State Drive Special Interest Group (SSD SIG)
Solid State Storage Technical Work Group
Solid State Storage System Technical Work Group
NVMe SSD Classification
Solid State Drive Form Factors
Home » Technology Focus Areas » Physical Storage » Compute, Memory, and Storage Initiative » Computational Storage » Computational Storage Technical Work Group

Computational Storage Technical Work Group

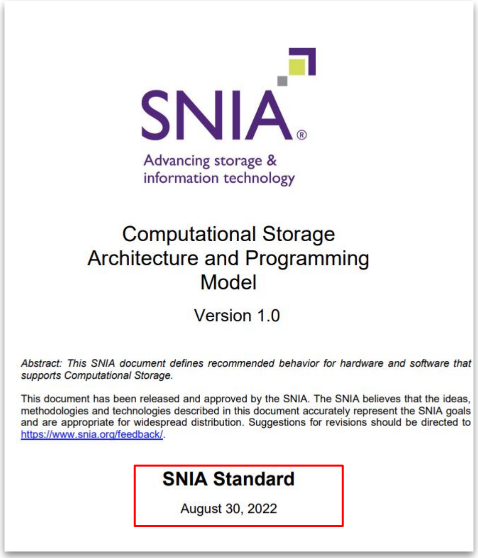
The SNIA Computational Storage Technical Work Group (TWG) has been formed to create standards to promote the interoperability of computational storage devices, and to define interface standards for system deployment, provisioning, management, and security. This will enable storage architectures and software to be integrated with computation in its many forms.

The Computational Storage TWG:

- Acts as a primary technical entity for the SNIA to identify, develop, and coordinate computational features to be added to storage devices.
- Produces or extends interfaces to accommodate new features.
- Promotes interoperability (for example plugfests) among devices and systems implementing a new computational storage feature.
- Coordinates the submission of new feature proposals to standards groups (e.g., NVM Express, T10, and T13).
- Creates software to encourage adoption of these updated features.
- Will assist and cooperate with other SNIA Technical Work Groups, including the Security, Object Drive, and Scalable Storage Management TWGs, in their efforts to incorporate or manage these features, and will consider leveraging other SNIA TWG and Alliance partner work

Current work in the Computational Storage TWG includes the *Computational Storage Architecture and Programming Model v0.3 rev 1*, currently in draft form for public review.

Computational Storage Architecture
Move Compute Closer to Storage
DRAM
CPU
Compute
API definitions here
Controller
Flash
SSD
CSD
...
CSD=Computational Storage Drive


SNIA
Advancing storage & information technology

Computational Storage Architecture and Programming Model

Version 1.0

Abstract: This SNIA document defines recommended behavior for hardware and software that supports Computational Storage.

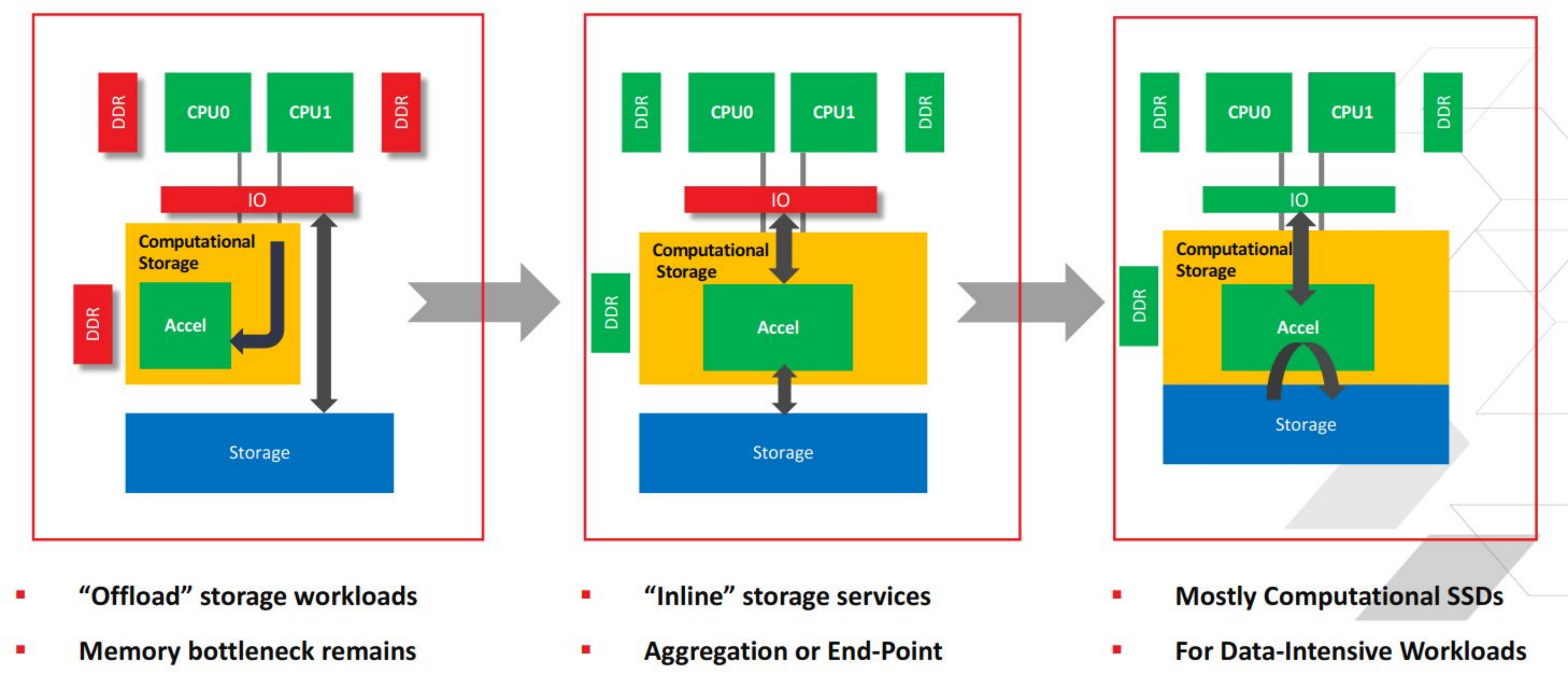
This document has been released and approved by the SNIA. The SNIA believes that the ideas, methodologies and technologies described in this document accurately represent the SNIA goals and are appropriate for widespread distribution. Suggestions for revisions should be directed to https://www.snia.org/feedback/.

SNIA Standard

August 30, 2022

- https://www.snia.org/computationaltwg **(should know what SNIA is)**
- https://www.snia.org/tech_activities/publicreview

There are Various Settings Possible



- “Offload” storage workloads
- Memory bottleneck remains

- “Inline” storage services
- Aggregation or End-Point

- Mostly Computational SSDs
- For Data-Intensive Workloads

Computational SSDs, Xilinx, https://www.snia.org/sites/default/files/SDCEMEA/2019/Presentations/Computational_SSDs_Final.pdf

So When Does Using CSD Makes Sense?

CSD: Computation Storage Device, or CS Computational Storage

When offloading computation to the device helps

- Large data transfer reduction is possible
- When data delivery or access does not need any CPU intervention
  - Example, put a video compressor in the FPGA for storing video files, compression, deduplication

When it might have limited gains?

- Compute heavy workloads with limited/small data transfers
- Little parallelism in the workload

Before We Conclude

A large field with different application domains, and names

- Near-Data Processing (NDP), In Storage Computation (ISC), Computational Storage (CS) and many more

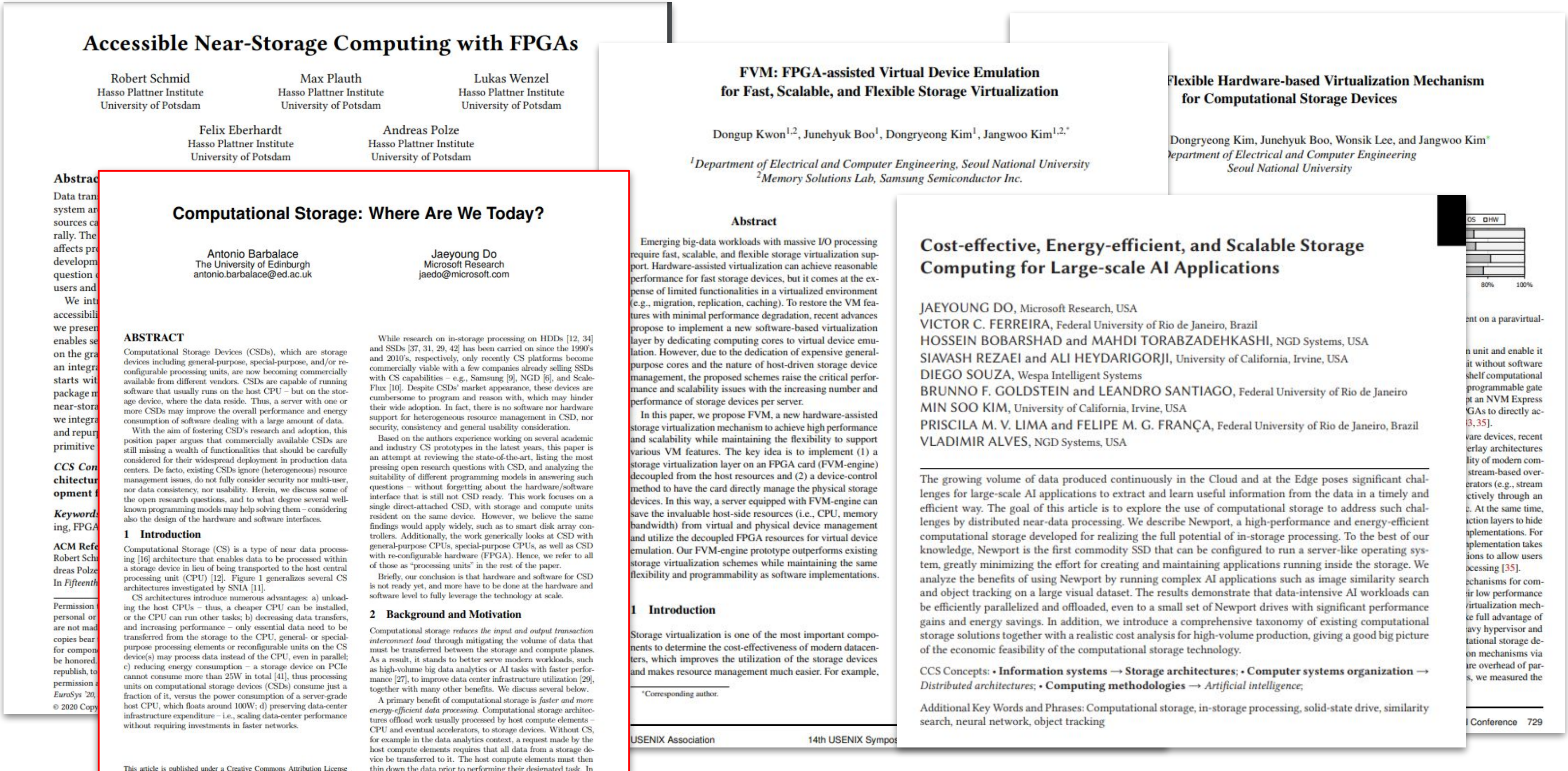
There are many flavors of programming...

1. Map/Reduce, Spark - also ship compute code to the data server for local execution
2. There is a big field of Database research on programmable storage where particular DB operators or complete queries are offloaded in storage drives
    a. Pushdown of filter predicates, aggregate operators from query plans

Programmability: custom untrusted code, protection, usability and expessibility

We are currently investigating how to design, build, and use programmable storage for data processing - *interested*?

Whole lots of new work in the past 2 years



Accessible Near-Storage Computing with FPGAs

Robert Schmid
Hasso Plattner Institute
University of Potsdam

Max Plauth
Hasso Plattner Institute
University of Potsdam

Lukas Wenzel
Hasso Plattner Institute
University of Potsdam

Felix Eberhardt
Hasso Plattner Institute
University of Potsdam

Andreas Polze
Hasso Plattner Institute
University of Potsdam

Computational Storage: Where Are We Today?

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

Jaeyoung Do
Microsoft Research
jaedo@microsoft.com

ABSTRACT

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

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

1 Introduction

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

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

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

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

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

2 Background and Motivation

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

A primary benefit of computational storage is *faster and more energy-efficient data processing*. Computational storage architectures offload work usually processed by host compute elements – CPU and eventual accelerators, to storage devices. Without CS, for example in the data analytics context, a request made by the host compute elements requires that all data from a storage device be transferred to it. The host compute elements must then thin down the data prior to performing their designated task. In

FVM: FPGA-assisted Virtual Device Emulation for Fast, Scalable, and Flexible Storage Virtualization

Dongup Kwon[1,2], Junehyuk Boo[1], Dongryeong Kim[1], Jangwoo Kim[1,2,*]

[1]Department of Electrical and Computer Engineering, Seoul National University
[2]Memory Solutions Lab, Samsung Semiconductor Inc.

Abstract

Emerging big-data workloads with massive I/O processing require fast, scalable, and flexible storage virtualization support. Hardware-assisted virtualization can achieve reasonable performance for fast storage devices, but it comes at the expense of limited functionalities in a virtualized environment (e.g., migration, replication, caching). To restore the VM features with minimal performance degradation, recent advances propose to implement a new software-based virtualization layer by dedicating computing cores to virtual device emulation. However, due to the dedication of expensive general-purpose cores and the nature of host-driven storage device management, the proposed schemes raise the critical performance and scalability issues with the increasing number and performance of storage devices per server.

In this paper, we propose FVM, a new hardware-assisted storage virtualization mechanism to achieve high performance and scalability while maintaining the flexibility to support various VM features. The key idea is to implement (1) a storage virtualization layer on an FPGA card (FVM-engine) decoupled from the host resources and (2) a device-control method to have the card directly manage the physical storage devices. In this way, a server equipped with FVM-engine can save the invaluable host-side resources (i.e., CPU, memory bandwidth) from virtual and physical device management and utilize the decoupled FPGA resources for virtual device emulation. Our FVM-engine prototype outperforms existing storage virtualization schemes while maintaining the same flexibility and programmability as software implementations.

1 Introduction

Storage virtualization is one of the most important components to determine the cost-effectiveness of modern datacenters, which improves the utilization of the storage devices and makes resource management much easier. For example,

[*]Corresponding author.

Flexible Hardware-based Virtualization Mechanism for Computational Storage Devices

Dongryeong Kim, Junehyuk Boo, Wonsik Lee, and Jangwoo Kim*
Department of Electrical and Computer Engineering
Seoul National University

Cost-effective, Energy-efficient, and Scalable Storage Computing for Large-scale AI Applications

JAEYOUNG DO, Microsoft Research, USA
VICTOR C. FERREIRA, Federal University of Rio de Janeiro, Brazil
HOSSEIN BOBARSHAD and MAHDI TORABZADEHKASHI, NGD Systems, USA
SIAVASH REZAEI and ALI HEYDARIGORJI, University of California, Irvine, USA
DIEGO SOUZA, Wespa Intelligent Systems
BRUNNO F. GOLDSTEIN and LEANDRO SANTIAGO, Federal University of Rio de Janeiro
MIN SOO KIM, University of California, Irvine, USA
PRISCILA M. V. LIMA and FELIPE M. G. FRANÇA, Federal University of Rio de Janeiro, Brazil
VLADIMIR ALVES, NGD Systems, USA

The growing volume of data produced continuously in the Cloud and at the Edge poses significant challenges for large-scale AI applications to extract and learn useful information from the data in a timely and efficient way. The goal of this article is to explore the use of computational storage to address such challenges by distributed near-data processing. We describe Newport, a high-performance and energy-efficient computational storage developed for realizing the full potential of in-storage processing. To the best of our knowledge, Newport is the first commodity SSD that can be configured to run a server-like operating system, greatly minimizing the effort for creating and maintaining applications running inside the storage. We analyze the benefits of using Newport by running complex AI applications such as image similarity search and object tracking on a large visual dataset. The results demonstrate that data-intensive AI workloads can be efficiently parallelized and offloaded, even to a small set of Newport drives with significant performance gains and energy savings. In addition, we introduce a comprehensive taxonomy of existing computational storage solutions together with a realistic cost analysis for high-volume production, giving a good big picture of the economic feasibility of the computational storage technology.

CCS Concepts: • **Information systems** → **Storage architectures**; • **Computer systems organization** → *Distributed architectures*; • **Computing methodologies** → *Artificial intelligence*;

Additional Key Words and Phrases: Computational storage, in-storage processing, solid-state drive, similarity search, neural network, object tracking

From this Lecture You Should Know

1. What is programmable storage, and why and when this idea make sense (and when it does not)
    a. Data reduction, aggregation, filtering
    b. Energy benefits
2. What are different flavor of programmability - hardware (CPUs, FPGAs, languages), software (runtime, compiler, languages), abstractions (RPCs, Flow-based programming, or virtual files)
3. The basic idea behind :
    a. Willow
    b. Smart Queries SSDs
    c. Biscuit
    d. INSIDER

[Optional] Further Reading

- Jaeyoung Do, Sudipta Sengupta, and Steven Swanson. 2019. Programmable solid-state storage in future cloud datacenters. Commun. ACM 62, 6 (June 2019), 54–62.
- Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. Query Processing on Smart SSDs: Opportunities and Challenges. In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, SIGMOD ’13, pages 1221–1230, New York, NY, USA, 2013.
- L. Woods, Z.Istvan, G.Alonso, Ibex: an intelligent storage engine with support for advanced SQL offloading, VLDB 2014.
- M. Sevilla, N. Watkins, I. Jimenez, P. Alvaro, S. Finkelstein, J. LeFevre, C. Maltzahn, "Malacology: A Programmable Storage System", in EuroSys 2017
- 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 Proceedings of the 13th USENIX conference on Operating Systems Design and Implementation (OSDI'18). USENIX Association, USA, 627–643.
- Shuotao Xu, Sungjin Lee, Sang-Woo Jun, Ming Liu,Jamey Hicks, and Arvind. Bluecache: A scalable distributed flash-based key-value store. Proc. VLDB Endow., 10(4):301–312, November 2016
- 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 (ISCA '16).
- 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 (August 2016), 924–935.
- D. Tiwari, S. Boboila, S. Vazhkudai, Y. Kim, X. Ma, P. Desnoyers, and Y. Solihin, “Active flash: Towards energy-efficient, in-situ data analytics on extreme-scale machines,” USENIX FAST 2013.
- Robert Schmid, Max Plauth, Lukas Wenzel, Felix Eberhardt, and Andreas Polze. 2020. Accessible near-storage computing with FPGAs. In <i>Proceedings of the Fifteenth European Conference on Computer Systems</i> (<i>EuroSys '20</i>).
- Kornilios Kourtis, Animesh Trivedi, Nikolas Ioannou, Safe and Efficient Remote Application Code Execution on Disaggregated NVM Storage with eBPF, https://arxiv.org/abs/2002.11528 (2020).
- Corne Lukken, Giulia Frascaria, Animesh Trivedi, ZCSD: a Computational Storage Device over Zoned Namespaces (ZNS) SSDs, 2021.