Analyzing and Modeling In-Storage Computing Workloads On EISC — An FPGA-Based System-Level Emulation Platform

Zhenyuan Ruan* Tong He Jason Cong
University of California, Los Angeles
{zainryan, tonghe, cong}@cs.ucla.edu

***Abstract*—Storage drive technology has made continuous improvements over the last decade, shifting the bottleneck of the data processing system from the storage drive to host/drive interconnection. To overcome this "data movement wall," people have proposed in-storage computing (ISC) architectures which add the computing unit directly into the storage drive. Rather than moving data from drive to host, it offloads computation from host to drive, thereby alleviating the interconnection bottleneck.**

Though existing work shows the effectiveness of ISC under some specific workloads, they have not tackled two critical issues: 1) ISC is still at the early research stage, and there is no available ISC device on the market. Researchers lack an effective way to accurately explore the benefits of ISC under different applications and different system parameters (drive performance and interconnection performance). 2) What kinds of applications can benefit from ISC, and what cannot? It is crucial to have a method to quickly discriminate between the types of applications before spending significant efforts to implement them.

This paper gives a response to the above problems. First, we build a complete FPGA-based ISC emulation system to enable rapid exploration. To the best of our knowledge, it is the first open-source [1], publicly accessible ISC emulation system. Second, we use our system to evaluate 12 common applications. The results give us the basic criteria for choosing ISC-friendly applications. By assuming a general drive program construct, we provide further insights by building an analytical model which enables an accurate quantitative analysis.

I. Introduction

In recent years, *near-data computing* has become a hot topic in the research community, e.g., [20], [18], [14], [21], [23]. Notably, the CRISP (Center for Research in Intelligent Storage and Processing in Memory), a multiyear research center funded by SRC and DARPA, is focusing on re-architecting the computing system by introducing computing units into data devices.

The idea of ISC is motivated by the ever-increasing performance gap between storage drive and host/drive interconnection. A recent paper [25] shows that the storage drive bandwidth has doubled every two years from 2007 to 2017. However, meanwhile the performance of the host/drive interconnection bus, i.e., PCIe, is unable to follow the trend. This performance gap impedes host users trying to leverage advancement in the storage drive technology. The presence of ISC aims to bridge this gap (§II). With the intelligent storage drive, host offloads the first round of computation into the drive (this contains computation patterns like filtering or reduction). By making this computation directly happen inside the drive, we can leverage the high internal drive performance. The volume of the output data is expected to be reduced; therefore, less data will be transferred through the bus, which alleviates the interconnection bottleneck.

After witnessing the effectiveness of ISC in some efforts, researchers wish to further explore its benefits under a broader space, i.e., different applications and system (drive and interconnection) performance parameters. Since ISC is still at its early research stage, there is no available ISC drive on the market. Existing work conducted evaluations using their own private prototypes which have *fixed system parameters* and are *not publicly accessible* for others (§III). To further push forward the concept of ISC in both academia and industry, it is critical to have an open-source system-level emulation platform. First, in order to enable a rich set of research work, researchers need an easy-to-use evaluation system to validate their design and conduct the architectural exploration. Every team that independently builds their own specific platform incurs wasteful duplication of efforts and can greatly slow down the research progress in this area. Second, vendors face a "chicken or the egg" problem of manufacturing ISC drives, and they will only do this after seeing enough benefits of ISC. Having such an open-source evaluation platform can help them better evaluate ISC in a broad space of applications and system performance parameters.

To address the pain point, we propose *EISC (Emulator for In-Storage Computing)*, an open-source FPGA-based emulation platform for ISC. Leveraging our previous work INSIDER [25] on the programming and system support for ISC, in this paper we focus on introducing the design and implementation of the emulation platform for ISC (§IV). Specifically we discuss how to emulate the yet nonexistent ISC drive. At the high level, EISC achieves this goal with a DRAM-equipped FPGA board; it uses the DRAM chips to mimic the storage chips and implements the drive controller logic in FPGA. The remaining FPGA resource is used to accommodate the application logic, enabling the drive ISC capability. One desired property is to have *configurable* system performance parameters. EISC realizes this in hardware by selectively asserting idle signals after the DRAM controller and the PCIe controller.

Along with the research advance in ISC, an unanswered is that *what kinds of application can benefit from ISC, and what cannot?* While existing work aims on demonstrating the power of ISC in some particular workloads (e.g., [20], [14], [28], etc.), they do not answer this general question. However, implementing a new application in the ISC device will definitely requires higher efforts compared with the standard host programming. It is crucial to have an analytical way to quickly tell that whether an application could benefit from ISC in the first place. To answer this question, in this work we leverage EISC to do design space exploration to examine the benefits of ISC (§V). We evaluate 12 common applications under different drive and interconnection performance parameters (§V-B). Guided by the results, we propose a set of guidelines for choosing ISC friendly applications, which allows programmers to qualitatively make quick decisions using two simple application characteristics (§V-C). By assuming a general drive program construct, we build an analytical model to enable an accurate quantitative analysis (§V-D).

II. Background

To combat the issue of limited interconnection performance (discussed in §I), researchers proposed the *in-storage computing (ISC)* architecture (e.g., [26], [20]). In this serction, we give a brief background introduction of ISC.

*The corresponding author.

[1]https://github.com/zainryan/EISC

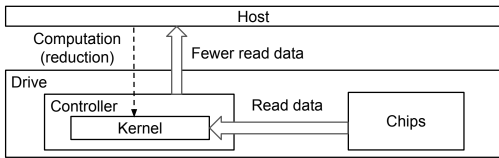


Fig. 1: Saving bus bandwidth from drive to host using ISC.

A. *Saving Interconnection Bandwidth from Drive to Host*

Rather than moving data from storage drive to host, ISC architecture moves computation from host to drive (see Figure 1). The controller equips an embedded computing unit which is able to execute host-offloaded tasks. It performs a *first-round* of computation in drive so that it can leverage the high in-storage read performance. Developers offload computation patterns like data filtering or reduction, thus the output data volume is reduced. In this way, much fewer data need to be transferred back to host. This alleviates the performance bottleneck of the interconnection from drive to host.

B. *Saving Interconnection Bandwidth from Host to Drive*

Similarly, ISC architecture enables us to save interconnection bandwidth from host to drive. Developers first offload tasks that involve data amplification patterns, e.g., file decompression. Then host writes data into the storage drive controller which performs in-storage computing. The output data, whose volume is larger than the original input data, is written to the storage chips; it takes advantage of the high in-storage write performance.

C. *FPGA-Based In-Storage Computing System*

There are multiple candidates for the ISC unit. Existing work, e.g., [31], [12], demonstrates the effectiveness of FPGA for ISC. Compared to ASIC, FPGA could be customized to support a wide range of applications, enabling general ISC. With processing units like ARM or X86, it is difficult to saturate the high-storage bandwidth—which could be as high as tens of GB/s. Compared to them, FPGA can more easily reach the performance goal through hardware customization. Finally, since the storage drive is a low-power device, the ISC unit should not significantly compromise the power efficiency of the drive. Compared to GPU, FPGA could achieve better energy efficiency. In this paper we target the FPGA-based ISC system.

III. MOTIVATION

Though receiving much attention in academia, the ISC architecture is still at a relatively early research stage; there is no general programmable ISC drive on the market yet. Vendors face a "chicken or the egg" problem of manufacturing ISC drives: they will only do this after seeing enough benefits of ISC. Existing work, e.g., [14], [18], [26], [20] , fail to answer this problem due to the following reasons.

a) Target Limited Applications: Existing work targets a few specific applications to demonstrate the advantages of ISC, e.g., [20], [18], [14]. However, they fail to show the cases that are not suitable for ISC; the generality of this architecture still remains unknown. Porting an application to a new architecture requires non-trivial efforts. It would be very helpful to provide insights on how to qualitatively or quantitatively judge whether an application could potentially benefit from ISC before actually implementing it.

b) Fixed System Parameters: Existing work only targets the prototyping platforms, and reports results from them. However, different storage drives have different performance parameters (i.e., bandwidth and latency), and different host systems have different interconnection bus performance; existing work is unable to tell the benefits under changed system parameters. In fact, as we will show later, the speedup brought by ISC is highly related to these parameters. It is important to demonstrate that ISC will be worthwhile under what system settings.

c) Not Publicly Accessible: Existing work are evaluated on their private testbeds. There is no publicly accessible ISC platform for the community; it prevents system designers from exploring the benefits of ISC under their own scenarios. Building a system prototype is a time-consuming process which is not affordable for every team. The lack of publicly accessible emulation platforms not only obstructs industry from evaluating the benefits of ISC, but also impedes the research progress in this area. It is crucial to have an open-source system-level emulation platform to enable potential users to engage in rapid exploration.

IV. DESIGN AND IMPLEMENTATION

A. *Design Goals and System Overview*

In order to overcome the above problems, we build a new emulation platform EISC. It achieves the following design goals.

a) End-to-End System Emulation: Rather than the functional simulation software like GEM5 [5] or PARADE [10], we decide to build an end-to-end system emulation platform. The goal is to provide a system-level emulation environment *as real as possible*. This is important for two reasons: 1) It can provide much more accurate results compared to the functional simulation; 2) It can create a realistic system prototype enabling users to integrate it with the system software stack and experiment with various applications. We achieve this goal by building an FPGA-emulated ISC drive prototype (§IV-B) and its corresponding software stack (§IV-C).

b) Configurable System Parameters: To enable system-level design exploration, EISC should allow users to configure important system parameters, i.e. the bandwidth and latency of the interconnection bus and the storage drive. EISC achieves this goal with two design choices. First, EISC adopts PCIe Gen3 X16 as the interconnection and uses DRAM chips to mimic the storage chips. Their performance is high enough to serve as the performance upper bound of the interconnection bus and the emerging storage chips. Second, we introduce the delay units and the throttle units into both the DMA controller (§IV-B2) and storage controller (§IV-B3). Those units are configurable via the host API which allows users to dynamically configure the latency and bandwidth lower than the aforementioned upper bounds.

c) Easy to Port Applications: As an emulation platform, one important aspect is to ensure the simplicity to port applications to our platform. EISC is an FPGA-based ISC system; thus we have to carefully design the accelerator kernel interface so that existing FPGA kernels can be easily ported (IV-C1). Also, the host API has to be straightforward so that the host program can easily interact with the EISC drive (IV-C2).

d) Publicly Accessible: We make EISC an open-source, publicly accessible emulation platform to benefit the community. To achieve this goal, we could either host a large-scale shared platform accessing service like Emulab [3] or implement our system upon a public cloud service. Here we choose the latter; we adapt EISC to the AWS F1 instance [1] which provides the necessary hardware/software environment (IV-D). Everyone can deploy the EISC system on the F1 instance with our code.

In the following sections we are going to introduce the design of EISC. First, we will introduce the design of EISC drive which is an FPGA-based ISC device (§IV-B). After that, we will introduce the corresponding host APIs to interact with the EISC drive (§IV-C).

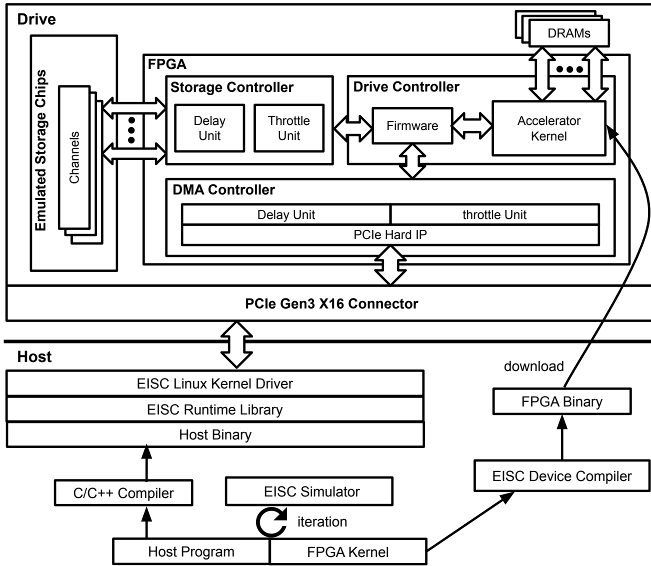


Fig. 2: The EISC architecture.

Finally, we will introduce how we adapt and implement EISC on the AWS F1 instance (§IV-D).

B. EISC Drive

The top half of Figure 2 presents the EISC drive architecture. In the high level, the EISC drive consists of emulated storage chips and an FPGA chip. We implement three major modules, i.e., the DMA controller, storage controller and drive controller, over the FPGA chip.

1) PCIe Interface: The EISC drive is inserted at the PCIe Gen3 X16 slot of the host motherboard. In the realistic settings, the storage drive uses far fewer PCIe lanes—mainly for cost and storage density considerations. Here, we choose PCIe Gen3 X16 to create an upper bound of the interconnection performance. With the delay and throttle units in the DMA controller (§IV-B2), we are able to emulate any performance number below this upper bound.

2) DMA Controller: The EISC DMA controller is an interface between the drive controller and the PCIe bus. Through the DMA controller, the host-sent data and command can be passed to the drive firmware. The PCIe hard IP in the bottom (of the DMA controller) is responsible for converting the raw PCIe signals into the AXI [2] interface. One design goal of EISC is to make system parameters configurable. To achieve configurable interconnection bus performance, we introduce a set of delay and throttle units to the DMA controller. The delay unit is responsible for instructing the delay cycles into the signals from/to the PCIe interface. Via invoking the host method $set_dma_delay_unit(x)$, the host user can increase the latency of the interconnection bus by x cycles to $lat_{original} + x \cdot freq_{FPGA}^{-1}$. The throttle unit is used for changing the duty ratio of the data signals from/to the PCIe interface. Host users could configure the throttle unit via the host method $set_dma_throttle_unit(x, y)$; it asks the throttle unit to insert y empty cycles after receiving x cycles of effective signals. After applying this command, the host user can decrease the bandwidth of the interconnection bus to $bw_{original} \cdot x/(x + y)$.

3) Storage Controller: In EISC we use DRAM chips to emulate the storage chips. We make this decision since the performance of DRAM chips could serve as an upper bound in the performance of high-performance storage chips. We implement a storage controller which assists the drive controller with accessing the emulated storage chips. To mimic the multi-channel property of storage chips in commodity drive, we attach multiple DRAM chips to the storage controller with separate connectors.

The first job of storage controller is to make the performance of emulated storage chips configurable. We achieve this by adding a set of delay units and throttle units, which are similar to the ones in §IV-B2. We also provide corresponding host APIs for users to dynamically configure them.

The second job of storage controller is to expose a unified storage chip interface for the drive controller. Similar to the commodity drive, we implement the block size (i.e., the finest accessing granularity) as 4 KB, which is a standard setting in most commodity storage drives. Thus, the addressing space of each emulated storage chip is partitioned into chunks of 4 KB. We adopt the low-order interleaving to merge the addressing space of multiple storage chips into a single unified space.

4) Drive Controller: The EISC drive controller consists of two modules: firmware and accelerator kernel. Similar to a commodity drive, we implement a simple flash transaction layer (FTL) in our firmware module; it decodes the host-sent drive commands, performs drive logical block address (LBA) to physical block address (PBA) translation, and issues the corresponding accessing requests to the storage controller.

The accelerator kernel is a unique component of EISC in which users could put their ISC logic. Commodity drive equips embedded DRAMs [13]. To emulate this, in our EISC drive, we attach DRAM chips to the accelerator kernel, which could serve as a buffer to store the intermediate data. Via the firmware module, the accelerator kernel is able to talk with the storage controller and the DMA controller. Therefore, the accelerator can read/write data from/to the storage chips and send/receive data from/to the PCIe interface.

C. EISC Host Stack

The bottom half of Figure 2 presents the host-side architecture which will be introduced in the followings.

1) Development Toolchains: Users could easily port an FPGA kernel into EISC. We provide an EISC compiler which accepts either the Xilinx high-level synthesis (HLS) kernel or the RTL kernel. The top-level definition of user-provided kernels should observe the following interface (shown in the HLS style).

```
void kernel(
    /* ISC related interface */
    hls::stream<ap_uint<512>> &input_channel,
    hls::stream<ap_uint<512>> &output_channel,
    hls::stream<unsigned int> &args_channel,
    /* DRAM accessing interface */
    hls::stream<Dram_Read_Req> &dram_read_req_channel,
    hls::stream<Dram_Read_Resp> &dram_read_resp_channel,
    hls::stream<Dram_Write_Req> &dram_write_req_channel,
    /* Debugging/monitoring interface */
    hls::stream<Peek_Req> peek_req_channel,
    hls::stream<Peek_Resp> peek_resp_channel
);
```

Listing 1: The interface of EISC drive kernel.

The user-provided HLS kernel acquires input from *input_channel*, performs computation, and writes output into *output_channel*. It can accept host-sent runtime arguments from *args_channel*.

It can also use drive-embedded DRAMs to store the intermediate results through the DRAM accessing interface (a common DRAM stream interface which is also used in ST-Accel [24]). Notably, this is a general interface and many existing kernels are written in this form (e.g., the ones in [24], [15]).

Once running the application on EISC, it is common that the performance does not match the expectation, or the output results

Parameter Configuration	1) void set_dma_delay_unit(uint32 x)
	2) void set_dma_throttle_unit(uint32 x, uint32 y)
	3) void set_storage_delay_unit(uint32 x)
	4) void set_storage_throttle_unit(uint32 x, uint32 y)
Kernel Runtime	5) void send_runtime_args(uint32 *args, int num)
ISC Read Kernel (§IV-C1)	6) void set_input_file(string input_file_path)
	7) void set_input_lba_vec(vector<uint64>input_lba_vec)
	8) unsigned long long recv_isc_unit(char *buf, uint64 len)
ISC Write Kernel (§IV-C1)	9) void set_output_file(string output_file_path)
	10) void set_output_lba_vec(vector<uint64>output_lba_vec)
	11) unsigned long long send_isc_unit(char *buf, uint64 len)
Debugging	12) unsigned int peek_reg(uint32 id);

TABLE I: EISC host APIs (§IV-C2).

are not correct. EISC provides a handy interface that enables host users to peek the states in the drive FPGA kernel for debugging. Host users issue $Peek_Req$ which will be stored in $peek_req_channel$. $Peek_Req$ is essentially a 32-bit identifier. After reading the identifier, the FPGA kernel can write the response data, which essentially contains the debug information, to $peek_resp_channel$. EISC runtime guarantees to deliver the response data back to host users.

We classify the EISC kernels into two categories according to their data flow directions: ISC read kernel (§II-A) and ISC write kernel (§II-B). For the ISC read kernel, its *input_channel* stores the data read from emulated storage chips, while its *output_channel* stores the data that will be sent back to host via the interconnection bus. For the ISC write kernel, the above bindings are reversed.

After accepting the user-provided kernel, the compiler front end performs the source-to-source code transformation to make it compatible with our system framework. Then, if it is an HLS kernel, Xilinx Vivado HLS will be invoked to generate the RTL kernel code. Next, the compiler backend will connect the generated RTL code with the RTL implementation of EISC drive (§IV-B), and generate a complete FPGA source project. Finally, the Xilinx Vivado tool suite is invoked to synthesize the project and generate the FPGA binary.

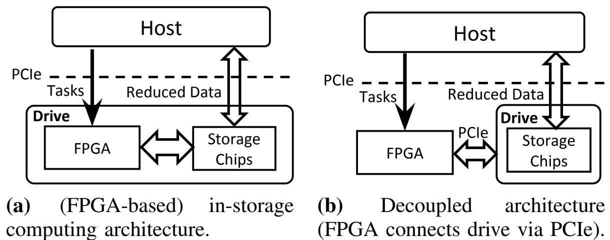
In addition to the compiler, EISC also provides the system-level simulation framework to accelerate the development iteration. It supports both C/C++-level and RTL-level simulations.

2) Host APIs: Table I presents the host APIs implemented in EISC. Rows 1 to 4 are the system parameter configuration APIs; users can invoke them to configure the bandwidth and latency of the interconnection (§IV-B2) and the emulated storage chip (§IV-B3). Row 5 is related to the kernel runtime; it allows users to send small runtime arguments to the accelerator kernel (§IV-C1). Rows 6 to 8 are used for the ISC read kernel. The kernel can read from a drive file; via 6 the data of the specified file is sent to the kernel $input_channel$ (§IV-C1). Alternatively, the kernel can read from multiple files; row 7 allows users to read from a vector of drive logical block addresses. The kernel execution output can be acquired by host via the function in row 8. Symmetrically, rows 9 to 11 are used for the ISC write kernel. 12 is used for debugging, which can issue $Peek_Req$ to the drive FPGA kernel as we discussed in §IV-C1; its return value corresponds to $Peek_Resp$.

D. Adaption to The AWS F1 Instance

One of our design goals is to make EISC publicly accessible to benefit the community. To achieve this goal, we adapt EISC to the AWS F1 instance [1] since it provides the necessary hardware environment for implementing the EISC drive.

First, the F1 FPGA board equips a Xilinx Virtex UltraScale+ FPGA board which could be used to build an EISC drive. The board is connected with host via PCIe Gen3 X16 which could deliver enough interconnection performance for our scenario (§IV-B1). Second, the F1 FPGA board equips four 16 GiB DDR4 DRAM chips (64 GiB in total). We have to statically partition them into emulated storage



(a) (FPGA-based) in-storage computing architecture. **(b)** Decoupled architecture (FPGA connects drive via PCIe).

Fig. 3: Two architectures used in our evaluation.

chips (§IV-B3) and accelerator kernel attached DRAMs (§IV-B4). The partition has to observe the following constraints: 1) Multiple DRAM chips have to be used to mimic the multi-channel storage chips; 2) Since the accelerator uses its DRAMs as the fast intermediate data buffer, their performance has to be no lower than the upper bound performance of the emulated storage chips. Given those constraints, we use two on-board DRAM chips to emulate the storage chips and use the rest of two DRAM chips to serve as the accelerator attached DRAMs. To reflect a practical scenario of the commodity drive, we limit the addressable space of the accelerator DRAMs to 4 GiB. Finally, we implement all EISC drive hardware logics (§IV-B) in the F1 FPGA board. The design is fully pipelined and is able to run at 250 MHz.

V. Evaluation

In this section we focus on answering the following questions:
Q1: How much programming effort is required to port an existing FPGA application into EISC?
Q2: For the applications used in evaluation, how do their speedups change under different system configurations (e.g., interconnection performance, storage drive performance)?
Q3: What kind of applications can benefit from the (FPGA-based) ISC architecture and what kind of applications cannot?
Q4: How to qualitatively or quantitatively discriminate between two kinds of applications in Q3.
Q5: For the applications that benefit from the ISC architecture, we further want to know how much fast drive bandwidth can they saturate. This information is critical for avoiding unnecessarily provisioning the drive bandwidth.

We answer **Q1** by porting existing FPGA programs to EISC and showing the corresponding efforts (§V-A). Among evaluated applications, *Gzip, SW* and *equal* are ported from the existing implementation from Xilinx.

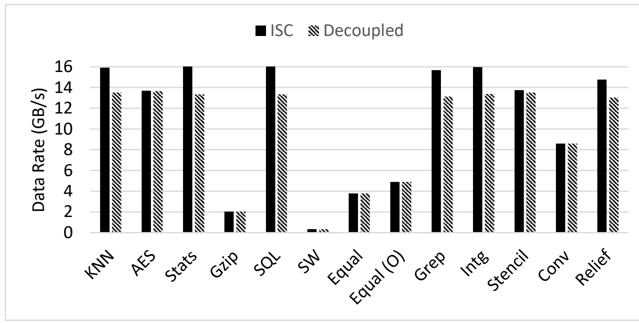
We answer **Q2** by applying EISC to 12 different applications (in Table II) to explore the benefits of ISC. We configure the drive and interconnection performance to EISC to measure the application performance under different system parameters (§V-B). The evaluation is done on the AWS f1.2xlarge instance.

We answer **Q3** by comparing the performance of the FPGA-based ISC architecture (Figure 3a) with the decoupled architecture (Figure 3b). We emulate the latter architecture in EISC by setting the drive performance into the same value of the host-drive interconnection performance (§V-B).

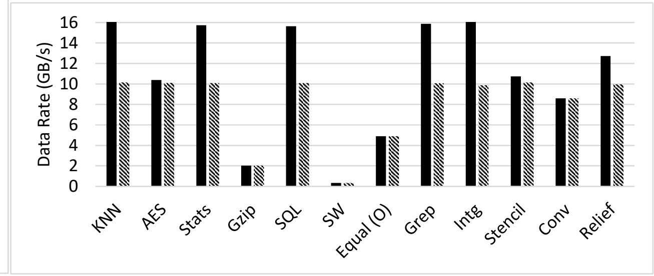
We answer **Q4** and **Q5** by conducting the qualitative analysis (§V-C) and quantitative analysis (§V-D) over the evaluation results.

A. Efforts to Port Existing FPGA Programs

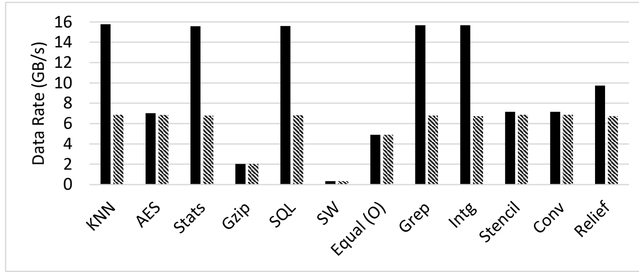
Table II lists the applications used in our evaluation, while Table III presents the programming efforts of porting existing FPGA applications into EISC. As we can see, the required code change is moderate. That is because users are only required to change the



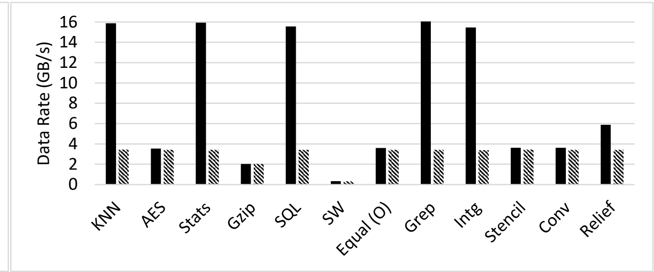
(a) Drive bandwidth = 16 GB/s, PCIe Gen3 X16.



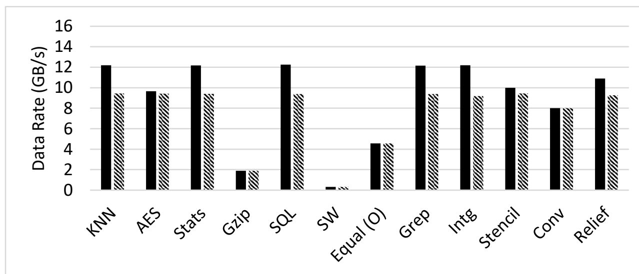
(b) Drive bandwidth = 16 GB/s, PCIe Gen3 X12.



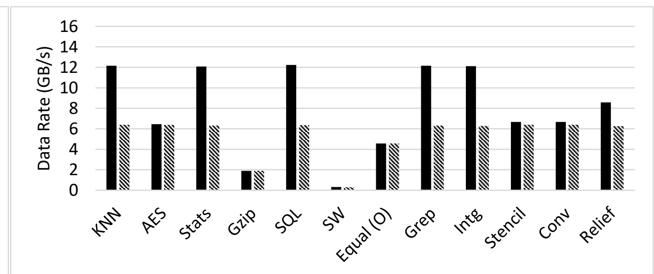
(c) Drive bandwidth = 16 GB/s, PCIe Gen3 X8.



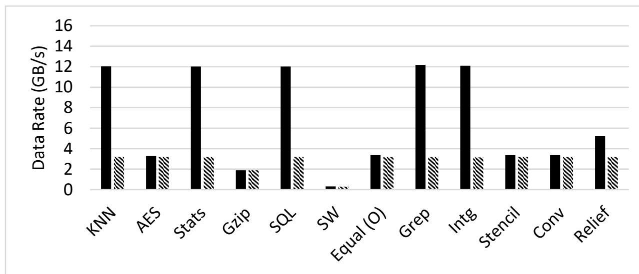
(d) Drive bandwidth = 16 GB/s, PCIe Gen3 X4.



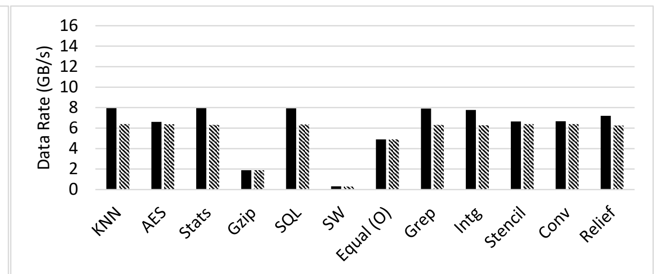
(e) Drive bandwidth = 12 GB/s, PCIe Gen3 X12.



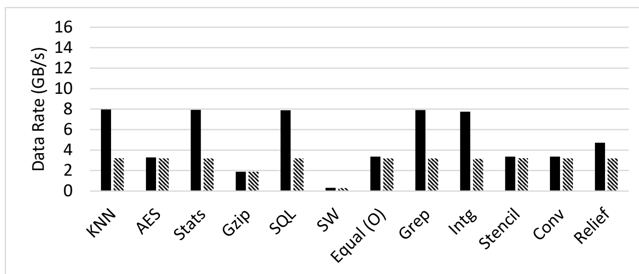
(f) Drive bandwidth = 12 GB/s, PCIe Gen3 X8.



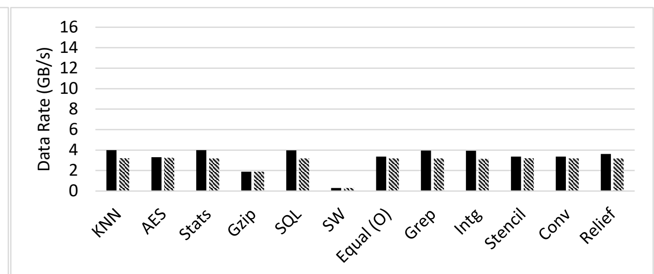
(g) Drive bandwidth = 12 GB/s, PCIe Gen3 X4.



(h) Drive bandwidth = 8 GB/s, PCIe Gen3 X8.



(i) Drive bandwidth = 8 GB/s, PCIe Gen3 X4.



(j) Drive bandwidth = 4 GB/s, PCIe Gen3 X4.

Fig. 4: Compare the end-to-end application performance between ISC architecture and decoupled architecture under different system settings.

Application	Description	Input File Size (GiB)
KNN	K-nearest neighbors, K = 32.	28
AES	128-bit AES ECB mode.	30
Stats	Per row statistical analyses.	24
Gzip	A classical compression algorithm.	4
SQL	A SQL query with select, where, sum, group by and order by.	30
SW	A local sequence alignment algorithm.	30
Equal	The image histogram equalizer.	30
Grep	String matching over large text.	30
Intg	Combine data from multiple sources.	30
Stencil	The 3x3 image stencil operation.	30
Conv	GEMM-based convolution algorithm.	30
Relief	A classical feature selection algorithm.	15

TABLE II: This table introduces the applications used in our evaluation.

Applications	Orig. LoC	Modified Loc	Modified Ratio
Gzip	883	75	8.5%
SW	760	68	9.0%
Equal (unoptimized)	172	28	16.5%
Equal (optimized)	207	30	14.5%

TABLE III: The programming efforts of porting existing FPGA applications.

ISC-friendly	KNN, Stats, SQL, Grep, Intg, Relief
ISC-unfriendly	AES, Gzip, SW, Equal(O), Stencil, Conv

TABLE IV: The applications in Table II can be classified into ISC-friendly (in which the ISC architecture can outperform the decoupled architecture) and ISC-unfriendly.

interfacing code to adapt the original program into the EISC interface; the code related to the computation logic is completely unmodified. Therefore, even users who do not understand the original FPGA code are able to adapt the program into EISC.

The original implementation of *equal* is not very optimized. It does not apply coarse-grained pipelining to overlap the load, compute and store stages. For the performance consideration, in the following evaluation we optimize the original code to apply pipelining. Thus readers will see *equal (O)* instead of *equal*.

B. ISC Architecture Versus Decoupled Architecture

We compare the performance of the ISC architecture against the decoupled architecture for applications in Table II. In the evaluation, we explore all the system performance settings (drive bandwidth = D, PCIe setting = P), where $D \in \{4, 8, 12, 16\}$ GB/s, $P \in Gen3\{X4, X8, X12, X16\}$, and bandwidth($P$) $\leq D$. We impose the last requirement since there is no benefit in providing higher interconnection bandwidth than the storage drive bandwidth. The evaluation results are shown in Figure 4. Despite the system setting, ISC architecture constantly outperforms decoupled architecture for a subset of applications. With this observation, we classify the applications into two categories in Table IV: ISC-friendly and ISC-unfriendly.

C. The Rule of Thumb for Choosing ISC Applications

We first investigated the ISC-unfriendly applications using the debugging/monitoring interface of EISC (introduced in §IV-C1 and §IV-C2). We discovered that they are bottlenecked by two different factors which will be discussed in the following.

Interconnection bound. Let D_I denote the size of the data transferred through the host-drive interconnection bus. D_D denotes the size of data read/written from/to the storage drive. We measure D_I and D_D, and calculate the relative data ratio = D_D/D_I, see Figure 5. For $AES, Equal(O), Stencil, Conv$, their relative data ratio = 1, which means $D_D = D_I$. In this case, though the internal drive bandwidth is higher than the interconnection bandwidth, we can not leverage it; the reason is that we need to transfer the same amount of data over the slow interconnection which bounds the whole kernel execution.

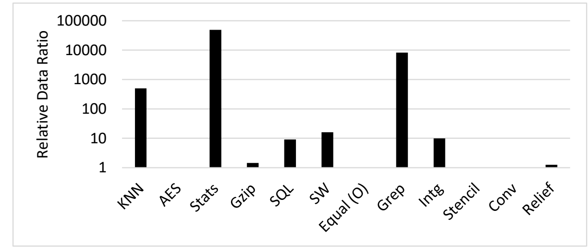


Fig. 5: The relative data ratio = data amount over storage drive / data amount over host-drive interconnection. *The hidden bar means ratio = 1.*

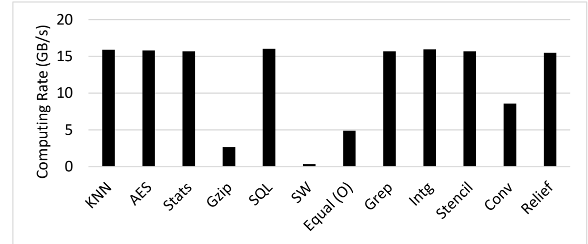


Fig. 6: The computing rates of different drive kernels.

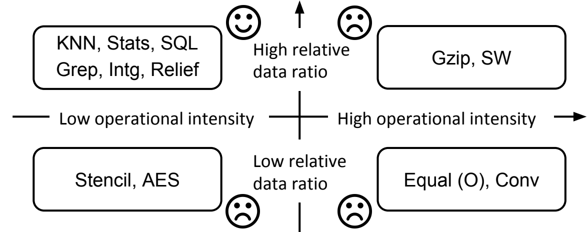


Fig. 7: The applications used in the evaluation (Table II) can be classified into four categories. Among them, the upper-left category is suitable for in-storage computing while others are not suitable.

Computation bound. We extract the time spent on drive kernel execution (which deducts the drive read time and output transferring time). With this, we calculate the drive kernel computing rate presented in Figure 6. As we can see, the computing rates of *Gzip* and *SW* are lower than 4 GB/s; they cannot saturate the storage drive bandwidth under all of our evaluation settings (§V-B). Therefore, these applications are bottlenecked by their kernel computing rates. The computing rate of $Equal(O)$ is about 5 GB/s which indicates its potential to achieve better performance in the ISC architecture when the storage bandwidth is 4 GB/s. However, the evaluation results in Figure 4j differ from this inference. The reason is that $Equal(O)$ has already been bottlenecked by the interconnection performance—as we discussed in the last paragraph. It is the same case for *Conv*.

Putting these two factors together, we are able to classify applications into four categories (see Figure 7). The X-axis is the operational intensity which is defined in the famous roofline model paper [30]. Here, for FPGA, we slightly revise the definition of operational intensity as *the number of cycles to process an input read.* [2] The Y-axis is the relative data ratio. The rule of thumb for

[2]We take the effect of pipelining into account. If an FPGA design is pipelined and has initiation interval = 2. Though, it may take more than 2 cycles to process an input, say 8 cycles, we use the number 2 when calculating the operational intensity.

choosing ISC-friendly applications is that *the application should have a low operational intensity (otherwise it will be bottlenecked by the computation) and a high relative data ratio (otherwise it will be bottlenecked by the slow interconnection bus)*.

D. The Analytical Model for Pipelined Drive Kernels

Based on the intuition in §V-C, we build an analytical model to gain system-level insights [32]. First, we assume the FPGA developers construct a load-compute-store pipeline to overlap the execution of these three stages. Notably, this is actually a common practice in writing FPGA programs, e.g., [11]. Second, we focus on modeling the read kernel (§II-A); we leave out the model of the write kernels (§II-B) which is symmetric.

1) $T_{ISC} = max(T_{Load}, T_{Comp}, T_{Store})$ Pipeline execution.
2) $T_{Load} = Size_{DriveData}/BW(Drive)$ Load drive data.
3) $T_{Store} = Size_{PCIeData}/BW(PCIe)$ Write output to PCIe.
4) $T_{Comp} = \delta Num_{Input}/Freq_{FPGA}$ δ is the operational intensity.
5) $Num_{Input} = Size_{DriveData}/Width_{Data}$ The input data count.

$Width_{Data}$ is the width of the kernel interface which means the number of bytes that can be acquired by the kernel per cycle. In EISC, $Width_{Data} = 64$ bytes (Listing 1). Combine 1) to 5), we get

$$T_{ISC} = max(\frac{Size_{DriveData}}{BW(Drive)}, \frac{Size_{DriveData} \cdot \delta}{Width_{Data} Freq_{FPGA}}, \frac{Size_{PCIeData}}{BW(PCIe)})$$

$$= Size_{DriveData} \cdot max(\frac{1}{BW(Drive)}, \frac{\delta}{Width_{Data} Freq_{FPGA}}, \frac{1}{\gamma BW(PCIe)})$$

where $\gamma = Size_{DriveData}/Size_{PCIeData}$ which denotes the relative data ratio (§V-B). Similarly, for the decoupled architecture, we have $T_{Decoupled} = Size_{DriveData} \cdot$

$$max(\frac{1}{BW(PCIe)}, \frac{\delta}{Width_{Data} Freq_{FPGA}}, \frac{1}{\gamma BW(PCIe)})$$

Given the fact that $BW(Drive) \geq BW(PCIe)$, $T_{ISC} \leq T_{Decoupled}$ is equivalent to

$$\frac{1}{BW(PCIe)} > max(\frac{1}{\gamma BW(PCIe)}, \frac{\delta}{Width_{Data} Freq_{FPGA}})$$

whose necessary and sufficient condition is:

$$\gamma > \gamma^* = 1 \text{ and } \delta < \delta^* = Width_{Data} \cdot Freq_{FPGA}/BW(PCIe) \quad (1)$$

Equation (1) shows the condition when an application is able to benefit from the ISC architecture. However, it does not answer another critical question, i.e., what is the threshold of the storage bandwidth that can be effectively leveraged by the application? Obviously, after a critical point, the application can no longer saturate the storage bandwidth. The critical point is determined by the following formula:

$$\frac{1}{BW(Drive)} > max(\frac{1}{\gamma BW(PCIe)}, \frac{\delta}{Width_{Data} Freq_{FPGA}})$$

whose necessary and sufficient condition is:

$$\gamma > \gamma^* = \frac{BW(Drive)}{BW(PCIe)} \text{ and } \delta < \delta^* = \frac{Width_{Data} \cdot Freq_{FPGA}}{BW(Drive)} \quad (2)$$

1) The Effectiveness of The Model: For the input-independent applications, e.g., *KNN, AES*, their γ and δ are irrelevant to the input data and can be analyzed statically. For the input-dependent applications, e.g., *Gzip, SQL*, their γ and δ are relevant to the input data; we can use simulation or run a sampling to get their γ and δ. We evaluate formula (1)(2) over all settings in Figure 4. The γ of evaluated applications can be found in Figure 5. We use the computing rate shown in Figure 6 to calculate $\delta = rate/(Freq_{FPGA} \cdot Width_{Data})$.

The evaluation of equation (1) is mostly the same as the process in §V-C, we leave it out due to the space constraint. For equation (2), since it is more strict than equation (1), we only evaluate it over three ISC-friendly applications: *SQL, Intg,* and *Relief*. These applications have the same frequencies, 250 MHz, in our implementation. In the evaluation we fix the PCIe setting as X1 (about 840 MB/s bandwidth), and change the drive bandwidth from 1 GB/s to 16 GB/s with the step = 1 GB/s. Results are shown in Figure 8. As we can see, *SQL* and *Intg* perfectly matches the theory: only when equation (2) can they get a better performance by increasing the drive bandwidth. However, *Relief* seems to be an exception case. The underlying cause is introduced in §V-D2.

2) The Limitation of The Model: Since the model assumes the pipelined execution of the FPGA kernel, it will not be accurate if the kernel is not constructed in a pipelined fashion. There are two cases where we found that the model does not work.

a) The kernel is not optimized.: In the setting of [*Drive bandwidth = 8 GB/s, PCIe Gen3 X4*], application *Equal* will not benefit from the ISC architecture according to our model. For the unoptimized version (i.e., the original version, see §V-A) of *Equal*, its measured performance is not shown in Figure 4 for disambiguation. However, we discovered that it does run faster under the ISC architecture. The reason is that it is not constructed in a perfect pipelined fashion, and there is some non-overlapped execution among load, compute and store stages. After applying pipelining, as we have seen, *Equal (O)* in Figure 4 behaves as expected.

b) Application inherently has a non-pipelined execution region.: *Relief* falls into this category. Its algorithm consists of two stages: training and filtering. The training stage goes through the sampled input and calculates the weight array; here it only involves the drive access and the kernel computation. Thus, during this stage, PCIe access time is inherently not overlapped with the drive access time. We can easily extend equation (2) to take this case into account.[3] However, it is more or less an application-specific process; we keep the original equation for generality.

VI. Related Work

a) In-Storage Computing: There is a rich literature on in-storage computing that tries to bridge the performance gap between the storage drive and the interconnection bus. A major portion of work explores the chance of offloading tasks to the existing drive-embedded CPU, which is originally designed for the firmware execution. Though they demonstrate some benefits on data analytics [20], key-value store [17] and object deserialization [29], as suggested in [28], [8], the drive-embedded CPU is generally too weak to saturate the high drive rate. After being equipped with the specialized hardware, considerable performance improvements on the SQL query [14], transaction [9] and file system operations [6] are achieved. However, their architectures only focus on specific workloads so that the whole system must be redesigned for other application scenarios. To simultaneously achieve high programmability and high performance, researchers also explore the possibility of using GPU [7] or FPGA [12] as the in-storage processing unit. One big disadvantage of the GPU-based ISC system is that it can significantly compromise the power efficiency of the storage drive, whose original power consumption is less than 10 watts. Using multiple low-power embedded GPUs instead of a full-fledged GPU might be a viable solution; nevertheless, no existing work has been explored in this direction. On the contrary, the FPGA-based ISC system is able to provide satisfied power efficiency with an additional 15 Watts per

[3]For example, we can add a leading term in the time formula (i.e., T_{ISC} and $T_{Decoupled}$ in §V-D) to reflect the unoverlapped portion of drive access time.

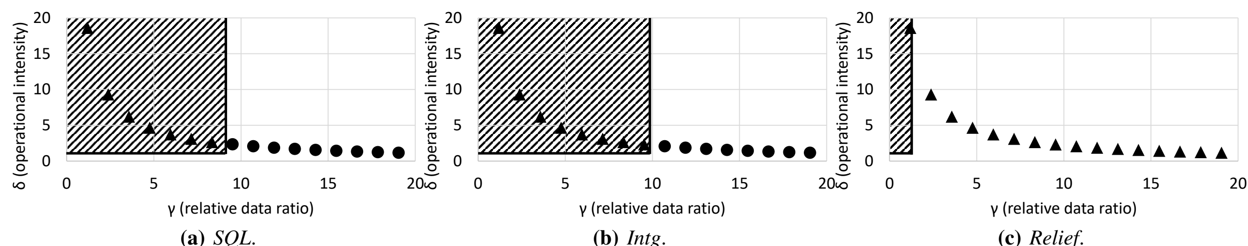


Fig. 8: We evaluate equation (2) by fixing the PCIe setting into X1, and changing the drive performance from 1 GB/s to 16 GB/s with the step = 1 GB/s. By changing the parameters, we draw (δ^*, γ^*) as the data points in the graph. For each applications, we draw $\gamma < \gamma_{APP}$ and $\delta > \delta_{APP}$, which forms a shaded area. For the data points (drive performance settings) that can still get better application performance by further increasing the drive performance, we mark them as triangles; otherwise we mark them as circles. According to equation (2), data points in the shaded area should be triangles and others should be cricles. We found *SQL* and *Intg* perfectly match the theory, whereas *Relief* does not match the theory at all.

flash board [19]. Our previous work [25] provides a programming abstraction and the system-level integration for FPGA-based ISC.

b) End-to-End System Emulation: Different than the functional simulation software like [5], EISC is a system-level emulation platform. We build a PCIe-based EISC drive and the corresponding software stack. The emulation in EISC is performed in an end-to-end fashion, bringing users an illusion that they are working on a real system prototype. A similar methodology has also been applied in other work, including system-level emulation of Intel Software Guard Extensions (SGX) [16], container-based emulation of the cloud-scale network [22], VMM-based emulation of hardware transactional memory [27], Linux kernel support for emulating persistent memory [4], etc.

VII. Conclusion

We present EISC, an open-source, publicly accessible system-level ISC emulator. By emulating the FPGA-based ISC architecture, it enables users to flexibly explore the benefits of ISC under different applications, drive performance and interconnection performance. The efforts for porting existing FPGA programs into EISC are moderate. Guided by the experiment results from EISC, we propose the rule of thumb for choosing ISC-friendly applications as well as an analytical model for pipelined drive kernels. We evaluate their effectiveness in 12 common workloads. We also show the limitations of our model in some exceptional cases, and present an analysis of the underlying cause.

Acknowledgements

We would like to thank anonymous ICCAD'19 reviewers for their 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, 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] Amazon EC2 F1 Instances. *https://aws.amazon.com/ec2/instance-types/f1/*.
[2] AXI Reference Guide. *https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf*.
[3] Emulab. *https://www.emulab.net/portal/frontpage.php*.
[4] Persistent Memory Wiki. *https://nvdimm.wiki.kernel.org/start*.
[5] N. Binkert et al. The Gem5 Simulator. *SIGARCH Comput. Archit. News*, 39(2), August 2011.
[6] A.M. Caulfield. Providing Safe, User Space Access to Fast, Solid State Disks. in ASPLOS, 2012.
[7] B.Y. Cho et al. XSD: Accelerating MapReduce by Harnessing the GPU inside an SSD. in WoNDP, 2013.
[8] S. Cho et al. Active Disk Meets Flash: A Case for Intelligent SSDs. in ICS, 2013.
[9] J. Coburn. Presentation of "From ARIES to MARS: transaction support for next-generation, solid-state drives", in SOSP, 2013. *http://sigops.org/s/conferences/sosp/2013/talks/coburn_mars_se04_04.pptx*.
[10] J. Cong et al. PARADE: A cycle-accurate full-system simulation Platform for Accelerator-Rich Architectural Design and Exploration. in ICCAD, 2015.
[11] J. Cong et al. Automated accelerator generation and optimization with composable, parallel and pipeline architecture. in DAC, 2018.
[12] A. De et al. Minerva: Accelerating Data Analysis in Next-Generation SSDs. in FCCM, 2013.
[13] J. Do et al. Query Processing on Smart SSDs: Opportunities and Challenges. in SIGMOD, 2013.
[14] B. Gu et al. Biscuit: A Framework for Near-data Processing of Big Data Workloads. in ISCA, 2016.
[15] L. Guo et al. Hardware acceleration of long read pairwise overlapping in genome sequencing: A race between fpga and gpu. in FCCM, 2019.
[16] P. Jain et al. OpenSGX: An Open Platform for SGX Research. in NDSS, 2016.
[17] Y. Jin et al. KAML: A Flexible, High-Performance Key-Value SSD. in HPCA, 2017.
[18] I. Jo et al. YourSQL: A High-performance Database System Leveraging In-storage Computing. *Proc. VLDB Endow.*, 9(12), August 2016.
[19] S.W. Jun et al. BlueDBM: An Appliance for Big Data Analytics. in ISCA, 2015.
[20] G. Koo et al. Summarizer: Trading Communication with Computing Near Storage. in MICRO-50, 2017.
[21] Bojie Li et al. Kv-direct: High-performance in-memory key-value store with programmable nic. in SOSP, 2017.
[22] H.H. Liu et al. CrystalNet: Faithfully Emulating Large Production Networks. in SOSP, 2017.
[23] Yuanwei Lu et al. Memory efficient loss recovery for hardware-based transport in datacenter. in APNet, 2017.
[24] Z. Ruan et al. ST-Accel: A High-Level Programming Platform for Streaming Applications on FPGA. in FCCM, 2018.
[25] Z. Ruan et al. INSIDER: Designing In-Storage Computing System for Emerging High-Performance Drive. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, 2019.
[26] S. Seshadri et al. Willow: A User-Programmable SSD. in OSDI, 2014.
[27] M. Swiech et al. VMM Emulation of Intel Hardware Transactional Memory. in ROSS, 2014.
[28] D. Tiwari et al. Active Flash: Towards Energy-Efficient, In-Situ Data Analytics on Extreme-Scale Machines. in FAST, 2013.
[29] H.W. Tseng et al. Morpheus: Creating Application Objects Efficiently for Heterogeneous Computing. in ISCA, 2016.
[30] S. Williams. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4), April 2009.
[31] L. Woods et al. Ibex: An Intelligent Storage Engine with Support for Advanced SQL Offloading. *Proc. VLDB Endow.*, 7(11), July 2014.
[32] P. Zhou et al. Doppio: I/o-aware performance analysis, modeling and optimization for in-memory computing framework. in ISPASS, 2018.