Storage Systems (StoSys) XM_0092

Lecture 8: Programmable Storage

Animesh Trivedi

Autumn 2023, Period 1

VU 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
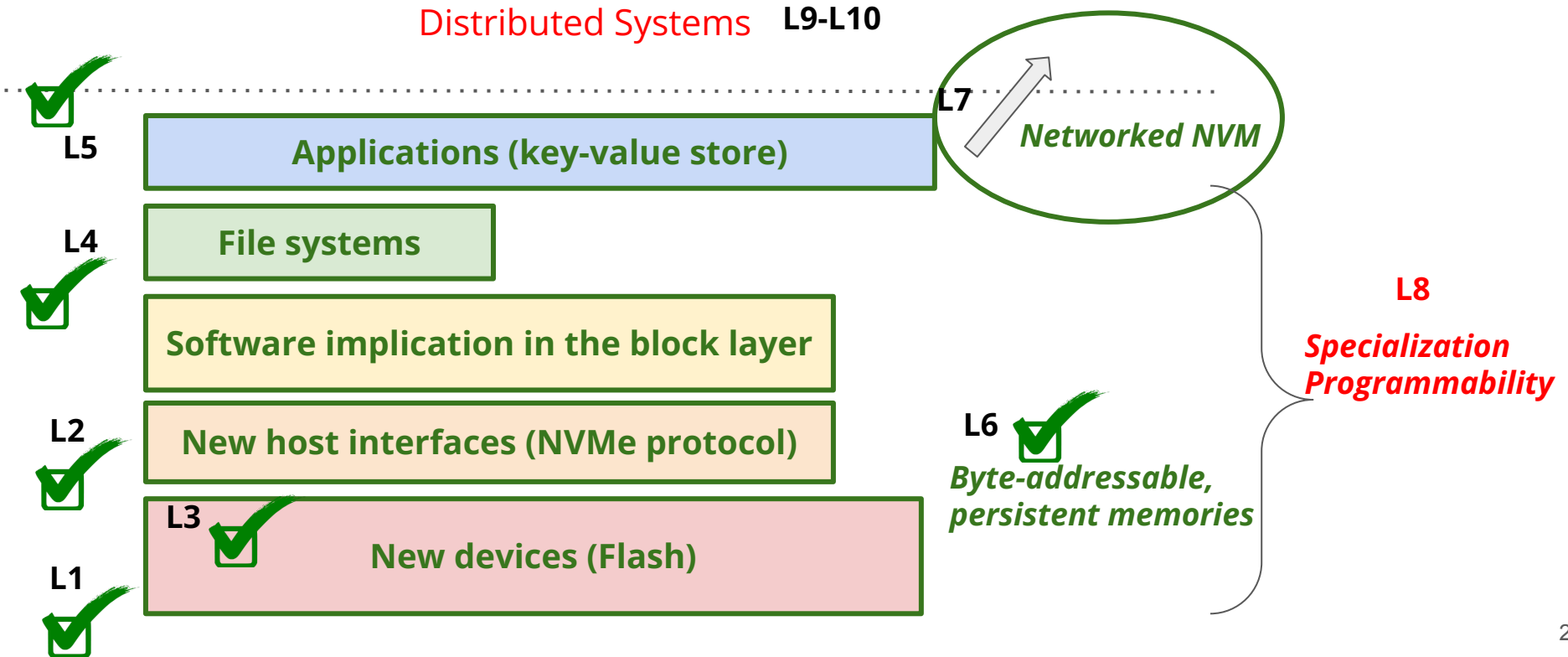
Any Guesses?

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)
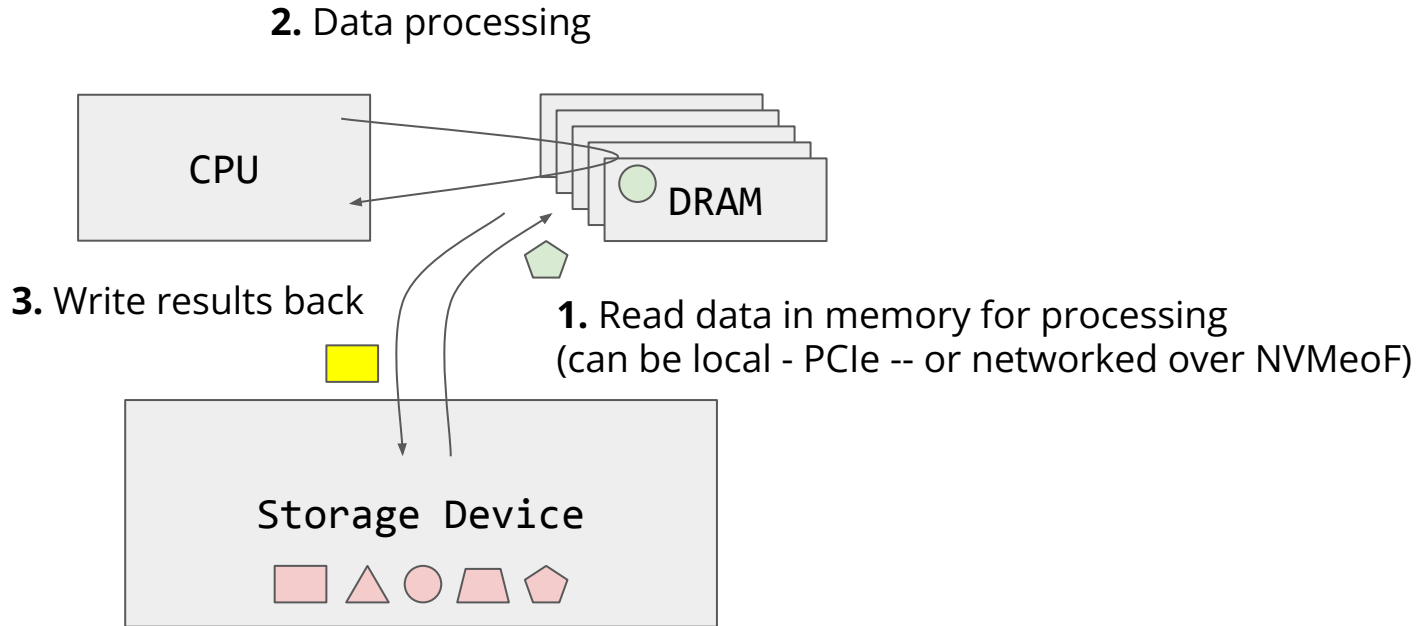
Storage Device

A basic model of how storage and data processing is organized typically
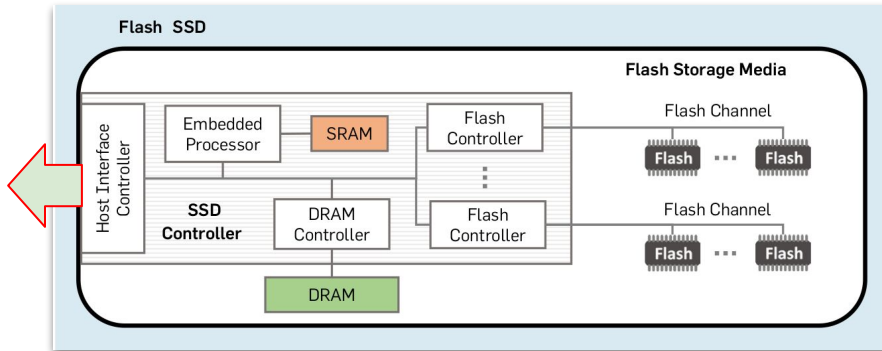**What are the challenges here?**

Key Challenge - Data Movement Wall
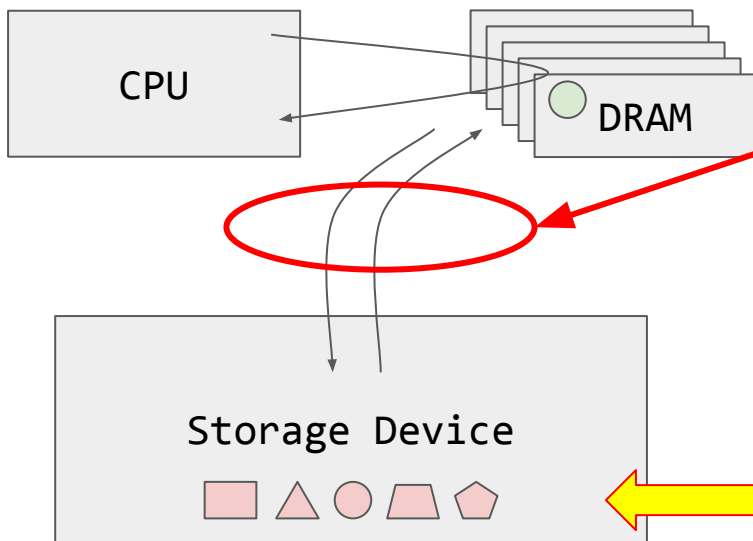
The _network_ (local or external) is a bottleneck.

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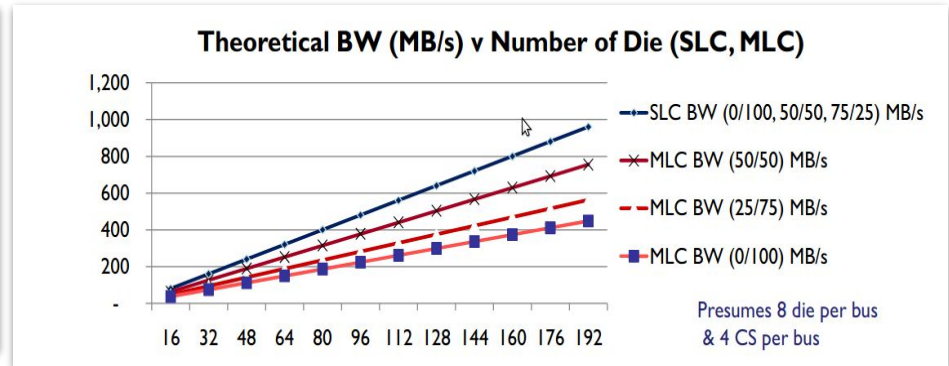
Flash devices consist of multiple independent packages, die, or planes

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

Flash SSD

Host Interface Controller

Embed Proces

SSD Controlle

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
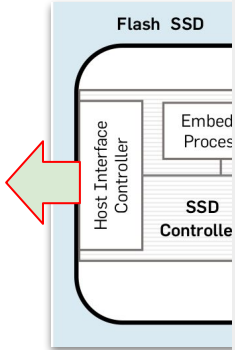
, MLC)

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

We have these in our servers!

Flash devi

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

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
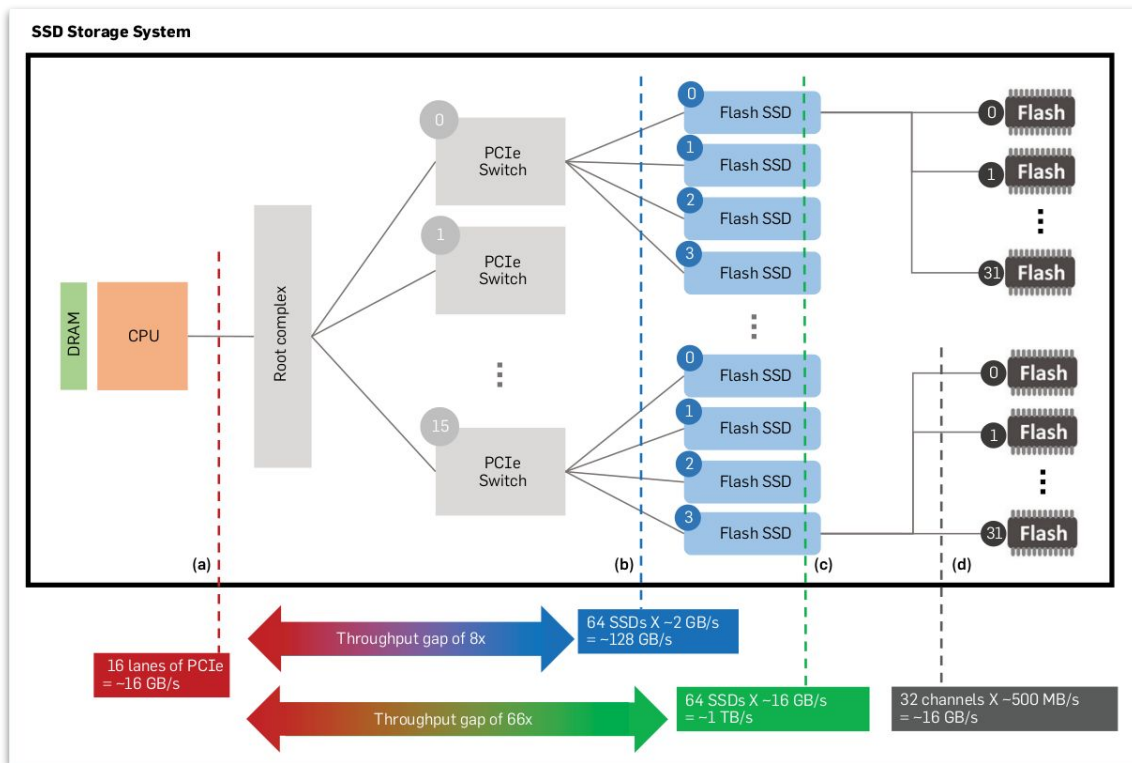
A rack-level SSDs deployment

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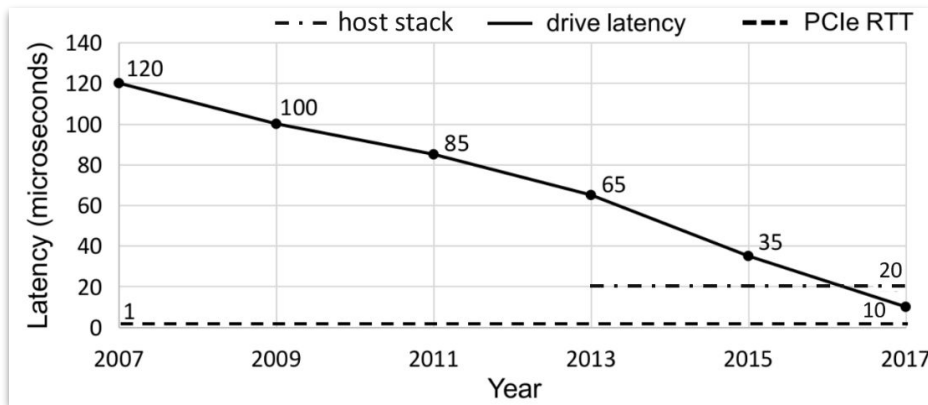
Crossing PCIe bus (v3.0, v4.0) can take time ~1 useconds

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?


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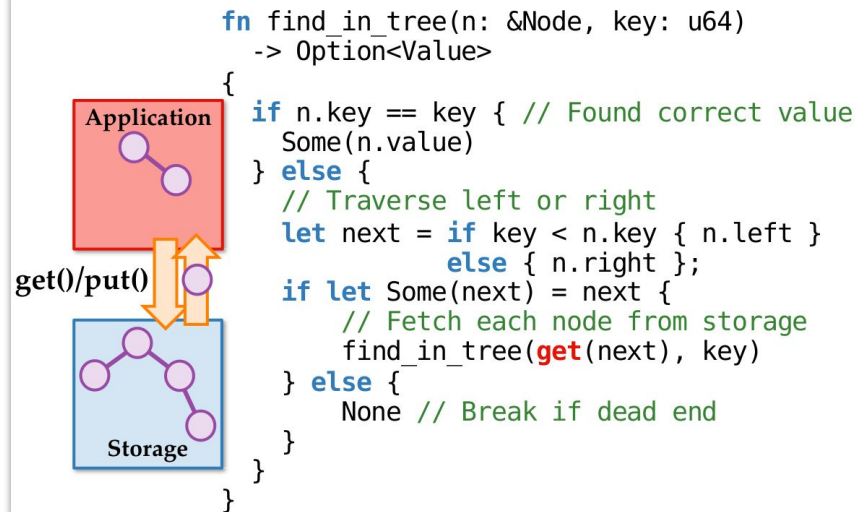
Kulkarni, Splinter: bare-metal extensions for multi-tenant low-latency storage, OSDI 2018.

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).
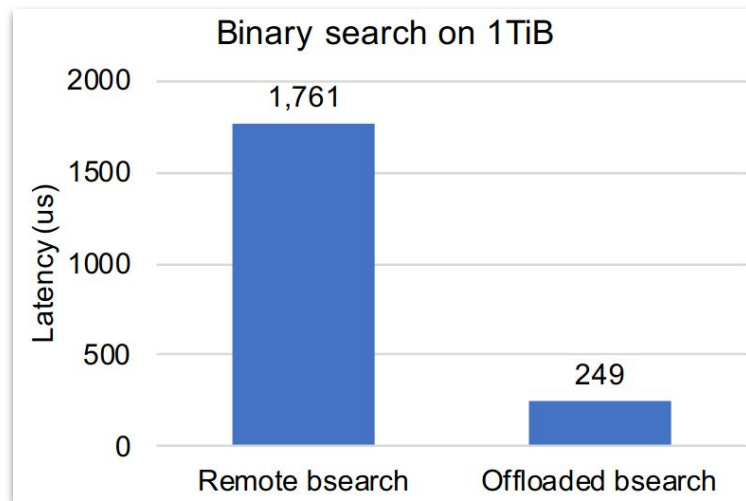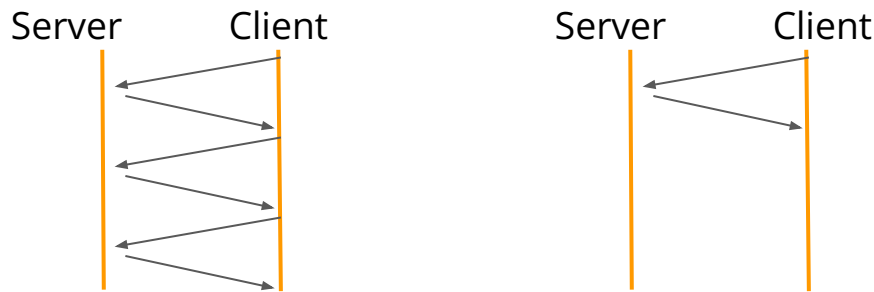
Enter: Programmable Storage

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
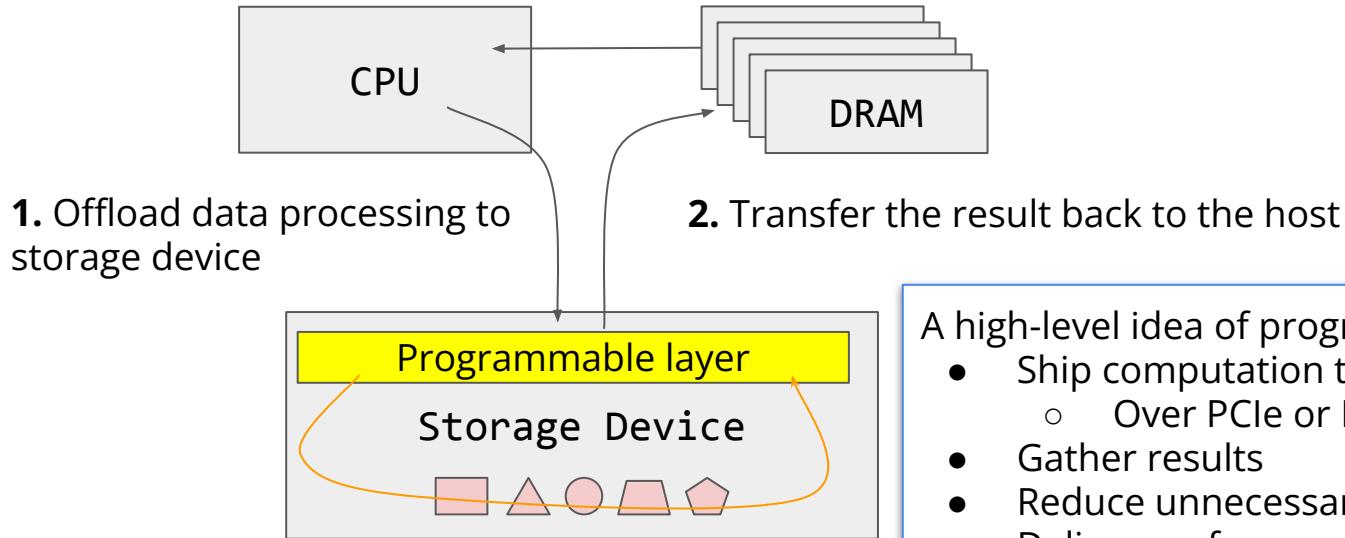
Storage Device

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


```
select (col0 == RED)
```
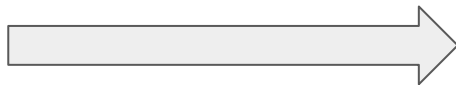
2 x 3 = 6 Bytes

```
project col1, where col2 > 8
```

2 Bytes

All columns = 1 bytes = 15 bytes table
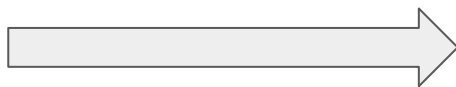
```
average(col1) = 3
max(col2) = 10
```

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 …

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-

17

Key Challenge

[Zhang'12] OS Offload

OS Bypass [Caulfield'12]

[Caulfield'12] Virtualization

[Bhaskaran'13]
[Saxena'12]

Caching

Specialized SSDs

[Kang'13]
[Coburn'13]
[Prabhakaran'08]

Transaction Support

Query Processing

[Do'13]
[Kang'13]

[Wang'14]
[De'13]

Key Value Store

Novel IO Abstractions

[Balakrishnan'12]
[Huang'12]
[Josephson'10]
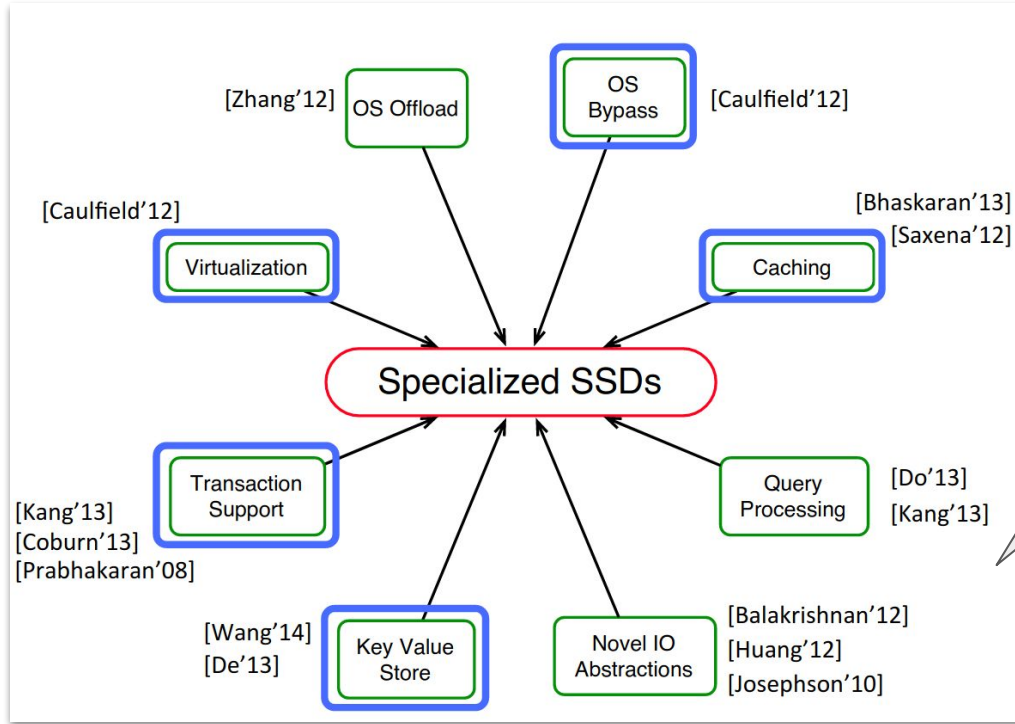
*How to build a usable abstraction to build these multiple **specialized** applications?*

Why specalization?

Willow Architecture

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

- Contains **Storage Processor Unit (SPUs)**
  - that process requests for their attached NVM storage
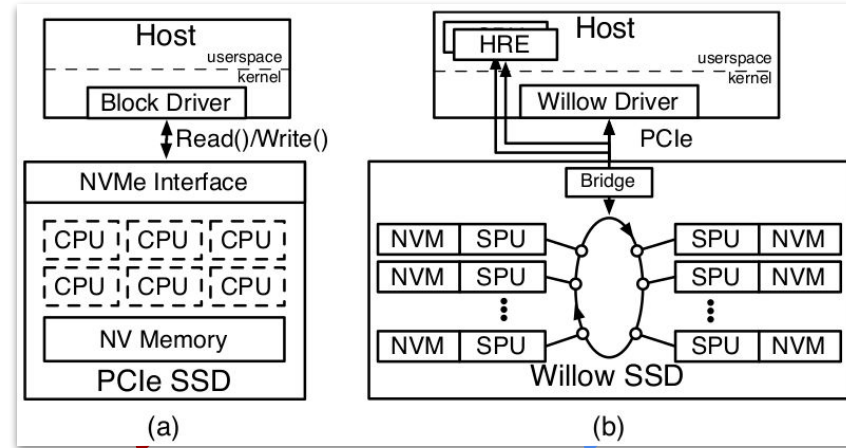
Normal NVMe SSD   Willow SSD (uses own PCIe protocol)

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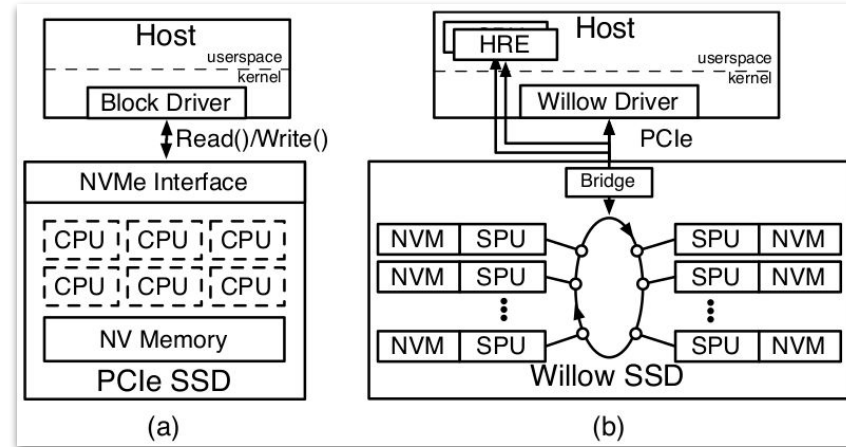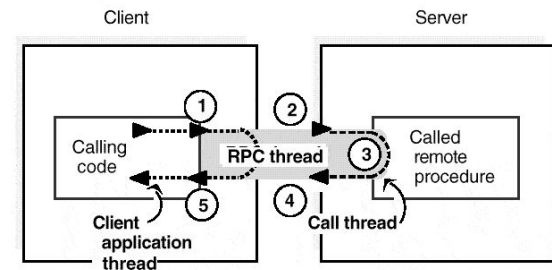
So how do these HREs, SPUs work together to offer a programmable SSD?

Remote procedure call

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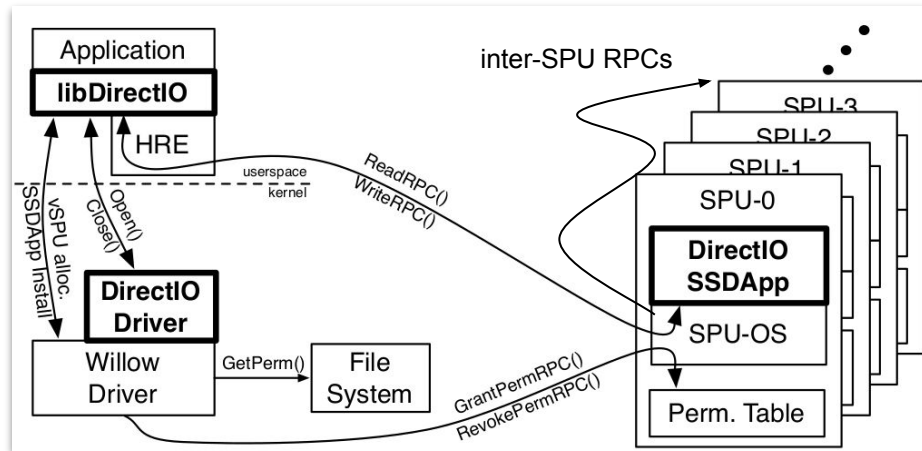
Here in the figure (example): Design for a Direct-Access Storage

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
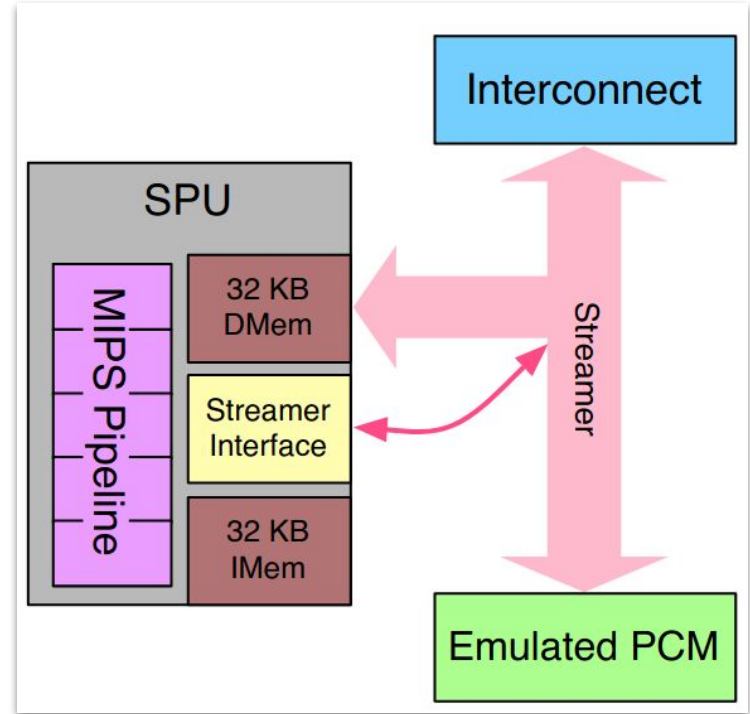
Protection and Sharing Features

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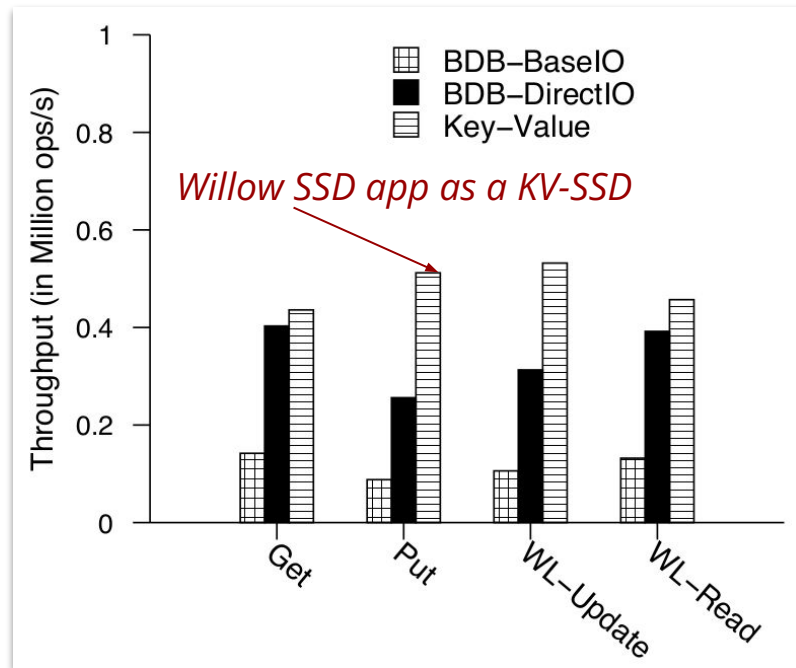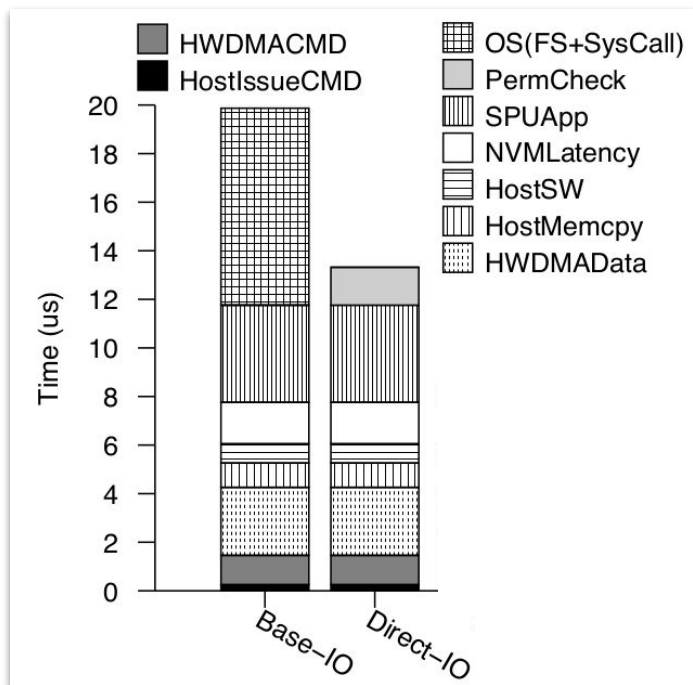
- Direct I/O helps to reduce FS + syscall overheads
- Key-value on Willow (RPC) can improve performance from 8% - 4.8x

This RPC-based Design - Flexibility

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

Relational Data Processing Frameworks (why?)

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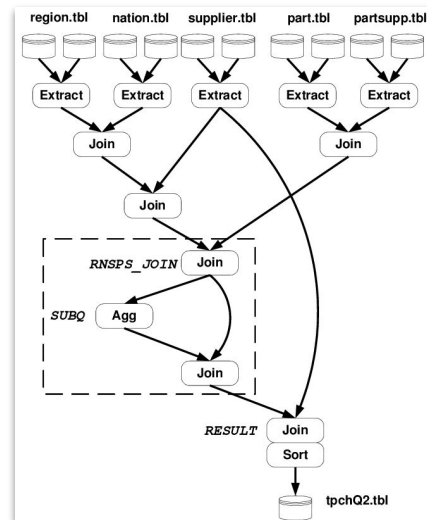
Proposed: implemented the simple selection and aggregation operators into the device FTL and integrated with SQL Server query plans

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!**

29
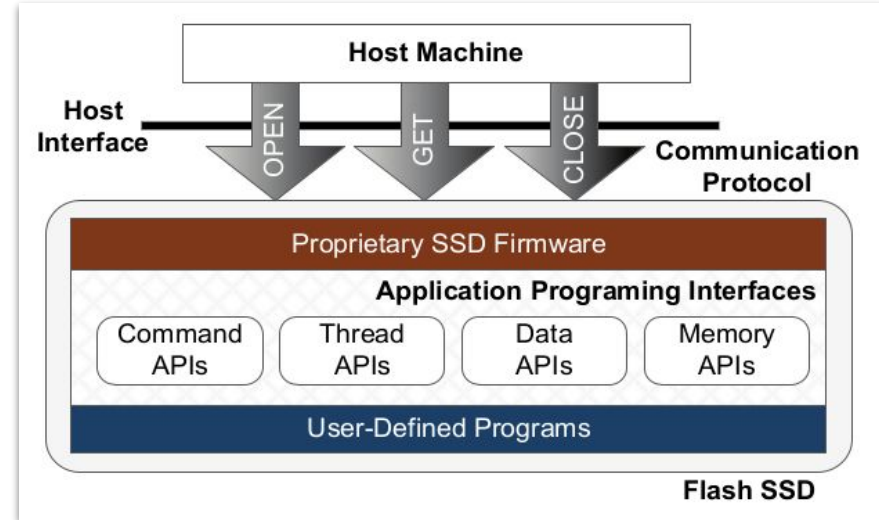
Performance

(a) Elapsed Time
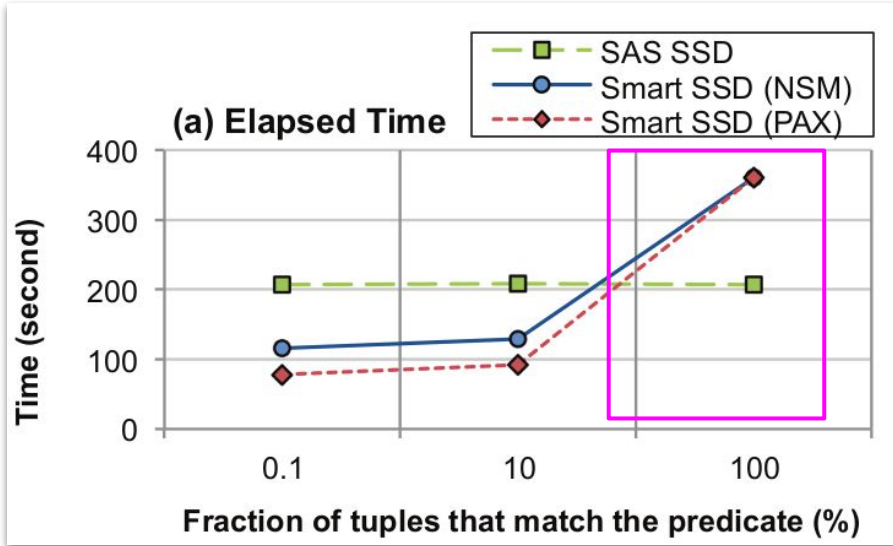
SELECT SecondColumn
FROM SyntheticTable
WHERE FirstColumn < [VALUE]

(a) Elapsed Time
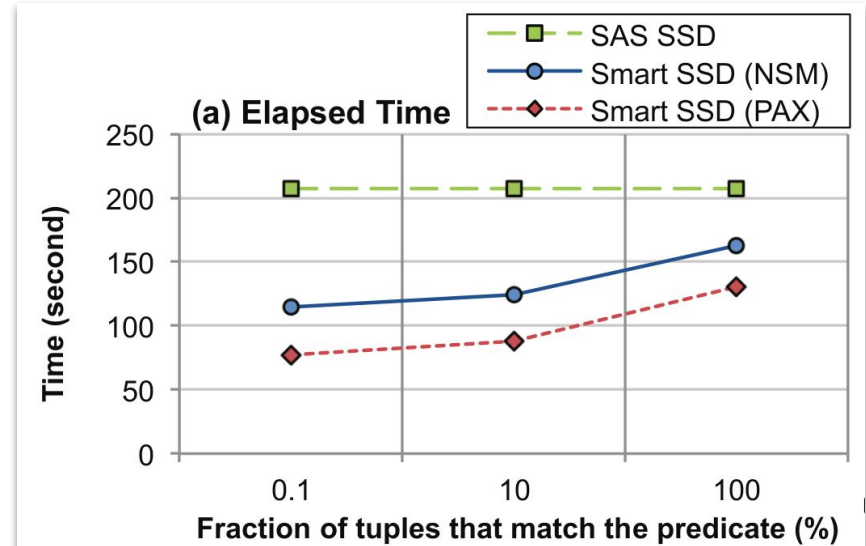
SELECT AVG (SecondColumn)
FROM SyntheticTable
WHERE FirstColumn < [VALUE]

Energy Efficiency
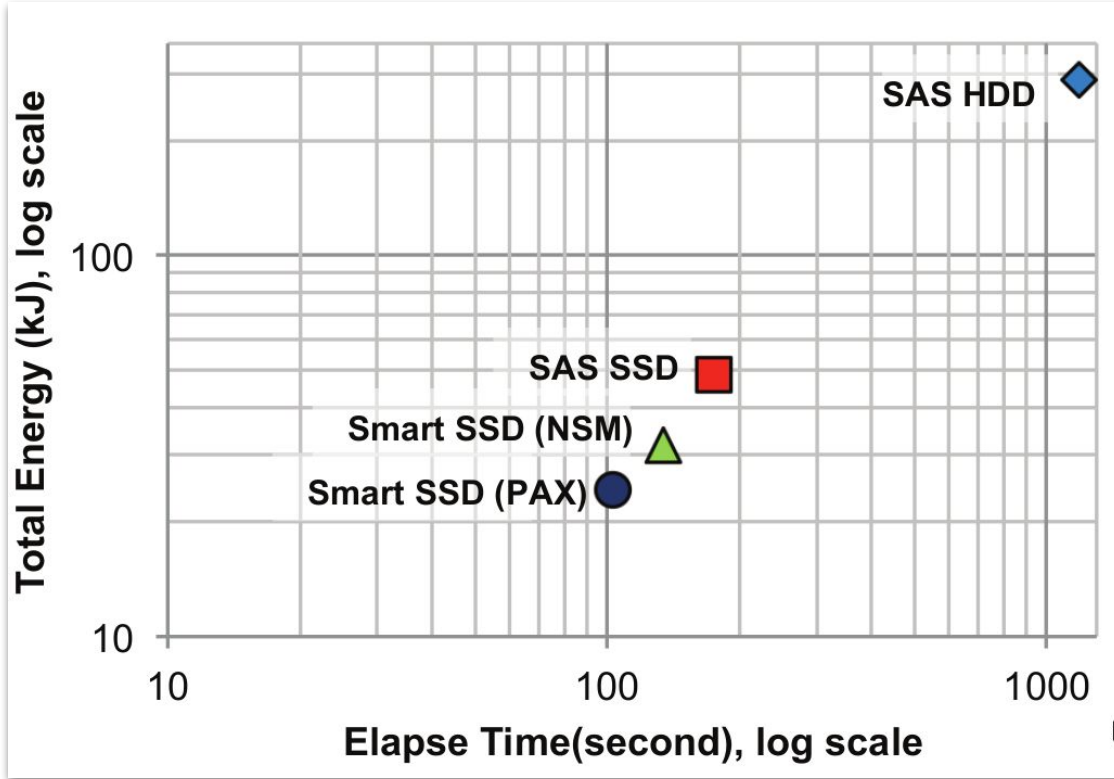
Compared to HDDs, SSDs are more energy efficient

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
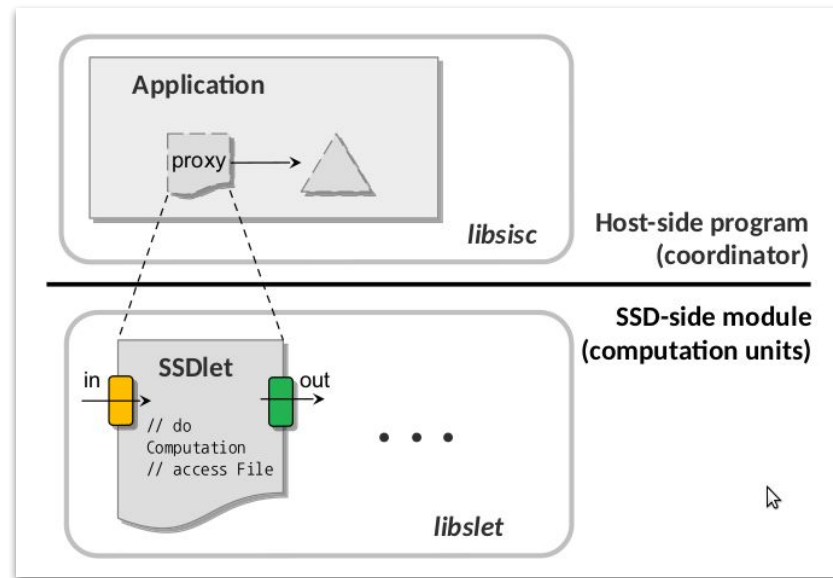
SSDlets and Applications

```
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
}}
```
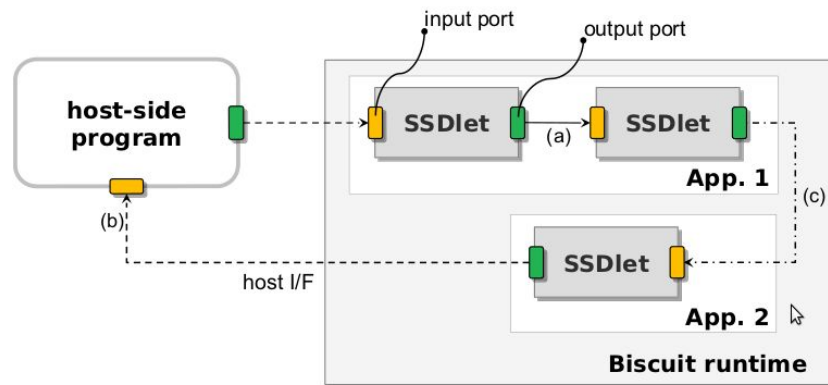

a. Inter-SSDlet (same application)
b. Host-device ports
c. Inter application ports

Important for coordination and staging of data

33

Word Count Application

Flow-based Prog.
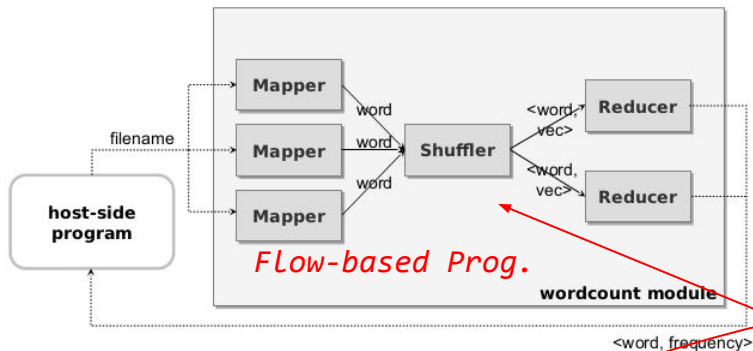
```
class Mapper : public SSDLet<OUT_TYPE<std::pair<std::string, uint32_t>>,
                             ARG_TYPE<File>> {
public:
    void run() {
        auto& file = getArgument<0>();
        FileStream fs(std::move(file));
        auto output = getOutputPort<0>();
        while (true) {
            ...
            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;
}
```

34

Performance

SSD Prototype has: Two ARM Cortex R7 cores @750MHz, L1$, no cache coherence, and Key-based pattern matcher per channel (filtering)
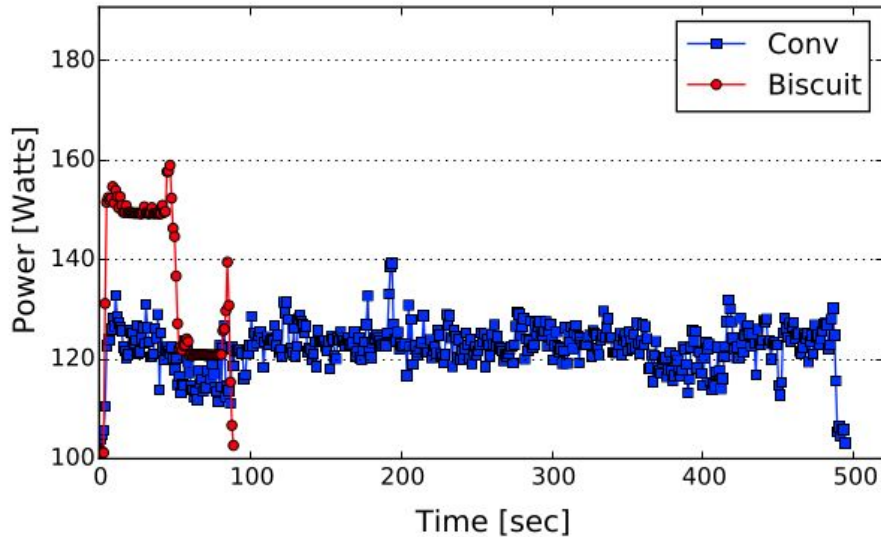
TPC-H Q1, base system energy 103 Watts
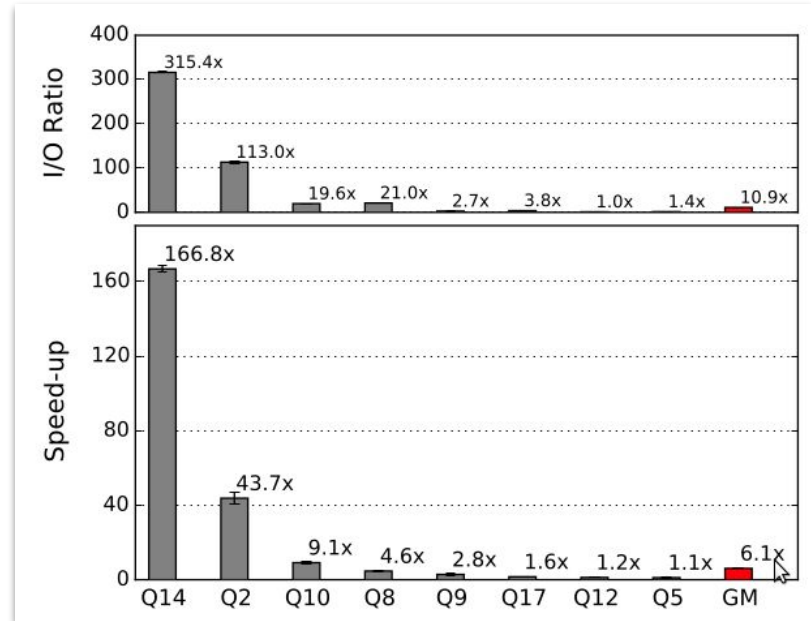
In Summary

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 <u>RPC-based</u> SSD design (with limited memory and multi-tenancy management) - uses SPUs
Smart Query and Biscuit: query processing designs, with operator offloading and <u>flow based programming</u> - 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)

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
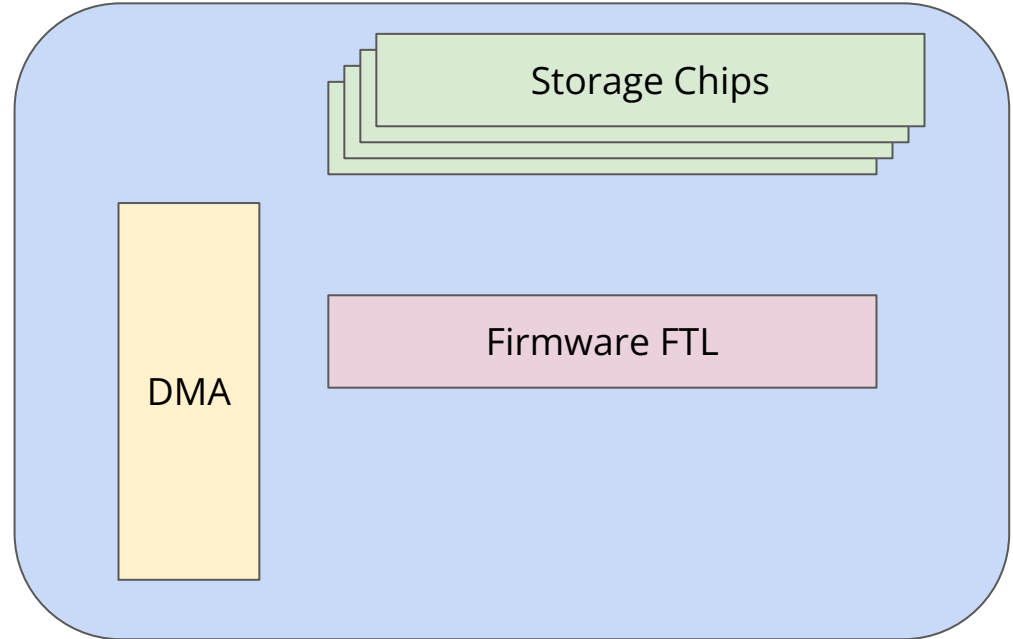2. Might lead to significant code modifications

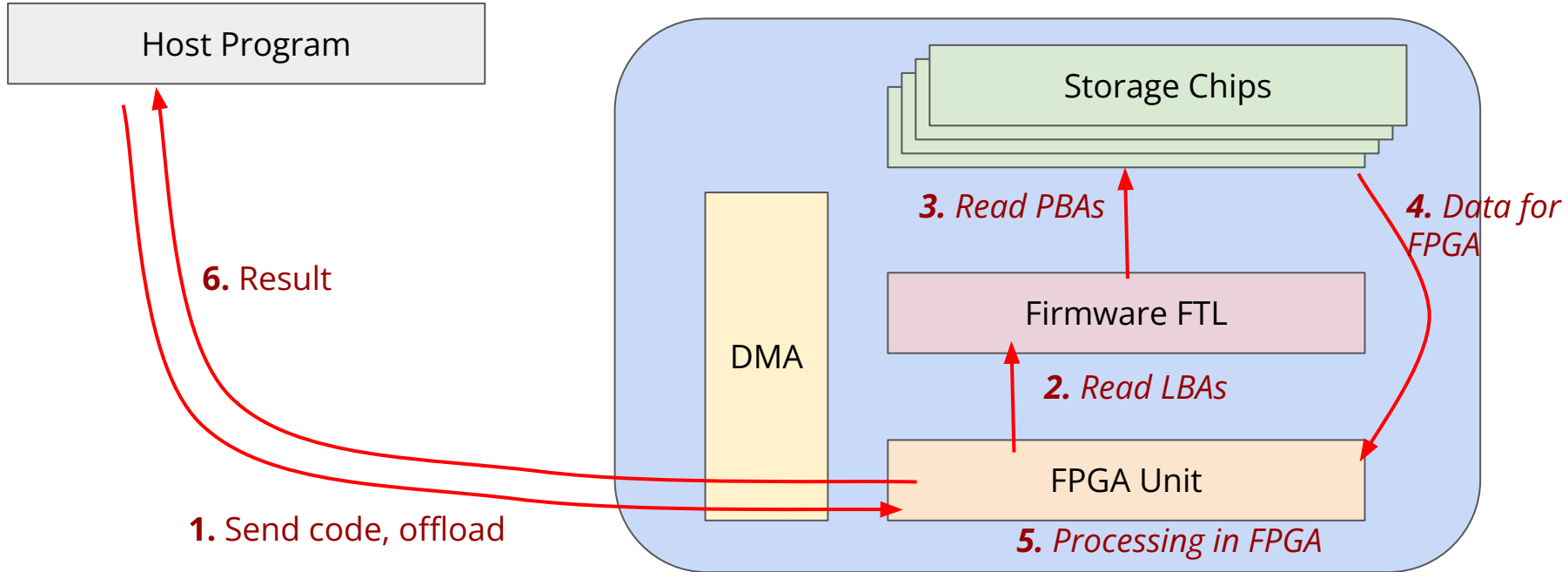| API and Abstractions |
| --- |
| Runtime |
| Hardware |

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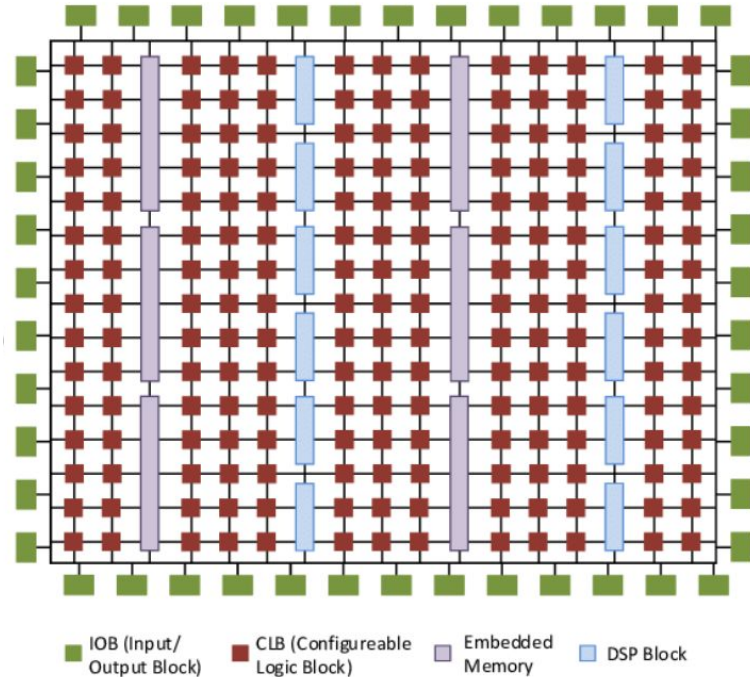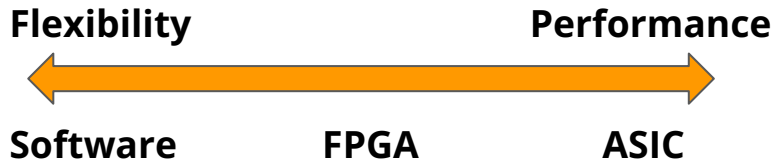
Image credit: B. Ronak et al, *Mapping for Maximum Performance on FPGA DSP Blocks*, https://ieeexplore.ieee.org/document/7229289

40

What is special about FPGA?

instructions

CPU

Memory

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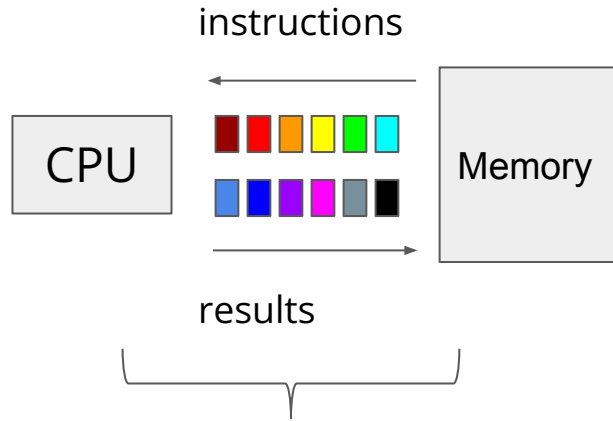
- Distance to memory
- Instruction dependencies
- Programming control units in CPUs
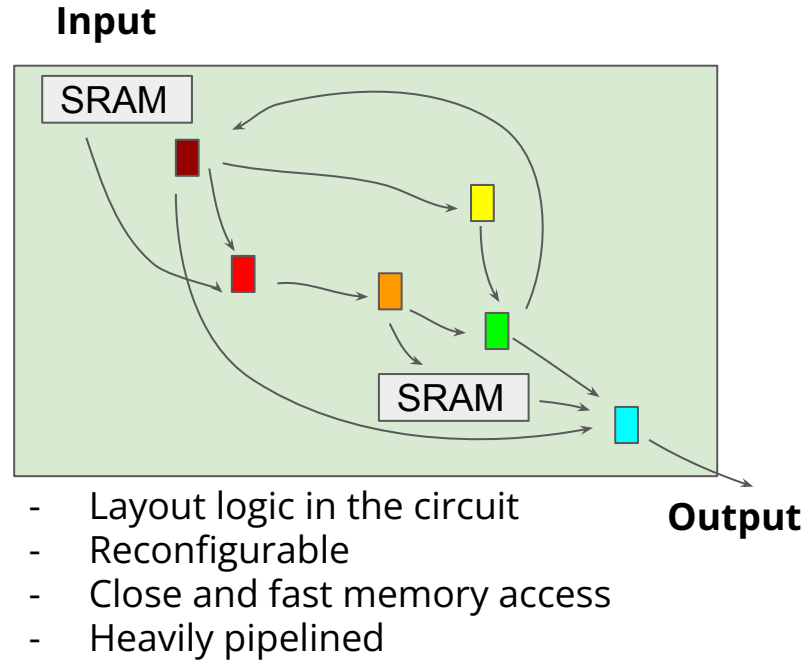
Input

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

Input

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

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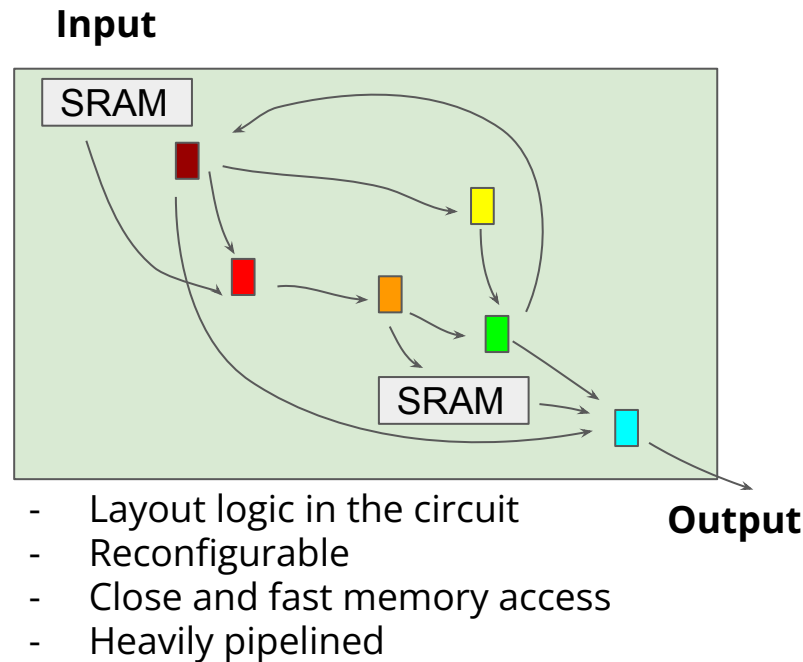
See *Darwin: A Genomics Co-processor Provides up to 15,000X Acceleration on Long Read Assembly.* ASPLOS 2018.

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

API and Abstractions

Runtime

Hardware

43

Programmability needs Support from the Whole Stack

Hardware → *Use FPGA*
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
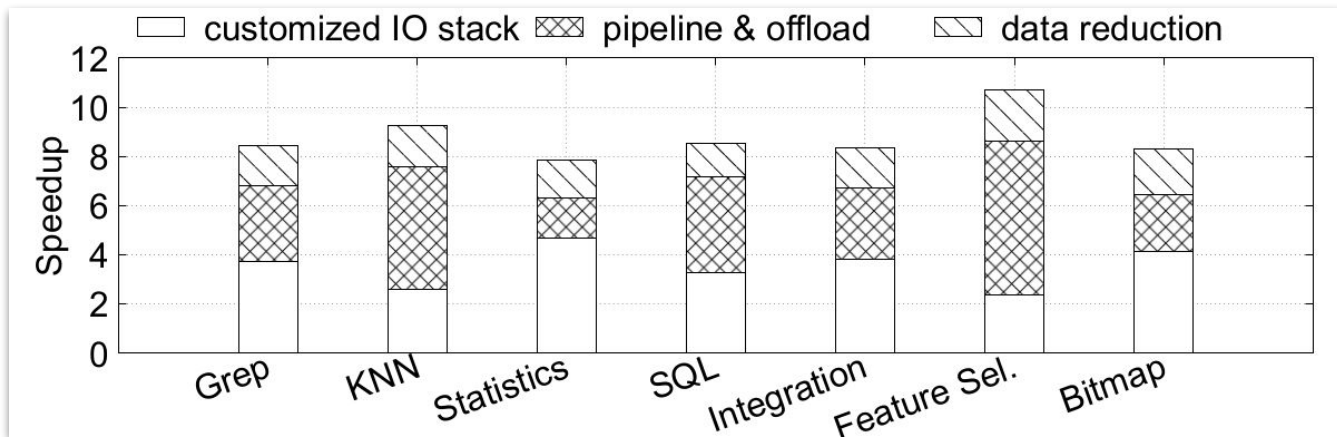2. Might lead to significant code modifications

API and Abstractions
Runtime
Hardware

INSIDER Architecture

Host Program

Conventional SSD

Storage Chips

DMA

Firmware FTL

INSIDER Architecture

INSIDER SSD

- **1.** Send code, offload
- **2.** *Read LBAs*
- **3.** *Read PBAs*
- **4.** *Data for FPGA*
- **5.** *Processing in FPGA*
- **6.** Result

Host Program

DMA

Storage Chips

Firmware FTL

FPGA Unit

INSIDER Architecture

INSIDER SSD

Host Program

Storage Chips

DMA

3. *Read PBAs*

4. *Data for FPGA*

Firmware FTL

6. Result

2. *Read LBAs*

FPGA Unit

1. Send code, offload

5. *Processing in FPGA*
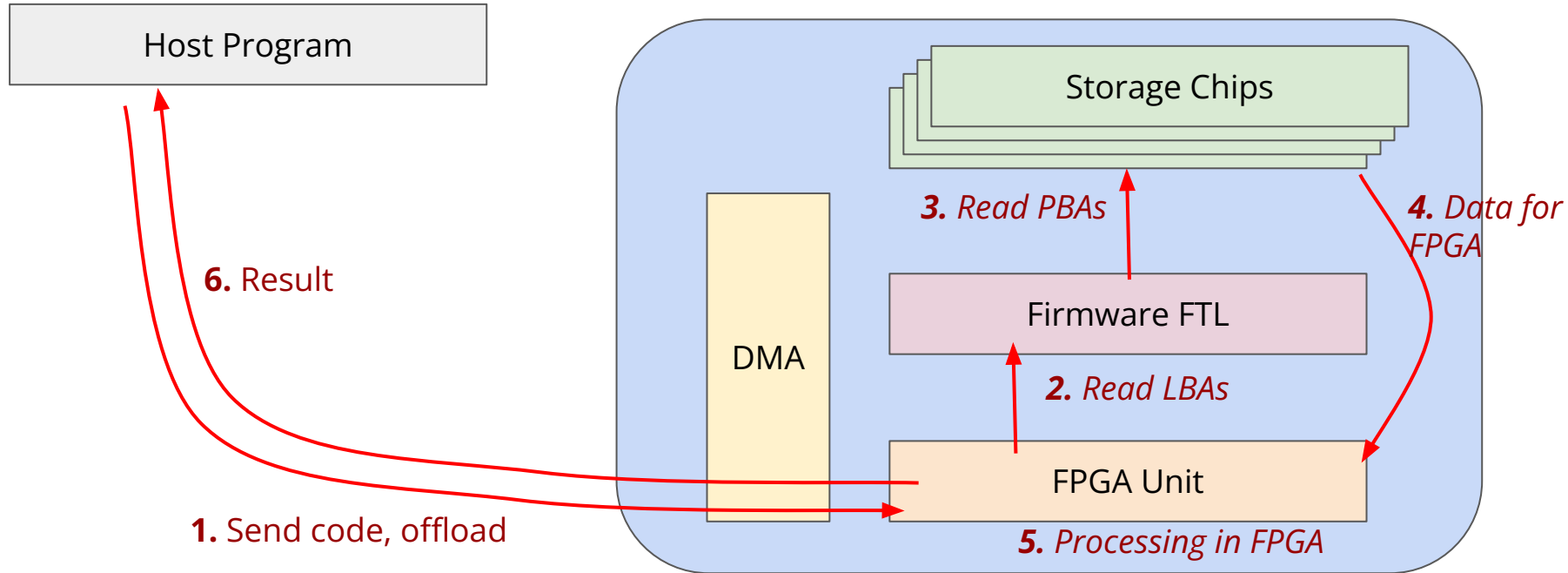
How to make sure a rogue FPGA program is not able to read any arbitrary storage location or write to any location?

INSIDER Architecture

INSIDER SSD

- Host Program
- Storage Chips
- DMA
- Firmware FTL
- FPGA Unit

1. Send code, offload

5. *Processing in FPGA*
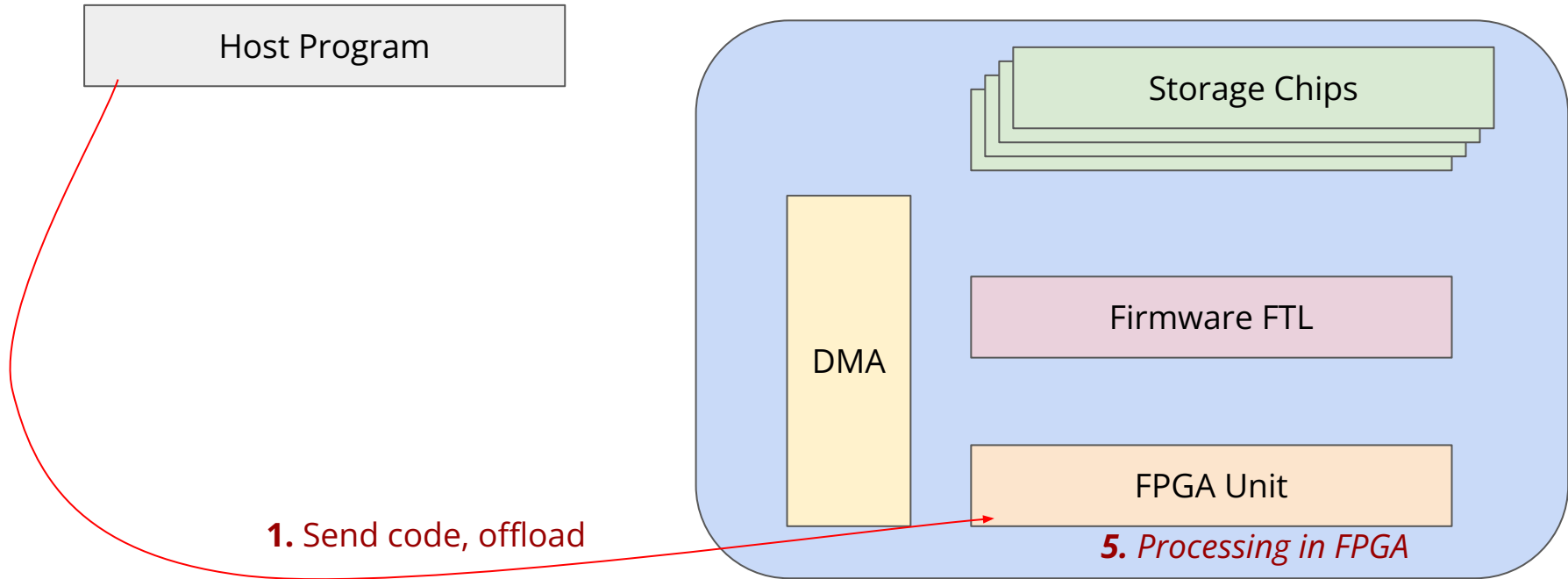
Idea 1: Make FPGA program "**Compute-Only**", hence the program itself cannot issue any r/w ops.

Then how to get data from flash chips for processing?

INSIDER Architecture
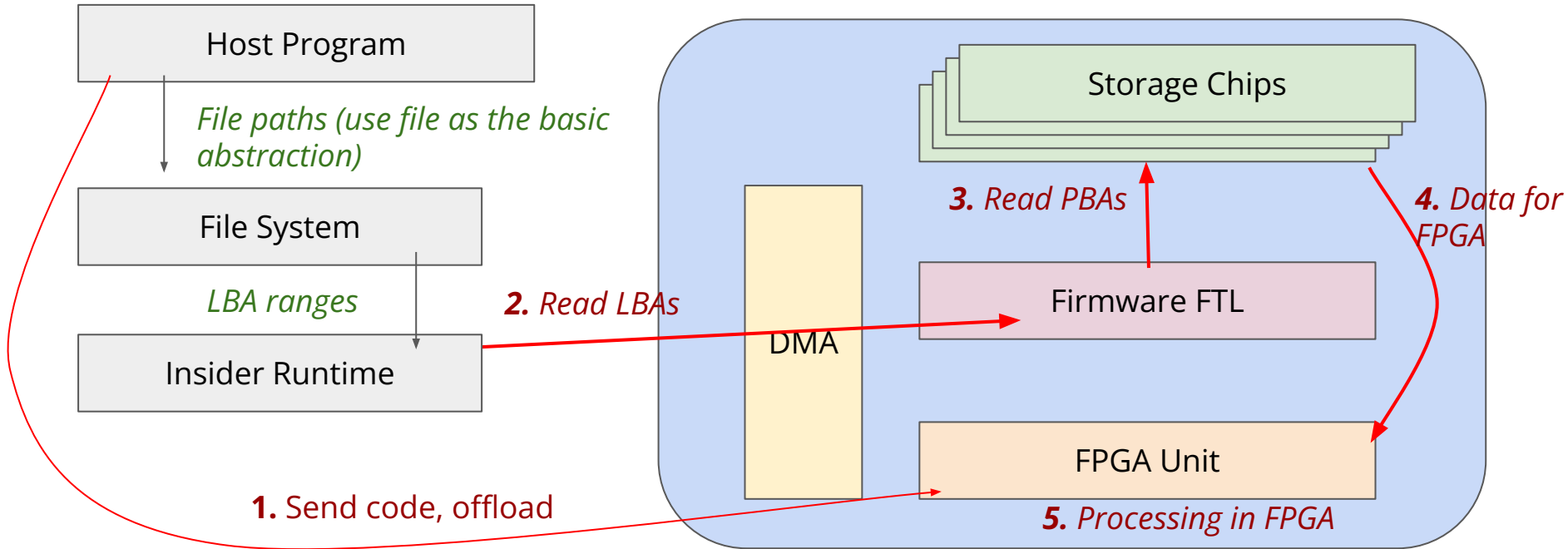
INSIDER SSD

Host Program

File paths (use file as the basic abstraction)

File System

LBA ranges

Insider Runtime

2. *Read LBAs*

DMA

Storage Chips

3. *Read PBAs*

4. *Data for FPGA*

Firmware FTL

FPGA Unit

1. Send code, offload

5. *Processing in FPGA*

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
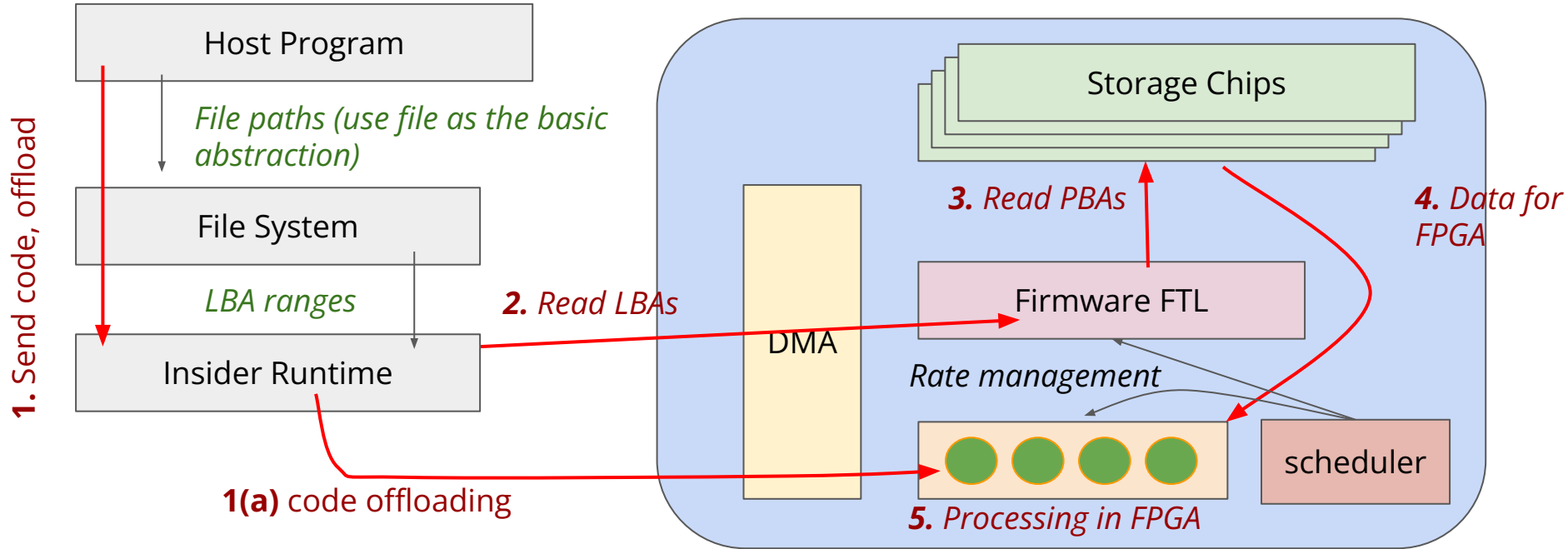
INSIDER SSD

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
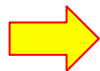
Runtime ➡️ *Compute-only programs with FPGA partitioning*

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

API and Abstractions
Runtime
Hardware

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 :)

Tells INSIDER which files to prep for reading, reserve id

```
// get a virtual file
vfile =

int fd

send_pa

int sz
int sz

// vsync - if written

vclose(fd);
```

...s permissions, and hold ...ssing

...am parameters

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

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!

Synchronize and close the file to release resources

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
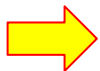
API and Abstractions
→ *Virtual files*
1. New APIs leads to less familiarity with developers
2. Might lead to significant code modifications

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

56

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

User 1

User 2

User 3

X

Programmable Storage
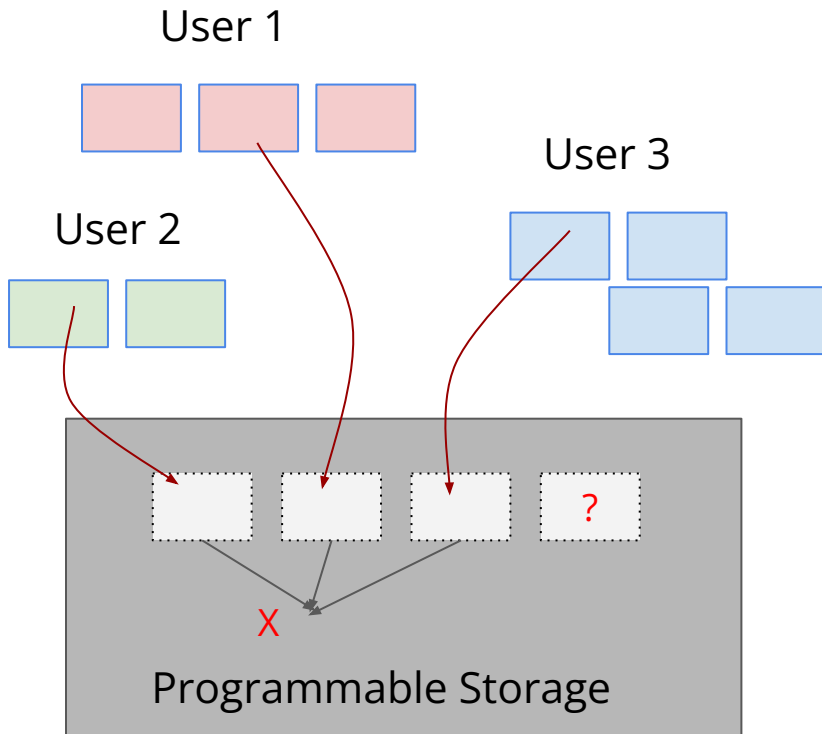
Installing and running user-provided extensions safely

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

59

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:

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

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

eBPF-based Kernel Programming

Figure 4: XRP architecture.

62

Computation Storage: New Emerging Standard

- 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
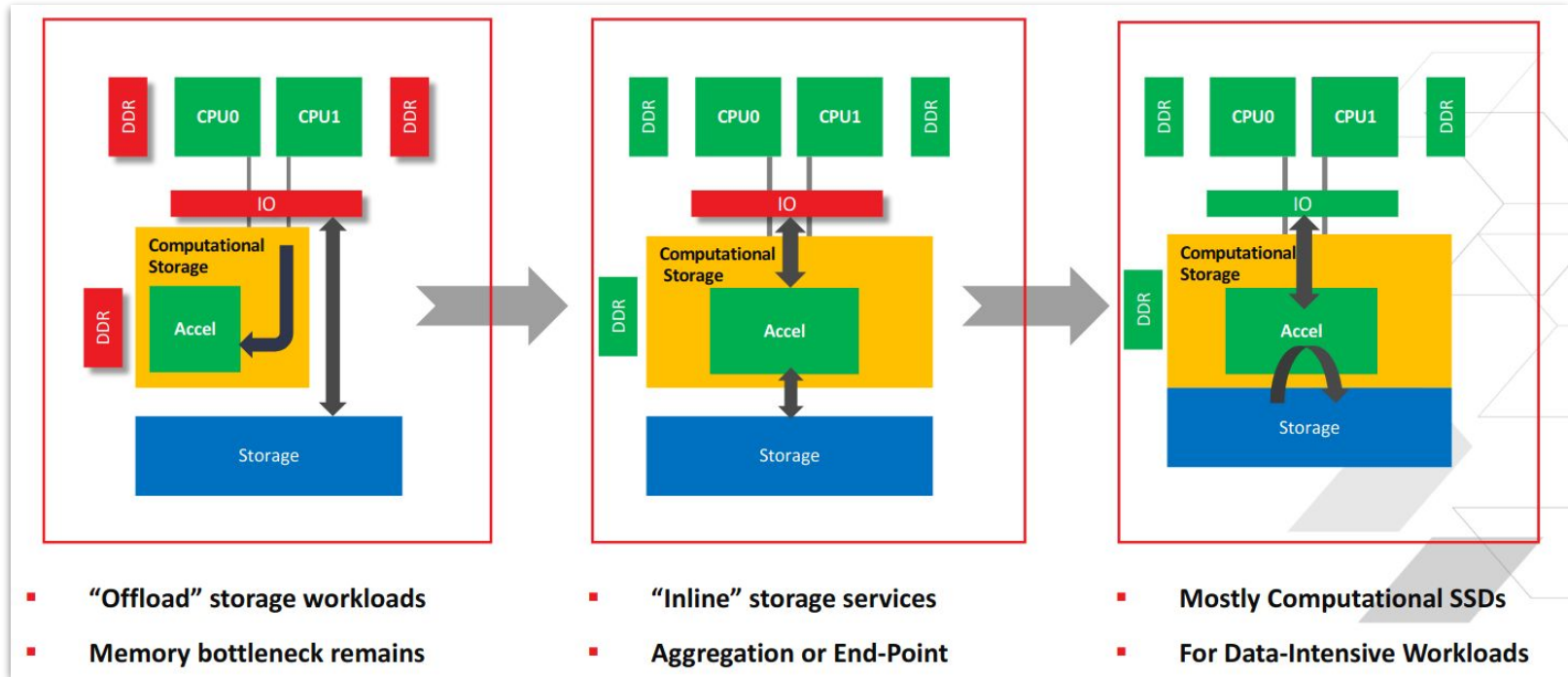
So When Does Using CSD Makes Sense?

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 <u>Database</u> 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*?

66

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

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.

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

67

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.