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 unprecedented scale of data volume. Large corporations like Facebook have stored over 300 PB of data at their warehouse, with an incoming daily data rate of 600 TB [62] in 2014. A recent warehouse-scale profiling [42] shows that data analytics has become a major workload in the datacenter. Operating on such a data scale is a huge challenge for system designers. Thus, designing an efficient system for massive data analytics has increasingly become a topic of major importance [23, 27].

The drive I/O speed plays an important role in the overall data processing efficiency—even for the in-memory computing framework [68]. Meanwhile, for decades the improvement 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 nonvolatile 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 between host and drive. This approach requires a significant programming effort and is not portable across different file systems. 3) **Lack of crucial system support.** In practice, the drive is shared among multiple processes. Unfortunately, existing work assumes a monopolized scenario; the isolation and resource scheduling between different ISC tasks are not explored. Additionally, data protection is an important concern; without it, offloaded programs can issue arbitrary R/W requests to operate on unwarranted data.

To overcome these problems, we present INSIDER, a full-stack redesigned storage system which achieves the following design goals.

*Corresponding author.

Saturate high drive rate. INSIDER introduces the FPGA-based reconfigurable controller as the ISC unit which is able to process the drive data at the line speed while retaining programmability (§3.1). The data reduction or the amplification pattern from the legacy code are extracted into a drive program which could be dynamically loaded into the drive controller on demand (§3.2.2). To increase the end-to-end throughput, INSIDER transparently constructs a system-level pipeline which overlaps drive access time, drive computing time, bus data transferring time and host computing time (§3.5).

Provide effective abstractions. INSIDER aims to provide effective abstractions to lower the barrier for users to leverage the benefits of ISC. On the host side, we provide virtual file abstraction which abstracts ISC as file operations to hide the existence of the underlying ISC unit (§3.3). On the drive side, we provide a compute-only abstraction for the offloaded task so that drive programmers can focus on describing the computation logic; the details of underlying data movement between different system components are hidden (§3.4).

Provide necessary system support. INSIDER separates the control and data planes (§3.2.1). The control plane is trusted and not user-programmable. It takes the responsibilities of issuing drive access requests. By performing the safety check in the control plane, we *protect* the data from being accessed by unwarranted drive programs. The ISC unit, which sits on the data plane, only intercepts and processes the data between the drive DMA unit and storage chips. This compute-only interface provides an *isolated* environment for drive programs whose execution will not harm other system components in the control plane. The execution of different drive programs is hardware-isolated into different portions of FPGA resources. INSIDER provides an *adaptive drive bandwidth scheduler* which monitors the data processing rates of different programs and provides this feedback to the control plane to adjust the issuing rates of drive requests accordingly (§3.6).

High cost efficiency. We define cost efficiency as the effective data processing rate per dollar. INSIDER introduces a new hardware component into the drive. Thus, it is critical to validate the motivation by showing that INSIDER can achieve not only better performance, but also better cost efficiency when compared to the existing work.

We build an INSIDER drive prototype (§4.1), and implement its corresponding software stack, including compiler, host-side runtime library and Linux kernel drivers (§4.2). We could mount the PCIe-based INSIDER drive as a normal storage device in Linux and install any file system upon it. We use a set of widely used workloads in the end-to-end system evaluation. The experiment results can be highlighted as follows: 1) INSIDER greatly alleviates the system interconnection bottleneck. It achieves 7X∼11X performance compared with the host-only traditional system (§5.2.1). In most cases, it achieves the optimal performance (§5.2.2). 2) INSIDER achieves 1X∼58X (12X on average) performance and 2X∼150X (31X on average) cost efficiency compared to the ARM-based ISC system (§5.5). 3) INSIDER only requires moderate programming efforts to implement applications (§5.2.3). 4) INSIDER simultaneously supports multiple offloaded tasks, and it can enforce resource scheduling adaptively and transparently (§5.3).

2 Background and Related Work

2.1 Emerging Storage Devices: Opportunities and Challenges

Traditionally, drives are regarded as a slow device for the secondary persistent storage, which has the significantly higher access latency (in ms scale) and lower bandwidth (in hundreds of MB per second) compared to DRAM. Based on this, the classical architecture for storage data processing presented in Fig. 3a has met users' performance requirements for decades. The underlying assumptions of this architecture are: 1) The interconnection performance is higher than the drive performance. 2) The execution speeds of host-side I/O stacks, including the block device driver, I/O scheduler, generic block layer and file system, are much faster than the drive access. While these were true in the era of the hard-disk drive, the landscape has totally changed in recent years. The bandwidth and latency of storage drives have improved significantly within the past decade (see Fig. 1 and Fig. 2). However, meanwhile, the evolution of the interconnection bus remains stagnant: there have been only two updates between 2007 and 2017.[1]

For the state-of-the-art platform, PCIe Gen3 is adopted as the interconnection [66], which is at 1 GB/s bidirectional transmission speed per link. Due to the storage density[2] and due to cost constraints, the four-lane link is most commonly used (e.g., commercial drive products from Intel [7] and Samsung [14]), which implies the 4 GB/s duplex interconnection bandwidth. However, this could be easily transcended by the internal bandwidth of the modern drive [24, 33, 34]. Their internal storage units are composed of multiple channels, and each channel equips multiple banks. Different from the serial external interconnection, this two-level architecture is able to provide scalable internal drive bandwidth—a sixteen-channel, single-bank SSD (which is fairly common now) can easily reach 6.4 GB/s bandwidth [46]. The growing mismatch between the internal and external bandwidth prevents us from fully utilizing the drive performance. The mismatch gets worse with the advent of 3D-stacked NVM-based storage which can deliver comparable bandwidth with DRAM [35, 54]. On the other hand, the end of Dennard scaling slows down the performance improvement of CPU, making it unable to catch the ever-increasing drive speed. The long-established block layer is now reported to be a major

[1] Although the specification of PCIe Gen 4 was finalized at the end of 2017, there is usually a two-year waiting period for the corresponding motherboard to be available in the market. Currently there is no motherboard supporting PCIe 4.0, and we do not include it in the figure.

[2] CPU has limited PCIe slots (e.g., 40 lanes for an Xeon CPU) exposed due to the pin constraint. Using more lanes per drive leads to low storage density. In practice, a data center node equips 10 or even more storage drives.

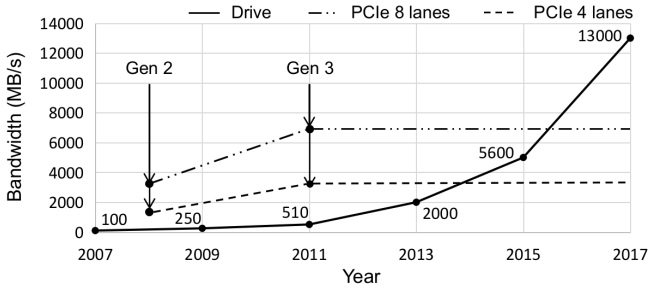


Figure 1: The bandwidth evolution of storage drives. Data are taken from [18] [4] [1] [8] [13] [16] in chronological order. This figure also presents the bandwidth evolution of PCIe (in 4 lanes and 8 lanes).

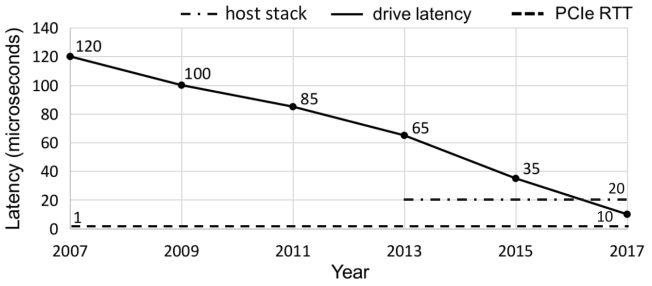


Figure 2: The latency evolution of storage drives. Data are taken from [15] [3] [9] [8] [10] [11] in chronological order. Meanwhile the latency of the host storage stack is about 20 μs [32], and the PCIe RTT (which includes the latency of bus and controller) is about 1 μs [50].

bottleneck of the storage system [28], and less than half raw drive speed is delivered to the end user [31, 56].

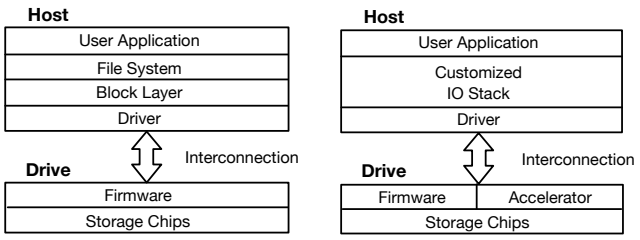
On the latency side, the state-of-the-art SSD delivers R/W latency below 10 μs [14], and the future NVM-based storage can potentially deliver sub-microsecond latency [30]. Meanwhile, the round-trip latency of PCIe still remains at about 1 μs [50], and the host-side I/O stack latency is even more than 20 μs [31, 32]. This implies that the latencies of host-side I/O stack are going to dominate the end-to-end latency.

In summary, the emerging storage devices bring hope—along with great challenges—to system designers. Unless the "data movement wall" is surpassed, high storage performance will not be delivered to end users.

2.2 Review of In-Storage Computing

In order to address the above system bottlenecks, the *in-storage computing (ISC)* architecture is proposed [48, 61], shown in Fig. 3b. In ISC, the host partially offloads tasks into the *in-storage accelerator* which can take advantage of the higher internal drive performance but is relatively less powerful compared to the full-fledged host CPU. For tasks that contain computation patterns like filtering or reduction, the output data volume of the accelerator, which will be transferred back to host via interconnection, is greatly reduced so that bottlenecks of interconnection and host I/O stacks are alleviated [33, 57, 63]. With customized IO stacks, the system bypasses the traditional OS storage stacks to achieve lower latency. With ISC, considerable performance and energy gains are achieved [25].

Historically, the idea of ISC was proposed two decades



(a) Classical. **(b)** In-storage computing.

Figure 3: Drive data processing architecture.

ago [43, 59], but did not become popular at that time. The reasons are twofold: 1) For the technology at that time, it was too expensive to integrate computing unit with storage drive; 2) More importantly, the drive performance was much lower than the performance of the host/drive bus, so in-storage computing could only bring limited performance benefits.

However, with the great improvement of VLSI technology in the past two decades, integration expense is greatly reduced. In fact, currently, every high-end SSD equips one or even multiple embedded CPUs. Meanwhile, the drive performance consistently increases, and goes beyond the performance of host/drive interconnection (see Fig. 1 and Fig. 2). This gap validates the motivation of ISC. Therefore, in recent years, we witness the revival of in-storage computing [49]. Most of the recent work focuses on offloading user-defined tasks to drive-embedded CPUs, which are originally designed to execute the firmware code, e.g., flash translation layer (FTL). However, this approach faces the following limitations.

Limited computing capability. Drive-embedded CPUs are usually fairly weak ARM cores which can be up to 100X slower compared to the host-side CPU (Table 3 in [63]). Based on this, offloading tasks to drive may lead to a decreased data processing speed by a factor of tens [33]. A recent work [48] proposes a dynamic workload scheduler to partition tasks between host and drive ARM processor. However, the optimal point they found is very close to the case in which all the tasks are executed at the host; this emphasizes that embedded cores are too feeble to provide a distinguishable speedup.

No effective programming abstractions. Existing work does not provide effective abstractions for programmers. On the host side, they 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. On the drive side, the drive program either manages the drive as a bare block device without a file system (FS), e.g., [48], or has to carefully cooperate with the host FS to access the correct file data, e.g., [32]. This distracts drive programmers from describing the computing logic and may not be portable across different FSes. It is important to provide effective abstractions to lower the barrier for users to leverage the benefits of ISC [26].

Lack of crucial system support. Naturally, the drive is shared among multiple processes, which implies the scenario of concurrently executing multiple ISC applications. This is

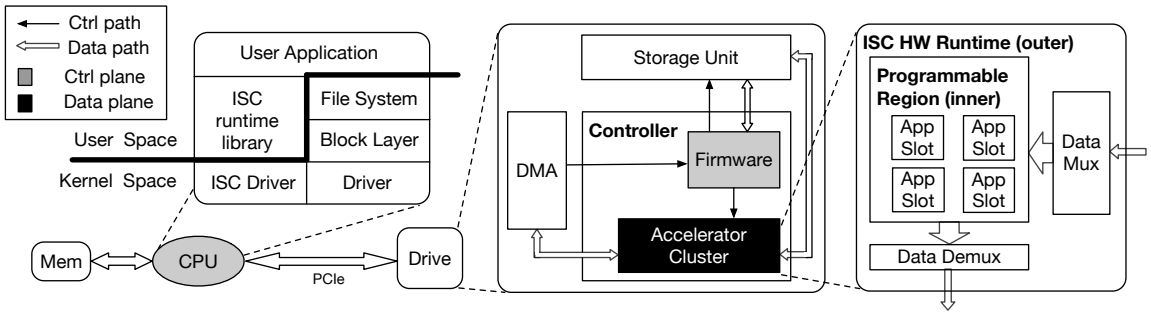


Figure 4: System architecture of INSIDER. INSIDER separates the control plane and the data plane; accelerator cluster sits on the data plane (black box) while the host-side library and drive-side firmware sit on the control plane (gray box).

especially important for the emerging storage drive since a single application may not fully saturate the high drive speed. It is crucial to provide support for protection, isolation and bandwidth scheduling. Without data protection, the malicious or erroneous ISC task may operate on unwarranted data; without isolation, the execution of one ISC task may harm the execution of other ISC tasks, or even the firmware execution; without bandwidth scheduling, some ISC tasks may forcibly occupy the drive, hampering fairness and liveness. However, existing work, e.g., [33, 34, 48], does not respond to these issues by assuming a monopolized execution environment.

Finally, another line of research equips the drive with an ASIC, which is the customized hardware chip designed for specific workloads. For instance, YourSQL [40] and Biscuit [38] equip a hardware IP with a key-based pattern matcher; work in [47] adopts a special hardware for database join and scan operations, etc. While ASIC-based solutions can achieve even much better performance compared to the high-end host CPU in their targeting applications, they are too specific to support other tasks. It requires the design of N chips to support N different applications; this introduces N times manufacturing, area size and energy cost. Thus, ASIC solutions are too inflexible to support general ISC.

3 INSIDER System Design

To overcome the problems above, we redesign the storage system across all layers, from the user layer down to the hardware layer. The design of INSIDER is introduced below.

3.1 FPGA-Based ISC Unit

The scenario of ISC puts forth several requirements to the in-drive processing unit.

High reconfigurability. As mentioned earlier, ASIC-based solutions can only target specific workloads. We wish the processing unit to be flexible enough to support general ISC.

Support massive parallelism. We analyze the computation patterns of data analytic workloads (§5.2) that are suitable for ISC. These applications expose abundant pipeline-level and data-level parallelism. The processing unit should have a proper architecture to capture those inherent parallelisms.

High energy efficiency. The storage drive is an energy-efficient device whose power consumption is just about 10 W [14]. The processing unit should not significantly compromise the energy efficiency of the drive.

	GPU	ARM	X86	ASIC	FPGA
Programmability	Good	Good	Good	No	Good
Data-level parallelism	Good	Poor	Fair	Best	Good
Pipeline-level parallelism	No	No	No	Best	Good
Energy efficiency	Fair	Fair	Poor	Best	Good

Table 1: Evaluating five candidates of ISC unit.

Given those requirements, we evaluate several candidates of ISC unit (see Table 1). FPGA comes out to be the best fit in our scenario. First, FPGA is generally reconfigurable and can form customized architectures for the targeted workloads. Second, through customization, FPGA can efficiently capture the inherent parallelism of applications. The data-level parallelism can be seized by replicating the processing elements to construct SIMD units [69]; the pipeline-level parallelism can be leveraged by constructing a deep hardware pipeline [60]. Finally, FPGA could achieve high energy efficiency between microprocessors and ASICs [58].

3.2 Drive Architecture

Fig. 4 presents the system architecture of INSIDER. We focus on introducing the drive-side design in this subsection.

3.2.1 Separating Control and Data Planes

The INSIDER drive controller consists of two decoupled components: the firmware logic and the accelerator cluster (i.e., the FPGA-based ISC unit). The firmware cooperates with the host-side ISC runtime and the ISC driver to enforce the *control plane* execution (marked in Fig. 4). It receives the incoming drive access requests from host, converts their logical block addresses into physical block addresses, and finally issues the requests to the storage unit. The accelerator cluster is separated out into the *data plane*. It does not worry about where to read (write) data from (to) the storage chip. Instead, it **intercepts and processes** the data between the DMA controller and the storage chip.

By separating control and data plane, we expose a *compute-only* abstraction for the in-drive accelerator. It does not proactively initiate the drive accessing request. Instead, it only passively processes the intercepted data from other components. The control plane takes the responsibilities of conducting file permission check at host and issuing drive accessing requests;

it prevents the drive data from being accessed by unwarranted drive programs. In addition, the compute-only abstraction brings an *isolated* environment for the accelerator cluster; its execution will not harm the execution of other system components in the control plane. The execution of different offloaded tasks in the accelerator cluster is further *hardware-isolated* into different portions of FPGA resources.

3.2.2 Accelerator Cluster

As shown in the rightmost portion of Fig. 4, the accelerator cluster is divided into two layers. The inner layer is a programmable region which consists of multiple application slots. Each slot can accommodate a user-defined application accelerator. Different than the multi-threading in CPU, which is time multiplexing, different slots occupy different portions of hardware resources simultaneously, thus sharing FPGA in spatial multiplexing. By leveraging partial reconfiguration [44], host users can dynamically load a new accelerator to the specified slot. The number of slots and slot sizes are chosen by the administrator to meet the application requirements, i.e., number of applications executing simultaneously and the resource consumption of applications. The outer layer is the hardware runtime which is responsible for performing flow control (§3.5) and dispatching data to the corresponding slots (§3.6). The outer layer is set to be user-unprogrammable to avoid safety issues.

3.3 The Host-Side Programming Model

In this section we introduce *virtual file abstraction* which is the host-side programming model of INSIDER. A virtual file is fictitious, but pretends to be a real file from the perspective of the host programmer—it can be accessed via a subset of the POSIX-like file I/O APIs shown in Table 2. The access to virtual file will transparently trigger the underlying system data movement and the corresponding ISC, creating an illusion that this file does really exist. By exposing the familiar file interface, the effort of rewriting the traditional code into the INSIDER host code is negligible.

We would like to point out that INSIDER neither implements the full set of POSIX IO operations nor provides the full POSIX semantics, e.g., crash consistency. The argument here is similar to the GFS [37] and Chubby [29] papers: files provide a familiar interface for host programmers, and exposing a file-based interface for ISC can greatly alleviate the programming overheads. Being fully POSIX-compliant is not only expensive but also unnecessary in most use cases.

3.3.1 Virtual File Read

Listing 1 shows a snippet of the host code that performs virtual file read. We will introduce the design of virtual file read based on the code order. Fig. 5 shows the corresponding diagram.

System startup During the system startup stage, INSIDER creates a hidden mapping file *.USERNAME.insider* in the host file system for every user. The file is used to store the virtual file mappings (which will be discussed soon). For security concerns, INSIDER sets the owner of the mapping file to the corresponding user and sets the file permission to 0640.

```
// register a virtual file
string virt = reg_virt_file(real_path,acc_id);
// open the virtual file
int fd = vopen(virt.c_str(),O_RDONLY);
if (fd != -1) {
  // send drive program parameters (if there are any)
  send_params(fd, param_buf, param_buf_len);
  int rd_bytes = 0;
  // read virtual file
  while (rd_bytes = vread(fd, buf, buf_size)) {
    // user processes the read data
    process(buf, rd_bytes);
  }
  // close virtual file, release resources
  vclose(fd);
}
```

Listing 1: Host-side code of performing virtual file read.

1). int vopen(const char *path, int flags)
2). ssize_t vread(int fd, void *buf, size_t count)
3). ssize_t vwrite(int fd, void *buf, size_t count)
4). int vsync(int fd)
5). int vclose(int fd)
6). int vclose(int fd, size_t *rfile_written_bytes)
7). string reg_virt_file(string file_path, string acc_id)
8). string reg_virt_file(tuple<string, uint, uint> file_sg_list, string acc_id)
9). bool send_params(int fd, void *buf, size_t count)

Table 2: INSIDER host-side APIs. *vwrite, vsync* will be discussed in §3.3.2 while others will be discussed in §3.3.1.

Registration. The host program determines the file data to be read by the in-drive accelerator by invoking *reg_virt_file* (method 7 in Table 2); it takes the path of a real file plus an application accelerator ID, and then maps them into a virtual file. Alternatively, *reg_virt_file* (method 8) accepts a vector of <file name, offset, length> tuples to support the gather-read pattern.[3] This allows us to create the virtual file based on discrete data from multiple real files. During the registration phase, the mapping information will be recorded into the corresponding mapping file, and the specified accelerator will be programmed into an available slot of the in-drive reconfigurable controller. INSIDER currently enforces a simple scheduling policy: it blocks when all current slots are busy.

File open. After registration, the virtual file can be opened via *vopen*. The INSIDER runtime will first read the mapping file to know the positions of the mapped real file(s). Next, the runtime issues the query to the host file system to retrieve the accessing permission(s) and the ownership(s) of the real file(s). Then, the runtime performs the file-level permission check to find out whether the *vopen* caller has the correct access permission(s); in INSIDER, we regard the host file system and INSIDER runtime as trusted components, while the user programs are treated as non-trusted components. If it is an unauthorized access, *vopen* will return an invalid file descriptor. Otherwise, the correct descriptor will be returned, and the corresponding accelerator slot index (used in §3.6) will be

[3] Currently INSIDER operates drive data at the granularity of 64 B, therefore the offset and length fields have to be multiples of 64 B. It is a limitation of our current implementation rather than the design.

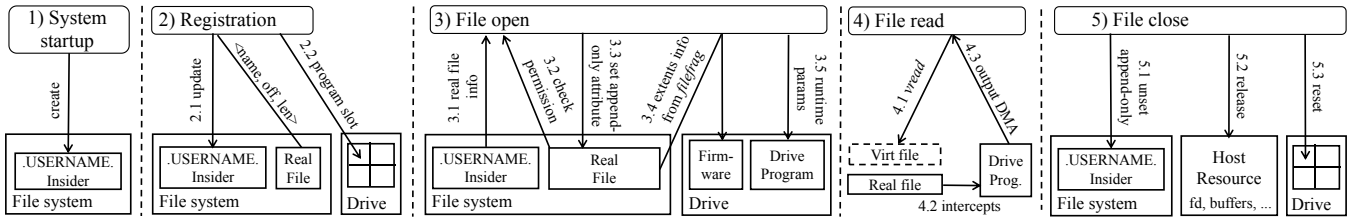


Figure 5: The system diagram of performing virtual file read. Only major steps are shown in this figure, see the text description in §3.3.1 for details.

sent to the INSIDER drive. After that, the INSIDER runtime asks the INSIDER kernel module to set the append-only attribute (if it is not already set by users before) on the mapped real file(s); this is used to guarantee that the current blocks of the real file(s) will not be released or replaced during the virtual file read.[4] Later on, INSIDER retrieves the locations of real file extents via the *filefrag* tool and transfers them to the drive. Finally, the host program sends runtime parameters of the accelerator program (discussed in §3.4), if there are any, via *send_params* to the drive.

File read. Now the host program can sequentially read the virtual file via *vread*. It first triggers the INSIDER drive to read the corresponding real file extents. The accelerator *intercepts* the results from the storage chips and invokes the corresponding data processing. Its output will be transferred back to the host via DMA, creating an illusion that the host is reading a normal file (which actually turns out to be a virtual file). The whole process is deeply pipelined without stalling. The detailed design of pipelining is discussed in §3.5. It seems to be meaningless to read the ISC results randomly, thus we do not implement a *vseek* interface.

File close. Finally, the virtual file is closed via *vclose*. In this step, the INSIDER runtime will contact the INSIDER kernel module to clear the append-only attribute if it was previously set in *vopen*. The host-side resource (e.g., file descriptor, the host-side buffer for DMA, etc.) will be released. Finally, the runtime sends the command to the INSIDER drive to reset the application accelerator to its initial state.

Virtual file read helps us to alleviate the bandwidth bottleneck in the drive → host direction. For example, for the feature selection application [64], the user registers a virtual file based on the preselected training data and the corresponding accelerator. The postselected result could be automatically read via *vread* without transferring the large preselected file from drive to host. Thus, the host program can simply use the virtual file as the input file to run the ML training algorithm.

3.3.2 Virtual File Write

Virtual file write works mostly in the same way but reverses the data path direction. We focus on describing the difference.

Registration. Virtual write requires users to preallocate enough space for the real file(s) to store the write output. If users leverage the *fallocate* system call to preallocate the file, they have to make sure to clear the *unwritten* flag on the file extents.[5] Otherwise, later updates on the real file may only be perceived via the INSIDER interface but not the OS interface.

File open. Besides the steps in §3.3.1, INSIDER runtime invokes *fsync* to flush dirty pages of the real file(s) to drive if there are any. This guarantees the correct order between previous host-initiated write requests and the upcoming INSIDER drive-initiated write requests.

File write. In the file write stage, users invoke *vwrite* to write data to the virtual file. The written data is transferred to INSIDER drive through DMA, and then will be *intercepted and processed* by the accelerator. The output data will be written into the corresponding real file blocks. INSIDER also provides *vsync* (method 4 in Table 2), which can be used by users to flush in-core *vwrite* data to the INSIDER drive.

File close. Besides the steps in §3.3.1, INSIDER runtime will drop the read cache of the real file(s), if there are any, to guarantee that the newly drive-written data can be perceived by the host. This is conducted via calling *posix_fadvise* with *POSIX_FADV_DONTNEED*. Via invoking a variant of *vclose* (method 6 in Table 2), users can know the number of bytes written to the real file(s) by the underlying INSIDER drive. Based on the returned value, users may further invoke *ftruncate* to truncate the real file(s).

Virtual file write helps us alleviate the bandwidth bottleneck in the host → drive direction, since less data needs to be transferred through the bus (they then gets amplified in drive). For example, the user can register a virtual file based on a compressed real file and a decompression drive program. In this scenario, only compressed data needs to be transferred through the bus, and the drive performs in-storage decompression to materialize the decompressed file.

Since the virtual file write is mostly symmetric to the virtual file read, in the following we will introduce other system designs based on **the direction of read** to save space.

3.3.3 Concurrency Control

In INSIDER, a race condition might happen in the following cases: 1) Simultaneously a single real file is being *vwrite* and *vread*; 2) Simultaneously a real file is being *vwrite* by different processes; 3) A single real file is being *vread*, and meanwhile it is being written by a host program. In these cases, the users may encounter non-determinate results.

[4] With the append-only attribute, *ftruncate* will fail to release blocks, and the file defragmentation tool, e.g., *xfs_fsr* will ignore these blocks [21].

[5] In Linux, some file systems, e.g., ext4, will put the *unwritten* flag over the file extents preallocated by *fallocate*. Any following read over the extents will simply return zero(s) without actually querying the underlying drive; this is designed for security considerations since the preallocated blocks may contain the data from other users.

```
1  struct APP_Data {
2   char bytes[64]; // 64-byte width data interface.
3   unsigned short len; // Number of valid bytes.
4   bool eop; // Mark the end of the file.
5  };
6
7  // The ISC filter kernel.
8  void filter(Queue<APP_Data> &app_input_data,
9              Queue<APP_Data> &app_output_data,
10             Queue<unsigned int> &app_input_params) {
11  int lower_bound, upper_bound;
12  bool valid_lower_bound = false;
13  bool valid_upper_bound = false;
14  while (1) {
15   if (!valid_lower_bound) { // Read lower bound.
16    app_input_params.read(lower_bound);
17    valid_lower_bound = true;
18   } else if (!valid_upper_bound) { // Read upper bound.
19    app_input_params.read(upper_bound);
20    valid_upper_bound = true;
21   } else { // Do filtering
22    APP_Data record;
23    app_input_data.read(record);
24    // Call a macro (omitted in code) to transform the
25    // first 4B of input bytes into int.
26    int key = EXTRACT_KEY(record.bytes);
27    // Check the filtering condition.
28    if (lower_bound <= key && key <= upper_bound) {
29     // Write the matched record into the output queue.
30     APP_Data filtered_record;
31     // All 64 bytes data are valid.
32     filtered_record.len = 64;
33     // Set the EOP accordingly.
34     filtered_record.eop = record.eop;
35     // Copy the input record data.
36     for (int i = 0; i < 64; i ++) {
37      filtered_record.bytes[i] = record.bytes[i];
38     }
39     app_output_data.write(filtered_record);
40    } else if (record.eop) {
41     // Mark the EOP of the output when hitting input EOP.
42     filtered_record.len = 0;
43     filtered_record.eop = true;
44     app_output_data.write(filtered_record);
45    }
46   }
47  }
48 }
```

Figure 6: A simple example of the INSIDER drive accelerator code.

The problem also applies to Linux file systems: for example, different host processes may write to a same file. Linux file systems do not automatically enforce the user-level file concurrency control and leave the options to users. INSIDER makes the same decision here. When it is necessary, users can reuse the Linux file lock API to enforce the concurrency control by putting the R/W lock to the mapped real file.

3.4 The Drive-Side Programming Model

In this section we introduce the drive-side programming model. INSIDER defines a clear interface to hide all details of data movements between the accelerator program and other system components so that the device programmer *only* needs to focus on describing the computation logic. INSIDER provides a drive-side compiler which allows users to program in-drive accelerators with C++ (see Fig. 6 for a sample program). Additionally, the INSIDER compiler also supports the traditional RTL (e.g., Verilog) for experienced FPGA programmers. As we will see in §5.2, only C++ is used in the evaluation, and it can already achieve near-optimal performance in our scenario (§5.2.2).

Drive program interface consists of three FIFOs—data input FIFO, data output FIFO and parameter FIFO, as shown in the sample code. Input FIFO stores the intercepted data which is used for the accelerator processing. The output data of the accelerator, which will be sent back to host and acquired by *vread*, is stored into output FIFO. The host-sent runtime parameters are stored in parameter FIFO. The input and the output data are wrapped into a sequence of flits, i.e., *struct APP_Data* (see Fig. 6). The concept of flit is similar to the "word size" in host programs. Each flit contains a 64-byte payload, and the *eop* bit is used for marking the end of the input/output data. The length of data may not be multiples of 64 bytes, the *len* field is used to indicate the length of the last flit. For example, 130-byte data is composed by three flits; the last flit has *eop* = true and *len* = 2.

The sample program first reads two parameters, upper bound and lower bound, from the parameter FIFO. After that, in each iteration, the program reads the input record from the input FIFO. Then the program checks the filtering condition and writes the matched record into the output FIFO. Users can define stateful variables which are alive across iterations, e.g., line 11 - line 13 in Fig. 6, and stateless variables as well, e.g., line 22. These variables will be matched into FPGA registers or block RAMs (BRAMs) according to their sizes. The current implementation does not allow placing variables into FPGA DRAM, but it is trivial to extend.

INSIDER supports **modularity**. The user can define multiple sub-programs chained together with FIFOs to form a complete program, as long as it exposes the same drive accelerator interface shown above. Chained sub-programs will be compiled as separate hardware modules by the INSIDER compiler, and they will be executed in parallel. This is very similar to the dataflow architecture in the streaming system, and we can build a map-reduce pipeline in drive with chained sub-programs. In fact, most applications evaluated in §5.2 are implemented in this way. Stateful variables across sub-programs could also be passed through the FIFO interface.

3.5 System-Level Pipelining

Logically, in INSIDER, *vread* triggers the drive controller to fetch storage data, perform data processing, and transfer the output back to host. After that, the host program can finally start the host-side computation to consume the data. A naive design leads to the execution time $t = t_{drive\_read} + t_{drive\_comp.} + t_{output\_trans.} + t_{host\_comp}$. As we will see in §5.2, this leads to a limited performance.

INSIDER constructs a deep system-level pipeline which includes all system components involved in the end-to-end processing. It happens **transparently** for users; they simply use the programming interface introduced in §3.3 and §3.4. With pipelining, the execution time is decreased to $max(t_{drive\_read}, t_{drive\_comp.}, t_{output\_trans.}, t_{host\_comp})$.

Overlap t_{drive_read} with t_{drive_comp}. We carefully design the INSIDER hardware logic to ensure that it is fully pipelined, so that the storage read stage, computation stage and output DMA stage overlap one another.

Overlap drive, bus and host time We achieve this by ① Pre-issuing the file access requests during *vopen* which would trigger the drive to perform the precomputation; ② Allocating the host memory in the INSIDER runtime to buffer the drive precomputed results. With ①, the drive has all the position information of the mapped real file, and it can perform computation at its own pace. Thus, the host-side operation is decoupled from the drive-side computation. ② further decouples the bus data transferring from the drive-side computation. Now, each time that the host invokes *vread*, it simply pops the precomputed result from host buffers. To prevent the

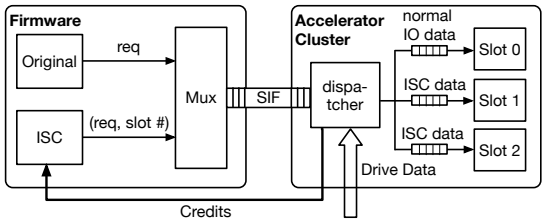


Figure 7: The drive architecture for supporting simultaneous multiple tasks.

drive from overflowing host buffers when host lags behind, INSIDER enforces credit-based flow control for each opened virtual file.

3.6 Adaptive Bandwidth Scheduler

Naturally, the drive is shared among multiple processes, which implies the scenario of parallel execution of multiple applications. For example, a single application may not fully saturate the high internal drive bandwidth so that the remaining bandwidth can be leveraged by others to improve the drive utilization. There are two concerns that should be addressed to support simultaneous multiple applications: 1) Given the fact that the drive is multiplexed among accelerators, we need a mechanism to dispatch drive data to the corresponding accelerator correctly. 2) Different accelerators have different data processing rates which can change dynamically. We need to implement a dynamic weighted fair queueing policy to schedule the drive bandwidth among accelerators adaptively.

Multiplexing. We develop a moderate extension to the original drive firmware (i.e., the one that does not support simultaneous multiple applications) to support multiplexing: we add an ISC unit and a multiplexer, see Fig. 7. The original firmware is used for handling the normal drive I/O requests as usual, while the ISC unit is used for handling the ISC-related drive requests. The ISC unit receives the file logical block addresses and the accelerator slot index from the INSIDER host runtime. The multiplexer will receive the request from both the ISC unit and the original firmware. The received request will be forwarded to the storage unit (not drawn in the figure), and its slot index will be pushed into the *Slots Index FIFO (SIF)*. The slot 0 is always locked for the pass-through logic, which is used for the normal drive read request since it does not need any in-storage processing. Thus, for the request issued by the original firmware, the MUX will push number 0 into SIF. After receiving the drive read data, the dispatcher of the accelerator cluster will pop a slot index from SIF and dispatch the data to the application FIFO connected to the corresponding application slot.

Adaptive Scheduling. The ISC unit maintains a set of *credit registers* (initialized to R) for all offloaded applications. The ISC unit will check registers of applications that have pending drive access requests, in a round-robin fashion. If the register of an application is greater than 0, the ISC unit will issue a drive access request in size C with its slot index, and then decrement its credit register. For the application with a higher data processing rate, its available FIFO size is going to

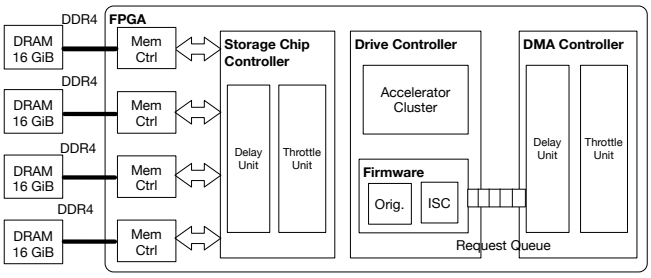


Figure 8: The diagram of the INSIDER drive prototype.

be decreased more quickly, which brings us feedback information for performing the adaptive drive bandwidth scheduling. Each time data is dispatched to the application FIFO, the dispatcher will check the available space of that FIFO. If it is greater than C, the dispatcher will release one credit to the corresponding credit register in the ISC unit.

In practice, we choose the drive access request size C to be the minimal burst size that is able to saturate the drive bandwidth, and choose R to be large enough to hide the drive access latency. Ideally, we could solve an optimization problem to minimize the FPGA buffer size which equals to $R \cdot C$. We leave out the details here.

4 Implementation

The implementation of INSIDER contains 10K lines of C/C++ code (including the traditional host code and the FPGA HLS code), 2K lines of Verilog code and 2K lines of script code. The FPGA-side implementation is done based on the ST-Accel framework [60]. We have already adapted both the drive prototype and the software stack to the AWS F1 (FPGA) instance for public access, see [5].

4.1 The INSIDER Drive Prototype

So far there is no such a drive device on the market yet. We build an INSIDER drive prototype ourselves based on an FPGA board, see Fig. 8. The drive is PCIe-based and its implementation contains three parts: storage chip controller, drive controller and DMA controller. We implement a simple firmware logic in the drive controller; it is responsible for handling host-sent drive access commands, and the functionalities of performing wear-leveling and garbage collection have not been implemented yet. The remaining FPGA resource is used to accommodate the logics of drive programs. To emulate the performance of an emerging storage drive, our prototype should connect to multiple flash chips. However, there is no FPGA board in the market that has enough U.2 or M.2 flash connectors to meet our requirements. Therefore, in our prototype, we use four DRAM chips to emulate the storage chips. We add a set of delay and throttle units into the storage chip controller and DMA controller; they are configurable via our host-side API, therefore we could dynamically configure them to change the performance metrics (*i.e.,* bandwidth and latency) of our emulated storage chips and interconnection bus.

Application	Description	Comment	Data Size (GiB)	Parameter	Devel.Time (Person-Day)	LoC[6]	
						Host	Drive
Grep [38]	String matching.	Fully offloaded. Virtual file read.	60	983040 rows, 65536-byte row. 32-byte search string.	3	51	193
KNN [59]	K-nearest neighbors.	Offload the distance calculation into drive, and put K-partial sort in host. Virtual file read.	56.875	K = 32, 14680064 training cases, 4096 dims, 1-byte weight.	2	77	72
Bitmap file decompression	Decompress the bitmap file.	Offload run-length decoding into drive. Other preparation steps like header parsing and format checking are put in host. Virtual file write.	3.3	Compression ratio is about 7, width = 187577, height = 129000, planes = 1, depth = 8.	4	94	145
Statistics [55, 63]	Statistical calculation per input row.	Offload the row-level data reduction operations into drive, and put the computation over the reduced row data in host. Virtual file read.	48	65536 rows, 196608 numbers per row, 4-byte number.	3	65	170
SQL query [40, 65]	A query consists of select, sum, where, group by, and order by.	Offload data filtering into drive, and put sorting and grouping in host. Virtual file read.	60	2013265920 records 32-byte record.	5	97	256
Integration [48]	Combine data from different sources.	Fully offloaded. Virtual file read.	61	1006632960 records, 64-byte record, 32-byte query.	5	41	307
Feature Selection [64]	Relief algorithm to prune features.	Fully offloaded. Virtual file read.	61	15728640 hit records, 15728640 miss records, 256 dims.	9	50	632

Table 3: A descriptions of applications used in the evaluation. We present the experimental data sizes and application parameters. Additionally, we show the developing effort by listing the lines of code and the development time.

4.2 The INSIDER Software Stack

This section briefly introduces the software stack of INSIDER. We have omitted the details due to space constraints.

Compiler. INSIDER provides compilers for both host and drive. The host-side compiler is simply a standard *g++* which, by default, links to the INSIDER runtime. The front-end of the drive-side compiler is implemented on LLVM, while its back-end is implemented on Xilinx Vivado HLS [22] and a self-built RTL code transformer.

Software Simulator. FPGA program compilation takes hours, which greatly limits the iteration speed of the program development. INSIDER provides a system-level simulator which supports both the C++-level and RTL-level simulation. The simulator reduces the iteration time from hours into (tens of) minutes.

Host-side runtime library. The runtime library bypasses the OS storage stack and is at the user space. When necessary, it will use the POSIX I/O interface to interact with the host file system. Its implementation contains the drive parameter configuring API plus all methods in Table 2. Additionally, the runtime library cooperates with the drive hardware to support the system-level pipelining and the credit-based flow control.

Linux kernel drivers. INSIDER implements two kernel drivers. The first driver registers the INSIDER drive as a block device in Linux so that it could be accessed as a normal storage drive. The second driver is ISC related: it manages the DMA buffer for virtual file operations and is responsible for setting/unsetting the append-only attribute to the real file(s) in *vopen*/*vclose*.

5 Evaluation

5.1 Experiment Setup

We refer to the performance metrics of the current high-end SSDs to determine the drive performance used in our evaluation. On the latency side, the current 3D XPoint SSD already achieves latency less than 10 μs [6, 39]. On the throughput side, the high-end SSD announced in 2017 [17] could achieve 13.0 / 9.0 GB/s sequential R/W performance. We project these numbers (according to the trend in Fig. 1, 2) to represent the performance of the next-generation high-performance drive. Table 4 provides details of our experiment setup. We use 32 CPU cores in the evaluation.

Host	Operating System	Linux LTS 4.4.169
	RAM	128 GB
	CPU	2*Intel Xeon E5-2686 v4
	File System	XFS
Drive	FPGA	Xilinx Virtex XCVU9P
	Capacity	64 GB
	Latency	5 μs
	Sequential R/W	16 GB/s (i.e., 14.9 GiB/s)
	Random 4K R/W	1200 KOPS
	Host/Drive Int. Speed	PCIe Gen3 x8 *or* x16
	Number of Slots	3

Table 4: Experiment setup.

5.2 Applications

We choose applications used in the existing work to evaluate the INSIDER system (see Table 3). We implement them by ourselves. All drive programs are implemented in C++.

5.2.1 Speedup and Its Breakdown

See Fig. 9 for the performance comparison of seven applications. We choose the traditional host-only implementation which uses the POSIX interface as the baseline. It uses OpenMP to parallelize the computation to take advantage of 32 CPU cores. The first optimization is to replace the POSIX interface with the ISC interface to bypass overheads of the host I/O stack. This is conducted by registering the virtual file based on the real file and the pass-through (PT) program. The PT program simply returns all the inputs it receives as outputs. Thus, by invoking *vread* over the virtual file, we acquire the data of the real file. In Fig. 9, *Host-bypass* is the abbreviation for this version, while the suffix x8 and x16 stand for using PCIe Gen3 x8 and x16 as the interconnection, respectively. With the host-side code refactoring, we can conduct pipelining to overlap the computation time and the file access time;

[6] It does not include empty lines, comments, logging, timer, etc.

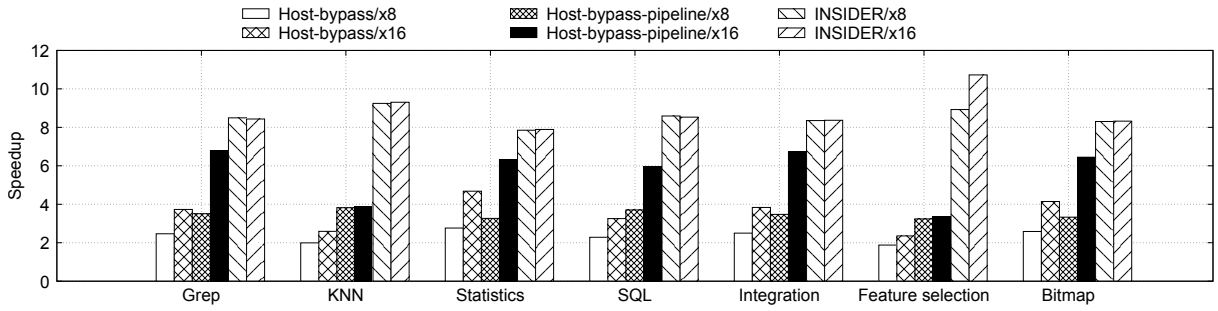
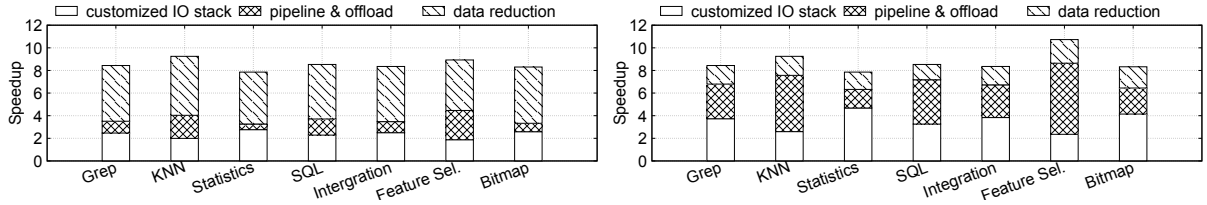


Figure 9: Speedup of optimized host-only versions and INSIDER version compared to the host-only baseline (§5.2.1).



(a) INSIDER*/x8* (*i.e,* the bus-limited case).

(b) INSIDER*/x16* (*i.e,* the bus-ample case).

Figure 10: The breakdown of the speedup achieved by INSIDER compared with the host-only baseline (§5.2.1).

this corresponds to *Host-bypass-pipeline* in Fig. 9. Finally, we leverage the ISC capability to offload computing tasks to the drive. For this version we largely reuse code from the baseline version since the virtual file abstraction allows us to stay at the traditional file accessing interface (§3.3) and INSIDER transparently constructs the system-level pipeline (§3.5). This corresponds to INSIDER in Fig. 9.

Note that the end-to-end execution time here includes the overheads of INSIDER APIs like *vopen, vclose*, but it does not include the overhead of reconfiguring FPGA, which is in the order of hundreds of milliseconds and is proportional to the region size [67]. We envision that in practice the application execution has time locality so that the overheads of reconfiguring will be amortized by multiple following calls.

The speedup of version INSIDER is derived from three aspects: 1) customized I/O stack (§4.2), 2) task offloading (§3.4) and system-level pipelining (§3.5), and 3) reduced data volume (which leads to lower bus time). See Fig. 10 for the speedup breakdown in these three parts. In the x8 setting, which has lower bus bandwidth, data reduction is the major source of the overall speedup. By switching from x8 to x16, the benefit of data reduction decreases, which makes sense since now we use a faster interconnection bus. Nevertheless, it still accounts for a considerable speedup. Meanwhile, pipelining and offloading contribute to a major part of the speedup.

As we discussed in §2.1, four-lane (the most common) and eight-lane links are used in real life because of storage density and cost constraints. INSIDER*/x16* does not represent a practical scenario at this point. The motivation for showing both the results of x8 and x16 is to compare the benefits of data reduction in both bus-limited and bus-ample cases.

5.2.2 Optimality and Bottleneck Analysis

Table 5 shows the performance bottleneck of different execution schemes for seven applications. For *Host-bypass*, limited PCIe bandwidth is the major bottleneck for the overall performance. In contrast, after enabling the in-storage processing, even in the PCIe x8 setting, there is only one case in which PCIe becomes the bottleneck (see INSIDER*/x8*). For most cases in INSIDER, the overall performance is bounded by the internal drive speed, which indicates that the **optimal performance** has been achieved. For some cases, like KNN and feature selection, host-side computation is the performance bottleneck for *Host-bypass*. This is alleviated in INSIDER since FPGA has better computing capabilities for the offloaded tasks. For INSIDER, *SQL query* is still bottlenecked by the host-side computation of the non-offloaded part.

	Host-bypass/x8	*Host-bypass/x16*	INSIDER*/x8*	INSIDER*/x16*
Grep	PCIe	PCIe	Drive	Drive
KNN	PCIe	Comp.	Drive	Drive
Statistics	PCIe	PCIe	Drive	Drive
SQL query	PCIe	Comp.	Comp.	Comp.
Integration	PCIe	PCIe	Drive	Drive
Feature selection	Comp.	Comp.	PCIe	Drive
Bitmap decompression	PCIe	PCIe	Drive	Drive

Table 5: The end-to-end performance bottleneck of different executing schemes over seven different applications. Here *PCIe, Drive and Comp.* indicate that the bottleneck is PCIe performance, drive chip performance and the host-side computation performance, respectively (§5.2.2).

5.2.3 Development Efforts

Table 3 also presents the developing efforts of implementing these applications in terms of lines of code (column *LoC*) and the developing time (column *Devel. Time*). With virtual file abstraction, all host programs here only require less than half an hour to be ported to the INSIDER; The main development time is spent on implementing the drive accelerator which requires drive programmers to tune the performance. This time is expected to be reduced in the future with continuous improvements on the FPGA programming toolchain. Addi-

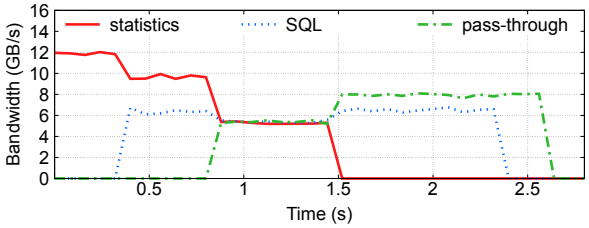


Figure 11: Data rates of accelerators that are executed simultaneously in drive. The drive bandwidth is 16 GB/s, and the bandwidth requested by *statistics*, *SQL* and *pass-through* are 12 GB/s, 6.4 GB/s and 8 GB/s, respectively. *statistics* starts before time 0 s and ends at about time 1.5 s. *SQL* starts at about time 0.4 s and ends at about time 2.4 s. *Pass-through* starts at about time 0.8 s and ends at about time 2.6 s.

	LUT	FF	BRAM	DSP
Grep	34416	24108	1	0
KNN	9534	11975	0.5	0
Statistics	14698	15966	0	0
SQL query	9684	14044	1	0
Integration	40112	6497	14	0
Feature selection	41322	44981	24	48
Bitmap decompression	60837	13676	0	0
INSIDER framework	68981	120451	309	0
DRAM and DMA IP cores	210819	245067	345.5	12
XCVU9P [19]	1181768	2363536	2160	6840
XC7A200T [2]	215360	269200	365	740

Table 6: The top half shows the FPGA resource consumption in our experiments. Generally, an FPGA chip contains four types of resources: look-up tables (LUTs), flip-flops (FFs), block RAMs (BRAMs, which are SRAM-based), digital signal processors (DSPs). The bottom half shows the initial available resource in FPGA XCVU9P and XC7A200T.

tionally, since INSIDER provides a clear interface to separate the responsibilities between host and drive, drive programs could be implemented as a *library* by experienced FPGA developers. This can greatly lower the barrier for host users when it comes to realizing the benefits of the INSIDER drive.

Still, the end-to-end developing time is much less compared to an existing work. Table 1 in work [61] shows that WILLOW requires thousands of LoC and one-month development time to implement some basic drive applications like simple drive I/O (1500 LoC, 1 month) and file appending (1588 LoC, 1 month). WILLOW is definitely an excellent work, and here the main reason is that WILLOW was designed at a lower layer to extend the semantics of the storage drive, while INSIDER focuses on supporting ISC by exposing a compute-only interface at drive and file APIs at host.

5.3 Simultaneous Multiprocessing

In this section we focus on evaluating the effectiveness of the design in §3.6. We choose *statistics, SQL query, and pass-through* as our offloaded applications. On the drive accelerator side, we throttle their computing speeds below the drive internal bandwidth so that each of them cannot fully saturate the high drive rate: $BW_{drive} = 16$ GB/s, $BW_{stat} = 12$ GB/s, $BW_{SQL} = 6.4$ GB/s, $BW_{PT} = 8$ GB/s. The host-side task scheduling has already been enforced by the host OS, and our goal here is to evaluate the effectiveness of the drive-side bandwidth scheduling. Hence, we modify the host programs so that they only invoke INSIDER APIs without doing the host-side computation. In this case, the application execution time is a close approximation of the drive-side accelerator execution time. Therefore, the data processing rate for each accelerator can be calculated as $rate = \Delta size(data)/\Delta time$.

Fig. 11 presents the runtime data rate of three accelerators that execute simultaneously in drive. As we can see, INSIDER will try best to accommodate the bandwidth requests of offloaded applications. When it is not possible to do so, i.e., the sum of total requested bandwidth is higher than the drive bandwidth, INSIDER will schedule bandwidth for applications in a fair fashion.

5.4 Analysis of the Resource Utilization

Table 6 presents the FPGA resource consumption in our experiments. The end-to-end resource usage consists of three parts: ① User application logic. Row *Grep* to row *Bitmap decompression* correspond to this part. ② INSIDER framework. Row INSIDER *framework* corresponds to this part. ③ I/O IP cores. This part mainly comprises the resource for the DRAM controller and the DMA controller. Row *DRAM and DMA IP cores* correspond to this part.

We note that ③ takes the major part of the overall resource consumption. However, these components actually already exist (in the form of ASIC hard IP) in modern storage drives [33], which also have a built-in DRAM controller and need to interact with host via DMA. Thus, ③ only reflects the resource use that would only occur in our prototype due to our limited evaluation environment. The final resource consumption should be measured as ①+②. Row *XCVU9P* [19] and row *XC7A200T* show the available resource of a high-end FPGA (which is used in our evaluation) and a low-end FPGA [7], respectively. We notice that in the best case, the low-end FPGA is able to simultaneously accommodate five resource-light applications (*grep, KNN, statistics, SQL, integration*). The key insight here is that, for the ISC purpose, we *only* need to offload code snippet involving data reduction (related to the virtual file read) or data amplification (related to the virtual file write), therefore the drive programs are frugal in the resource usage.

5.5 Comparing with the ARM-Based System

Methodology. We assume that only the FPGA-based ISC unit is replaced by the ARM CPU, and all other designs remain unchanged. We extract the computing programs from the traditional host-only implementation used in §5.2. Since we assume the system-level pipelining (§3.5) is also deployed here, the final end-to-end time of the ARM-based platform could be calculated as $T_{e2e} = max(T_{host}, T_{trans}, T_{ARM})$, where T_{host} denotes the host-side processing time and T_{trans} denotes the host/drive data transferring time. Here, T_{host} and T_{trans} are taken from the measured data of INSIDER at §5.2. We target Cortex-A72 (using parameters in [12]), which is a high-end quad-core three-way superscalar ARM processor. We conduct runtime profilings over an ARM machine to extract

[7] We do not directly use XC7A200T in the evaluation since we cannot find a low-end FPGA board with large DRAM, which forces us to use XCVU9P.

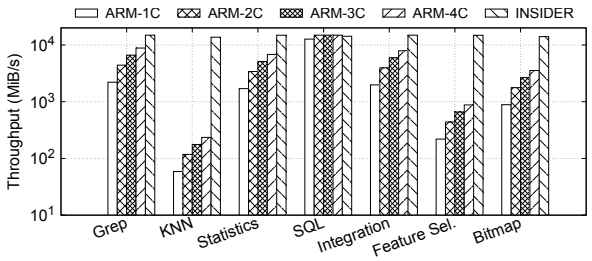


Figure 12: End-to-end data processing rates of INSIDER and the ARM-based platforms. ARM-*N*C means to use *N* core(s).

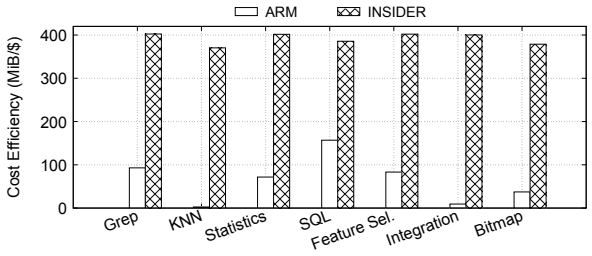


Figure 13: Cost efficiency (defined as data processing rates per dollar) of INSIDER and the ARM-based platforms. We do not include the cost of storage drive, whose price varies significantly across configurations.

the number of program instructions. The program execution time is then calculated *optimistically* by assuming that it has *perfect IPC* and *perfect parallelism* over multiple cores.

Fig. 12 (in log scale) shows the end-to-end data processing rates of INSIDER and the ARM-based platform. The speedup of INSIDER is highly related to the computation intensity of examined applications, but on average, INSIDER could achieve 12X speedup. For *KNN*, which is the most compute-intensive case, INSIDER could achieve 58X speedup; while for *SQL query*, which has the least computation intensity, the ARM-based platform could achieve the same performance.

We further present the cost efficiency of the ARM and INSIDER platforms, which is defined as the data processing rate per dollar. As discussed in §5.4, FPGA XC7A200T is already able to meet our resource demand; thus we use it in this evaluation. The wholesale price of FPGA is much less compared to its retail price according to the experience of Microsoft [36]. For a fair comparison, we use the wholesale prices of FPGA XC7A200T ($37 [20]) and ARM cortex-A72 ($95 [12]). We did not include the cost of storage drive in this comparison. Fig. 13 shows the cost efficiency results. Compared with the ARM-based platform, INSIDER achieves 31X cost efficiency on average. Specifically, it ranges from 2X (in *SQL query*) to 150X (in *KNN*).

6 Future Work

In-storage computing is still in its infancy. INSIDER is our initial effort to marry this architectural concept with a practical system design. There is a rich set of interesting future work, as we summarize in the following.

Extending INSIDER for a broader scenario. First, from the workload perspective, an extended programming model is desired to better support the data-dependent applications like key-value store. The current programming model forces host to initiate the drive access request, thus it cannot bypass the interconnection latency.

Second, from the system perspective, it would be useful to integrate INSIDER with other networked systems to reduce the data movement overheads. Compared to PCIe, performance of the network is further constrained, which creates yet another scenario for INSIDER [45]. The design of INSIDER is mostly agnostic to the underlying interconnection. By changing the DMA part into RDMA (or Ethernet), INSIDER can support the storage disaggregation case, helping cloud users to cross the "network wall" and take advantage of the fast remote drive. Other interesting use cases include offloading computation to HDFS servers and NFS servers.

Data-centric system architecture. Traditionally, the computer system is designed to be computing-centric, in which the data from IO devices are transferred and then processed by CPU. However, the traditional system is facing two main challenges. First, the data movement between IO devices and CPU has proved to be very expensive [53], which can no longer be ignored in the big data era. Second, due to the end of Dennard Scaling, general CPUs can no longer catch up with the ever-increasing speed of IO devices. Our long-term vision is to refactor the computer system into being data-centric. In the new architecture, CPU is only responsible for control plane processing, and it offloads data plane processing directly into the customized accelerator inside of IO devices, including storage drives, NICs [50, 52], memory [51], etc.

7 Conclusion

To unleash the performance of emerging storage drives, we present INSIDER, a full-stack redesigned storage system. On the performance side, INSIDER successfully crosses the "data movement wall" and fully utilizes the high drive performance. On the programming side, INSIDER provides simple but effective abstractions for programmers and offers necessary system support which enables a shared executing environment.

Acknowledgements

We would like to thank our shepherd, Keith Smith, and other anonymous reviewers for their insightful feedback and comments. We thank Wencong Xiao and Bojie Li for all technical discussions and valuable comments. We thank the Amazon F1 team for AWS credits donation. We thank Janice Wheeler for helping us edit the paper draft. This work was supported in part by CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, the NSF NeuroNex award #DBI-1707408, and the funding from Huawei, Mentor Graphics, NEC and Samsung under the Center for Domain-Specific Computing (CDSC) Industrial Partnership Program. Zhenyuan Ruan is also supported by a UCLA Computer Science Departmental Fellowship.

References

[1] Anobit Announces Best-in-Class Flash Drives for Enterprise and Cloud Applications. https://www.businesswire.com/news/home/20110914005522/en/Anobit-Announces-Best-in-Class-Flash-Drives-Enterprise-Cloud.

[2] Artix-7 FPGA Product Table. https://www.xilinx.com/products/silicon-devices/fpga/artix-7.html#productTable.

[3] CES 2009: pureSilicon 1TB Nitro SSD. https://www.slashgear.com/ces-2009-puresilicon-1tb-nitro-ssd-1230084/.

[4] DTS. http://www.storagesearch.com/dts.html.

[5] INSIDER Github Repository. https://github.com/zainryan/INSIDER-System.

[6] Intel Optane SDD 900P Series. https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/gaming-enthusiast-ssds/optane-900p-series/900p-280gb-aic-20nm.html.

[7] Intel Optane SSD DC P4800X. https://www.intel.com/content/www/us/en/solid-state-drives/optane-ssd-dc-p4800x-brief.html.

[8] Intel Solid-State Drive 910 Series Product Specification. https://ark.intel.com/products/67009/Intel-SSD-910-Series-800GB-12-Height-PCIe-2_0-25nm-MLC.

[9] Intel X25-M 80GB SSD Drive Review. http://www.the-other-view.com/intel-x25.html.

[10] Lite-On SSD News. http://www.liteonssd.com/m/Company/news_content.php?id=LITE-ON-INTRODUCES-THE-NEXT-GENERATION-EP2-WITH-NVME-PROTOCOL-AT-DELL-WORLD-2015.html.

[11] Memory and Storage / Solid State Drives / Intel Enthusiast SSDs / Intel Optane SSD 900P Series. https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/gaming-enthusiast-ssds/optane-900p-series/900p-280gb-2-5-inch-20nm.html.

[12] Microprocessors - MPU QorIQ Layerscape. https://www.mouser.com/ProductDetail/NXP-Freescale/LS1046ASN8T1A?qs=sGAEpiMZZMup8ZLti7BNCxtNz7%252BF43hzZlkvLaqOJ8c%3D.

[13] Samsung Demos Crazy-Fast PCIe NVMe SSD At 5.6 GB Per Second At Dell World. https://hothardware.com/news/samsung-demos-crazy-fast-pcie-nvme-ssd-at-56-gb-per-second-showcases-16tb-ssd-at-dell-world.

[14] Samsung NVMe SSD 960 Pro. https://www.samsung.com/us/computing/memory-storage/solid-state-drives/ssd-960-pro-m-2-512gb-mz-v6p512bw/.

[15] SanDisk:Solid State Disk Drive. https://www.anandtech.com/show/2151/4.

[16] Seagate announces 64TB NVMe SSD, Updated Nytro NVMe and SAS Lineup at FMS 2017. https://www.custompcreview.com/news/seagate-announces-64tb-nvme-ssd-updated-nytro-nvme-sas-lineup-fms-2017/.

[17] Seagate Nytro 5910 NVMe SSD. https://www.seagate.com/files/www-content/datasheets/pdfs/nytro-5910-nvme-ssdDS1953-4-1804US-en_US.pdf.

[18] Storage news - 2007, October week 3. http://www.storagesearch.com/news2007-oct3.html.

[19] UltraScale+ FPGAs Product Tables and Product Selection Guide. https://www.xilinx.com/support/documentation/selection-guides/ultrascale-plus-fpga-product-selection-guide.pdf.

[20] XC7A200T-1FFG1156C(IC Embedded FPGA Field Programmable Gate Array 500 I/O 1156FCBGA). https://www.alibaba.com/product-detail/XC7A200T-1FFG1156C-IC-Embedded-FPGA-Field_60730073325.html.

[21] XFS defragmentation tool will ignore the file which has append-only or immutable attribute set. https://kernel.googlesource.com/pub/scm/fs/xfs/xfsprogs-dev/+/v4.3.0/fsr/xfs_fsr.c#968.

[22] Xilinx Vivado HLS. https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

[23] Nitin Agrawal and Ashish Vulimiri. Low-Latency Analytics on Colossal Data Streams with SummaryStore. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 647–664, New York, NY, USA, 2017. ACM.

[24] Duck-Ho Bae, Jin-Hyung Kim, Sang-Wook Kim, Hyunok Oh, and Chanik Park. Intelligent SSD: a turbo for big data mining. In *Proceedings of the 22nd ACM international conference on Conference on information and knowledge management*, CIKM '13, pages 1573–1576, New York, NY, USA, 2013. ACM.

[25] R. Balasubramonian and B. Grot. Near-Data Processing [Guest editors' introduction]. *IEEE Micro*, 36(1):4–5, Jan 2016.

[26] Antonio Barbalace, Anthony Iliopoulos, Holm Rauchfuss, and Goetz Brasche. It's Time to Think About an Operating System for Near Data Processing Architectures. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, pages 56–61, New York, NY, USA, 2017. ACM.

[27] Laurent Bindschaedler, Jasmina Malicevic, Nicolas Schiper, Ashvin Goel, and Willy Zwaenepoel. Rock You like a Hurricane: Taming Skew in Large Scale Analytics. In *Proceedings of the Thirteenth European Conference on Computer Systems*, EuroSys '18, 2018.

[28] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems. In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, pages 22:1–22:10, New York, NY, USA, 2013. ACM.

[29] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[30] Adrian M. Caulfield, Joel Coburn, Todor Mollov, Arup De, Ameen Akel, Jiahua He, Arun Jagatheesan, Rajesh K. Gupta, Allan Snavely, and Steven Swanson. Understanding the Impact of Emerging Non-Volatile Memories on High-Performance, IO-Intensive Computing. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[31] Adrian M. Caulfield, Arup De, Joel Coburn, Todor I. Mollow, Rajesh K. Gupta, and Steven Swanson. Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-volatile Memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 385–395, Washington, DC, USA, 2010. IEEE Computer Society.

[32] Adrian M. Caulfield, Todor I. Mollov, Louis Alex Eisner, Arup De, Joel Coburn, and Steven Swanson. Providing Safe, User Space Access to Fast, Solid State Disks. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 387–400, New York, NY, USA, 2012. ACM.

[33] Sangyeun Cho, Chanik Park, Hyunok Oh, Sungchan Kim, Youngmin Yi, and Gregory R. Ganger. Active Disk Meets Flash: A Case for Intelligent SSDs. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 91–102, New York, NY, USA, 2013. ACM.

[34] 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. ACM.

[35] P. Fernando, S. Kannan, A. Gavrilovska, and K. Schwan. Phoenix: Memory speed hpc i/o with nvm. In *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*, pages 121–131, Dec 2016.

[36] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure accelerated networking: Smartnics in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, Renton, WA, 2018. USENIX Association.

[37] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 20–43, Bolton Landing, NY, 2003.

[38] 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. Biscuit: A Framework for Near-data Processing of Big Data Workloads. In

Proceedings of the 43rd International Symposium on Computer Architecture, ISCA '16, pages 153–165, Piscataway, NJ, USA, 2016. IEEE Press.

[39] F. T. Hady, A. Foong, B. Veal, and D. Williams. Platform Storage Performance With 3D XPoint Technology. *Proceedings of the IEEE*, 105(9):1822–1833, Sep. 2017.

[40] Insoon Jo, Duck-Ho Bae, Andre S. Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel D. G. Lee, and Jaeheon Jeong. YourSQL: A High-performance Database System Leveraging In-storage Computing. *Proc. VLDB Endow.*, 9(12):924–935, August 2016.

[41] Myoungsoo Jung and Mahmut Kandemir. Revisiting Widely Held SSD Expectations and Rethinking System-level Implications. In *Proceedings of the ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '13, pages 203–216, New York, NY, USA, 2013. ACM.

[42] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a Warehouse-scale Computer. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 158–169, New York, NY, USA, 2015. ACM.

[43] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. A Case for Intelligent Disks (IDISKs). *SIGMOD Rec.*, 27(3):42–52, September 1998.

[44] Ahmed Khawaja, Joshua Landgraf, Rohith Prakash, Michael Wei, Eric Schkufza, and Christopher J. Rossbach. Sharing, Protection, and Compatibility for Reconfigurable Fabric with AmorphOS. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 107–127, Carlsbad, CA, 2018. USENIX Association.

[45] Byungseok Kim, Jaeho Kim, and Sam H. Noh. Managing Array of SSDs When the Storage Device Is No Longer the Performance Bottleneck. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, Santa Clara, CA, 2017. USENIX Association.

[46] Sungchan Kim, Hyunok Oh, Chanik Park, Sangyeun Cho, and Sang-Won Lee. Fast, energy efficient scan inside flash memory SSDs. In *Proceeedings of the International Workshop on Accelerating Data Management Systems (ADMS)*, 2011.

[47] Sungchan Kim, Hyunok Oh, Chanik Park, Sangyeun Cho, Sang-Won Lee, and Bongki Moon. In-storage Processing of Database Scans and Joins. *Inf. Sci.*, 327(C):183–200, January 2016.

[48] Gunjae Koo, Kiran Kumar Matam, Te I, H. V. Krishna Giri Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, and Murali Annavaram. Summarizer: Trading Communication with Computing Near Storage. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, pages 219–231, New York, NY, USA, 2017. ACM.

[49] Philip Kufeldt, Carlos Maltzahn, Tim Feldman, Christine Green, Grant Mackey, and Shingo Tanaka. Eusocial Storage Devices: Offloading Data Management to Storage Devices that Can Act Collectively. *;login:*, 43(2), 2018.

[50] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 137–152, New York, NY, USA, 2017. ACM.

[51] Shuangchen Li, Dimin Niu, Krishna T. Malladi, Hongzhong Zheng, Bob Brennan, and Yuan Xie. DRISA: A DRAM-based Reconfigurable In-Situ Accelerator. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, pages 288–301, New York, NY, USA, 2017. ACM.

[52] Yuanwei Lu, Guo Chen, Zhenyuan Ruan, Wencong Xiao, Bojie Li, Jiansong Zhang, Yongqiang Xiong, Peng Cheng, and Enhong Chen. Memory efficient loss recovery for hardware-based transport in datacenter. In *Proceedings of the First Asia-Pacific Workshop on Networking, APNet 2017, Hong Kong, China, August 3-4, 2017*, pages 22–28, 2017.

[53] Onur Mutlu, Saugata Ghose, Juan Gómez-Luna, and Rachata Ausavarungnirun. Processing data where it makes sense: Enabling in-memory computation. *Microprocessors and Microsystems*, 2019.

[54] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. An Analysis of Persistent Memory Use with WHISPER. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 135–148, New York, NY, USA, 2017. ACM.

[55] Jian Ouyang, Shiding Lin, Zhenyu Hou, Peng Wang, Yong Wang, and Guangyu Sun. Active SSD Design for Energy-efficiency Improvement of Web-scale Data Analysis. In *Proceedings of the 2013 International Symposium on Low Power Electronics and Design*, ISLPED '13, pages 286–291, Piscataway, NJ, USA, 2013. IEEE Press.

[56] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. SDF: Software-defined Flash for Web-scale Internet Storage Systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 471–484, New York, NY, USA, 2014. ACM.

[57] D. Park, J. Wang, and Y. S. Kee. In-Storage Computing for Hadoop MapReduce Framework: Challenges and Possibilities. *IEEE Transactions on Computers*, PP(99):1–1, 2016.

[58] A. Putnam. (Keynote) The Configurable Cloud - Accelerating Hyperscale Datacenter Services with FPGA. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 1587–1587, April 2017.

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

[60] Z. Ruan, T. He, B. Li, P. Zhou, and J. Cong. St-accel: A high-level programming platform for streaming applications on fpga. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 9–16, April 2018.

[61] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A User-Programmable SSD. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 67–80, Broomfield, CO, 2014. USENIX Association.

[62] Cassidy R. Sugimoto, Hamid R. Ekbia, and Michael Mattioli. *Big Data Is Not a Monolith*. The MIT Press, 2016.

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

[64] Ryan J Urbanowicz, Melissa Meeker, William LaCava, Randal S Olson, and Jason H Moore. Relief-based feature selection: introduction and review. *arXiv preprint arXiv:1711.08421*, 2017.

[65] Louis Woods, Zsolt István, and Gustavo Alonso. Ibex - An Intelligent Storage Engine with Support for Advanced SQL Off-loading. *PVLDB*, 7(11):963–974, 2014.

[66] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. Performance Analysis of NVMe SSDs and Their Implication on Real World Databases. In *Proceedings of the 8th ACM International Systems and Storage Conference*, SYSTOR '15, pages 6:1–6:11, New York, NY, USA, 2015. ACM.

[67] Jiansong Zhang, Yongqiang Xiong, Ningyi Xu, Ran Shu, Bojie Li, Peng Cheng, Guo Chen, and Thomas Moscibroda. The feniks fpga operating system for cloud computing. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, APSys '17, pages 22:1–22:7, New York, NY, USA, 2017. ACM.

[68] Peipei Zhou, Zhenyuan Ruan, Zhenman Fang, Megan Shand, David Roazen, and Jason Cong. Doppio: I/O-Aware Performance Analysis, Modeling and Optimization for In-Memory Computing Framework. In *IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '18, 2018.

[69] H. R. Zohouri, N. Maruyama, A. Smith, M. Matsuda, and S. Matsuoka. Evaluating and Optimizing OpenCL Kernels for High Performance Computing with FPGAs. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 409–420, Nov 2016.