Accessible Near-Storage Computing with FPGAs

Robert Schmid
Hasso Plattner Institute
University of Potsdam

Max Plauth
Hasso Plattner Institute
University of Potsdam

Lukas Wenzel
Hasso Plattner Institute
University of Potsdam

Felix Eberhardt
Hasso Plattner Institute
University of Potsdam

Andreas Polze
Hasso Plattner Institute
University of Potsdam

Abstract

Data transfers impose a major bottleneck in heterogenous system architectures. As a mitigation strategy, compute resources can be introduced in places where data occurs naturally. The increased diversity of compute resources in turn affects programming models and practicalities of software development for near-data compute kernels and raises the question of how those resources can be made accessible to users and applications.

We introduce the *Metal FS* framework to improve the accessibility of FPGA-based *near-storage accelerators*: Firstly, we present a near-storage-compute-aware file system that enables self-contained, reusable compute kernels to operate on the granularity of file data streams. Secondly, we provide an integrated build process for FPGA overlay images that starts with the acquisition of compute kernels through a package manager and finally allows to dynamically configure near-storage compute pipelines consisting of them. Thirdly, we integrate the framework into Linux as a file system driver and repurpose Unix Pipes as a well-known operating system primitive to orchestrate near-storage compute pipelines.

CCS Concepts • **Information systems → Storage architectures**; • **Software and its engineering → Development frameworks and environments**.

Keywords Near-Storage Computing, Near-Data Processing, FPGA, Programming Model, Data-Flow Paradigm

ACM Reference Format:
Robert Schmid, Max Plauth, Lukas Wenzel, Felix Eberhardt, and Andreas Polze. 2020. Accessible Near-Storage Computing with FPGAs. In *Fifteenth European Conference on Computer Systems (EuroSys '20), April 27–30, 2020, Heraklion, Greece.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3342195.3387557

EuroSys '20, April 27–30, 2020, Heraklion, Greece

ACM ISBN 978-1-4503-6882-7/20/04…$15.00
https://doi.org/10.1145/3342195.3387557

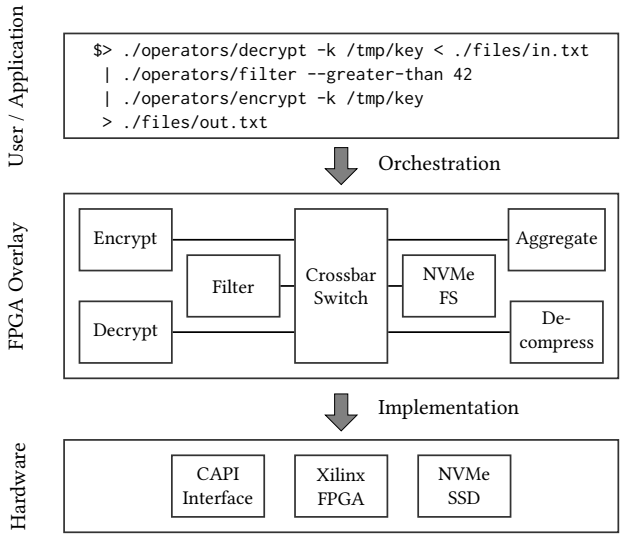

Figure 1. Shell commands issued by a user or application are executed by orchestrating a pipeline of pre-defined functional elements of a coarse-grained FPGA overlay. The overlay can be generated for most CAPI-enabled FPGAs.

1 Introduction

In data-intensive applications, the von Neumann bottleneck inhibits further performance improvements in todays computer systems as data cannot be transferred to CPUs fast enough. Advancements in coupling compute units more closely with memory as well as faster non-volatile storage technologies drive the development of near-data processing architectures, where application-level computations are performed in proximity to the data source [29].

Moreover, physical constraints increasingly limit the potential of performance improvements for general purpose processors. This trend is made clear by the end of Dennard Scaling and the current slowdown of Moore's law [8]. Thus, the development of application-specific compute accelerators receives increasing attention, promising superior performance per watt characteristics.

In comparison to GPUs, which are well established for application acceleration, the memory bandwidths available to FPGA platforms are catching up, as Fang et. al [10] describe in their survey paper on future opportunities for accelerating in-memory databases with FPGAs. However, they conclude that a central concern towards FPGA adoption in the field remains that most of todays database engineers are not trained to create low-level hardware circuit implementations.

Addressing this concern in general, different approaches improve the accessibility of FPGA development for software programmers, including high-level synthesis compilers, standardized programming interfaces (e.g. OpenCL [28]) as well as dynamically reconfigurable FPGA overlays [32]. However, none of these approaches are considering the use of FPGAs in near-storage computing scenarios.

In this paper, we present *Metal FS*, a data-flow oriented framework that makes near-storage computing facilities of combined FPGA+NVMe devices available to a wide audience. We propose the concept of a standardized, self-contained and composable *Operator* as the basic functional unit in Metal FS. Both operators and the storage media are exposed on the file system level, making them accessible to both users and applications. Furthermore, Metal FS enables developers to easily implement custom operators based on a streamlined development worflow based on Vivado HLS.

Figure 1 gives an overview about the features of Metal FS: Users and applications compose near-storage compute pipelines from operators using pipe expressions. Afterwards, Metal FS delegates the requested data transformation to a coarse-grained FPGA overlay. The required overlay image is synthesized from user-provided operator sources and transparently handles the interaction with CAPI and NVMe memory interfaces.

To provide more context to our implementation, the next section summarizes related lines of work. Afterwards, section 3 presents the concept of Metal FS; section 4 goes in detail about the implementation. Finally, section 5 shows that the proposed accessibility improvements go well with creating performant and energy-efficient near-storage compute accelerators and we give a final conclusion in section 6.

2 Related Work

Metal FS adopts concepts from previous research on near-memory computing, user-space I/O, FPGA overlays and operating system integrations of reconfigurable hardware.

Near-memory compute architectures are a focus of recent research that revolves around innovations in memory technology, new hardware interconnect protocols and requirements of data-intensive applications. Singh et al. [29] provide an overview and classification of previous works in the field. The considered memory technologies range from traditional non-volatile flash and DRAM to new 3D-stacked memories with integrated logic resources that allow for processing-in-memory (*PIM*). However, in typical near-memory compute architectures, a CPU, FPGA or ASIC is used as the processing unit which is separate from the memory itself.

A particular challenge towards mainstream adoption of near-memory compute is efficient cache coherent access to the virtual memory of the host process from the accelerator. Emerging interconnect standards such as CAPI [34]/ OpenCAPI [35] and CXL [27] facilitate the exchange of data between the CPU and computing memory by providing cache coherence.

Several approaches have already demonstrated the potential of near-storage computing facilities. Focusing on the acceleration of database workloads, the approaches by Woods et al. [39] and Do et al. [7] represent early prototypes for offloading database query processing tasks to near-storage computing facilities. Woods et al. employ an FPGA on the path between the host and a SATA SSD as compute resources, whereas Do et al. push application logic down to the SSD firmware. Seshadri et al. [26] introduce the concept of *storage processor units* (SPUs) embedded in storage devices, which can execute arbitrary code provided by the application. The work of Gu et al. [13] advances a data-flow model, where near-storage application logic is provided in chainable *SSDlets* that are executed on the ARM-based controller of a commercial off-the-shelf PCIe SSD. An FPGA-based approach to near-storage computing has also been presented by Ruan et al. [23]. Their prototype uses the concept of virtual files, which map to the contents of regular files and augment them with near-storage operations. All approaches have in common that their near-storage accelerators are managed by applications or custom runtime systems. Here, Metal FS argues that storage resources, even when augmented with compute facilities, should remain in the hands of the operating system.

Aside from applying near-storage computing, the access to NVMe-based SSD storage can be accelerated by creating specialized I/O drivers: The NVMeDirect framework [18] allows to directly interact with NVMe devices from user space, eliminating most of the overhead incurred by the kernel-space I/O stack. FastPath [33] additionally offloads the coordination of NVMe submission and response queues to the integrated FPGA of a Xilinx Zync SoC board. Data transfers from main memory to disk are bypassing the CPU and additional computations such as encryption on the data path are envisioned. Both frameworks provide raw disk access only and do not include the features of a full file system in contrast to Metal FS.

The widespread adoption of FPGAs as compute accelerators is gaining traction through a number of developments that lower the entry barrier for software developers to program FPGAs. Aside from High-Level Synthesis (*HLS*) languages and standardized programming models available on the market today, ongoing research into FPGA overlays [32]

promises to enable code portability across different FPGA platforms and drastically reduced design iteration times.

FPGA overlays serve as an intermediate device architecture that on one hand is the compilation target for high-level programming languages such as C++ or Java, and on the other hand has optimized implementations for different FPGA families, even supporting FPGAs from different vendors. They can be regarded as a form of resource virtualization within a single FPGA [37]. The overlay granularity can range from fine-grained structures (e.g. Koch et al. [19]) over coarse-grained reconfigurable arrays (*CGRAs*, e.g. DySer [16]) to complex processor-like overlays [32].

In the domain of operating system research, BORPH [31] and M3 [3] are previous efforts to create an OS that treats computations on reconfigurable hardware as equal counterparts of software processes. This includes the interoperability of either type of process using adaptations of the Unix Pipe concept. Whereas BORPH introduces the concept of hardware processes into the Linux kernel, M3 is a standalone operating system for heterogenous systems which includes a packet-switched FPGA overlay. Metal FS follows a more lightweight approach by augmenting unmodified Linux kernels to allow using Unix Pipes for FPGA orchestration.

3 Concept

Metal FS is an open source project[1] that aims to make near-storage computing accelerators accessible to a wider audience. Specifically, it defines an FPGA overlay architecture and augments the Linux file system so that no prior knowledge about hardware accelerators is necessary to compose custom data-flow processing pipelines. The abstract idea behind Metal FS has been outlined briefly in a preceeding poster [25]. In this section, we are going to elaborate on the concepts of Metal FS in greater detail.

In near-storage computing scenarios, it is suitable to provide the process specification for the FPGA at an even higher level of abstraction compared to previous overlay implementations: For instance, when transferring data from files on an NVMe storage device to the CPU, an intermediary FPGA accelerator typically performs coarse-grained operations such as encryption, compression or filtering on the fly.

Such elementary operations are reusable in different application domains and also allow for more complex data transformations through composition. Hence, the entire process specification consists of a chain of pre-defined operations to be applied to a data stream that is linked to a file on the NVMe storage file system.

From a developer's perspective, the productivity in creating FPGA accelerated applications is best, if previously defined high-level components can be reused and only minimal adaptations for a particular use case are required, if any. This insight also drove the design of the Unix operating system, which provides a set of standard utilities that *do one thing and do it well* and allows composition by using the pipe operator in a shell.

Inspired by this idea, Metal FS allows users to use standard shell syntax, including the pipe operator, as a means of programming FPGA-accelerated near-data computations that are mapped to pre-defined functional elements of a coarse-grained FPGA overlay.

The testbed used during development is comprised of an OpenPower S824L system equipped with a Nallatech 250S NVMe+FPGA accelerator card and supports the IBM CAPI 1 protocol, which allows for coherent host memory access from the accelerator. As upcoming generations of NVMe+FPGA devices are expected to offer improved performance, we decided to implement Metal FS on top of the CAPI Snap framework [21] which provides portability across future CAPI-enabled FPGA boards.

Metal FS consists of three major components that are introduced hereinafter.

3.1 Operators and FPGA Overlay

The Metal FS FPGA overlay is structured around *operators*, which are functional units that each perform a self-contained data stream transformation. When synthesizing a Metal FS overlay for a specific FPGA, users can choose a number of operators to be included in the design from a library of commonly used functions. During the build process, the hardware interfaces of the operators are connected to a central crossbar stream switch which controls their execution order at runtime. Consequently, the runtime parameterization of the overlay consists of the switch configuration as well as per-operator configuration data (e.g. encryption keys).

If an application specific functionality is missing from the operator library, users can define custom operators using common hardware description languages, as well as Vivado HLS (see Listing 1). By providing a standardized development environment for creating operators with Vivado HLS, Metal FS aims to make this process as streamlined as possible.

```
void op_example(stream &in, stream &out) {
    stream_element element;
    do {
        element = in.read();
        // insert operator logic here
        out.write(element);
    } while (!element.last);
}
```

Listing 1. Vivado HLS skeleton of a Metal FS operator.

[1] Available on GitHub: https://github.com/osmhpi/metal_fs

3.2 Near-Storage Compute Aware File System

Similar to NVMeDirect [18], Metal FS provides a user-space API for invoking data transfers from and to the NVMe drive. As in FastPath [33], the FPGA autonomously interacts with the NVMe controller. Instead of directly exposing physical storage locations however, Metal FS implements a full file system and therefore allows to interface with the storage resources using file-level operations.

The underlying implementation consists of both hardware and software components. At its core, each file is represented as a list of extents, where each extent consists of a starting block address and a length. Accordingly, the FPGA overlay contains a block mapper that translates logical file offsets into physical block addresses based on a file extent list.

All file system metadata is maintained on the CPU. Thus, file operations that modify the structure of the file system require CPU interaction to the benefit of greatly reducing the complexity of the hardware implementation. On the other hand, the FPGA autonomously conducts the performance critical read and write operations.

In order to enable near-storage computing, each file access internally passes through the crossbar switch that connects all operators. Thus, for read and write operations, the file system API allows to specify operators that should be applied to the file data stream as it is being transferred to or from the NVMe drive.

Finally, Metal FS includes a Fuse driver to mount the contents of the NVMe storage into the Linux file system. By default, no computations are performed during file accesses; however the driver can easily be configured to transparently include accelerated data transformations such as encryption or compression (cf. Figure 2).

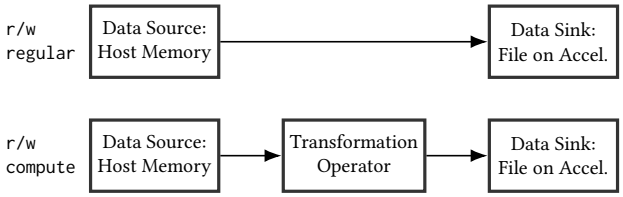

Figure 2. The `read` and `write` file system operations are leveraging the configurable FPGA overlay to apply arbitrary transformations on the data-path.

3.3 Orchestrating Near-Memory Computations using Unix Pipes

The goal of Metal FS is to improve accessibility of near-storage accelerators for developers and users alike. Hence, a key feature is the ability to use familiar shell syntax for orchestrating near-storage computations on files stored on the NVMe+FPGA device.

This is partly realized through the Fuse driver introduced in the previous section which makes the files in Metal FS accessible through the Linux file system. In addition, a mechanism is required to expose the operators from the FPGA overlay as executables to a standard Unix shell.

Instead of introducing a new type of binary format or even altering the definition of processes in Linux, we achieve this through proxy software executables that communicate with the file system driver which coordinates all FPGA processing.

```
./operators/decrypt -k /tmp/key  \
       < ./files/encrypted_values_on_nvme_ssd
   | ./operators/filter --gt 42 --lt 50  \
   | ./operators/aggregate  \
   > ~/aggregated_values
```

Listing 2. Example of defining a near-storage compute pipeline with Unix pipes.

4 Implementation

This section provides more details about the internal structure of the FPGA overlay, the available software APIs to execute operator pipelines and how near-storage compute pipelines are inferred from Unix Pipe expressions.

4.1 Overlay Architecture

The Metal FS FPGA overlay combines user-defined operators with an I/O subsystem for interfacing with different data sources as well as supporting components for host interaction. Furthermore, it includes performance counting instrumentation and benchmarking facilities.

NVMe-related components are optional and hence images for FPGA cards without integrated storage can be synthesized by parameterizing the build process accordingly. The overlay implementation relies heavily on the protocols defined in the Axi standard [2] and Axi-compatible IP provided by Xilinx Vivado.

Operators and Stream Switch

Each user-defined operator must provide input and output Axi Stream interfaces, an Axi Lite slave port for receiving configuration parameters as well as an interrupt output for signaling completion. The Vivado HLS compiler generates hardware blocks with this interface if the entry point of the operator function contains corresponding compiler directives. As operator configuration parameters, the framework currently supports boolean, integral and raw buffer values of arbitrary sizes.

Given a fixed clock rate, the size of a stream word limits the maximum data throughput of the overlay. Hence, users can configure the global stream word size to be used when building an overlay image. Operator implementations may adapt to this setting by using a constant that is defined at build time.

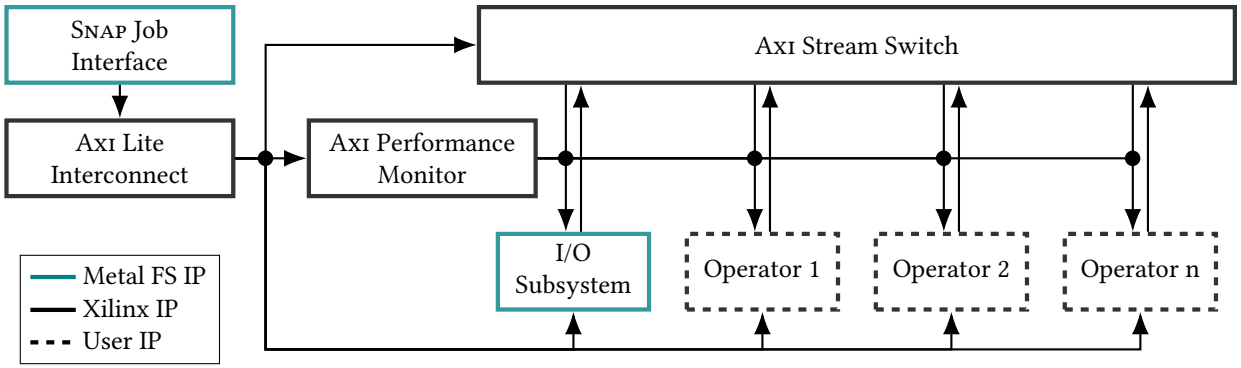

Figure 3. Internal structure of the FPGA overlay design. The Snap Job Interface receives configuration data from the host and configures the corresponding system components through an interconnect bus. A performance monitor observes the transactions on the stream interfaces.

To allow dynamic operator combinations in a synthesized image, the overlay includes a crossbar stream switch rather than relying on dynamic partial reconfiguration. The stream ports of each operator are connected to a Xilinx Axi Stream Switch that is additionally linked to the I/O subsystem (see Figure 3).

According to the specification of a Snap action, a Job Interface component responds to processing requests from the host. It orchestrates the overlay by forwarding the pipeline configuration defined by the host to the respective components through a Xilinx Axi Interconnect.

As part of developing an operator it is important to measure its performance in isolation and as part of a processing pipeline. Although performance estimates for hardware components can be obtained in simulation, the effective processing throughput and latencies often depend on the characteristics of production payloads and I/O devices that are too complex to emulate during simulation. Hence, the overlay image includes a Xilinx Axi Performance Monitor that captures cycle-accurate performance metrics on the stream interfaces of an operator selected at runtime.

I/O Subsystem

The I/O subsystem that is part of the overlay connects different address-based data sources with a pair of input and output Axi Stream interfaces (see Figure 4). Since the I/O subsystem currently only offers a single pair of streams, there can only be a single active pipeline configuration at a time.

A Xilinx Axi Interconnect maps the Host Memory and Card DRAM interfaces provided by Snap into a single address space. Afterwards, a Xilinx Axi DataMover receives commands for reading and writing chunks of data and performs the conversion into data streams. If the processing pipeline involves NVMe storage, the NVMe Arbiter handles concurrent streams of read and write commands and forwards them to the NVMe host component that is part of the Snap hardware framework. Since NVMe transfers in Snap always pass through the Card DRAM, data from NVMe can be accessed using the regular DataMover.

For operator- and I/O-benchmarking purposes, the overlay includes a data generator and discarding data sink. Both are able to fully saturate the stream interfaces that are linked to the main Axi Stream Switch (see Figure 3) and can replace the DataMover as a stream source and sink on demand.

All data transfer commands are issued by the Job Interface component which coordinates the pipeline execution. Not depicted in Figure 4 is the block mapper that translates logical file offsets into physical storage block numbers. It is implemented in Vivado HLS as part of the Job Interface and performs the translation before sending read and write commands to the NVMe arbiter. Additional implementation details of the block mapper are documented in [22].

Manifest-Driven Build Process

The Snap framework developed by the OpenPower Foundation includes a unified build interface based on Makefiles that covers the entire accelerator design process from assembling user IP over simulation to image synthesis.

Metal FS refines the first step of this process by automatically generating the overlay as a single Vivado *Block Design* from a Vivado Tcl script that is invoked by Snap. It composes the Block Design from user-defined operator IP, standard Xilinx components and Metal FS hardware blocks.

Users define an image manifest in a JSON file to parameterize the overlay build. Image parameters include the global stream width, the target FPGA type, contained operators and their identifiers.

Firstly, the process generates Vivado IP files from the sources of each referenced operator. Each operator must define an additional JSON manifest that specifies supported configuration options and the name of the top-level component in the operator sources. Metal FS currently provides integrated build support for Vivado HLS- as well as Verilog- and VHDL-based operators.

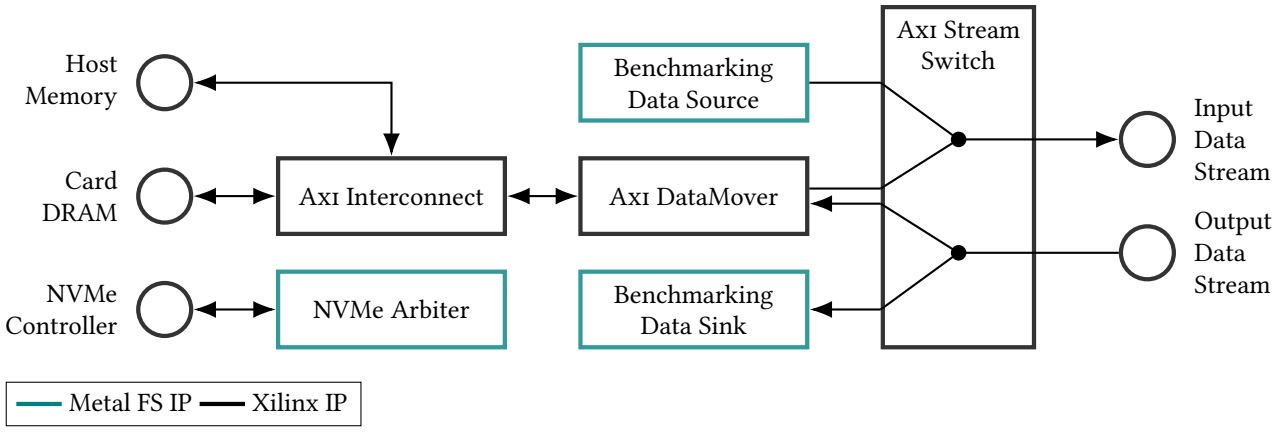

Figure 4. Flow of data in the Metal FS I/O subsystem. Host memory and DRAM are mapped at different offsets in the Axi Crossbar address space. Before starting a processing pipeline, the Data Selector Axi Stream Switch is configured to choose the data sources and sinks to be connected to the overlay stream switch.

Subsequently, the operator IP is instantiated in the Block Design together with a combined manifest for the image and all operators (see Figure 5). This allows the software to automatically discover the available operators in a deployed Metal FS FPGA image.

The build process also resolves operators that are published as `npm` packages. Originally built for Node.js libraries, `npm` is a package manager that works programming language independent. Hence, it is well suited to distribute hardware descriptions adhering to the standardized operator interface.

In order to streamline the setup of the development environment which requires specific versions of Linux distributions and Xilinx Vivado, we provide a ready-made Docker image that includes all necessary dependencies. Since Docker offers a runtime for all major operating systems, the development environment is also portable through Docker. Part of the image is the free Vivado WebPACK edition which allows to test Metal FS overlay images in simulation.

4.2 Near-Storage Compute Filesystem

To obtain a fully featured file system, a software component complements the block mapper that is part of the FPGA overlay. The file system implements a worst-fit memory allocation scheme and maintains indices of allocated memory regions and inodes, including permissions and other file attributes.

The underlying LMDB key-value store [36] provides transactional consistency and persistence of the metadata outside of the near-storage compute device. LMDB internally implements a memory-mapped B+-tree for each index table.

In Metal FS, the *inodes* table contains a mapping from inode ids to file attributes as well as extent lists for regular files, or buffers of directory entries in case of directories. Directory entries are pairs of filenames and inode ids.

For managing assignments of storage regions, two index tables exist: Firstly, the extents index keeps track of the current partitioning of the storage resources into free and occupied extents. Since the extents' start block offsets are used as the index keys, linear scans over the partitioning can be performed with constant time complexity which is useful for expanding the last extent of a file and to fuse adjacent extents when a file is deleted.

In a secondary index, an entry exists for each unoccupied extent, where the keys are combined of extent length and offset. The internal sorting of LMDB indices provides quick access to the largest unoccupied extent which is selected during worst-fit memory allocation.

For users of the file system, a basic C API (see Listing 3) provides functionality for creating and manipulating files without explicitly using near-storage compute. It resembles

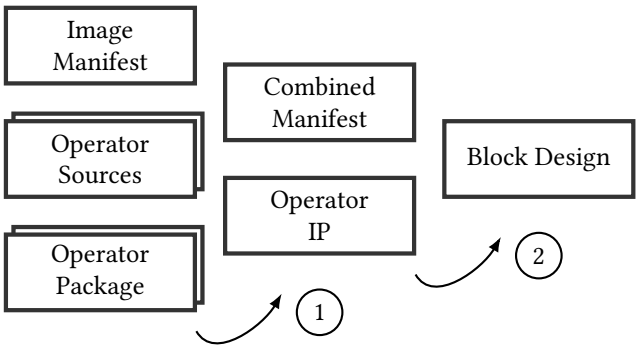

Figure 5. Inputs and artifacts in the Metal FS overlay build process. (1) Local operator sources and packages from `npm` are assembled into Vivado IP. (2) Together with a combined image manifest, the overlay Block Design is generated.

```
uint64_t file;
char data[] = "Hello World";
mtl_create("/test.txt", &file);
mtl_write(data, sizeof(data), file);
mtl_close(file);
```

Listing 3. Usage example of the C filesystem API. Based on a global configuration, near-storage computations are transparently performed during file accesses.

```
OperatorFactory factory;

Operator encryption =
    factory.createOperator("encrypt");
encryption.setOption("key", "secret");

char data[] = "Hello World";
Pipeline pipeline(
    std::move(encryption)/*, [...] */);
pipeline.run(
    DataSource(data, sizeof(data)),
    FileDataSink("/test.txt", sizeof(data))
);
```

Listing 4. Usage example of the operator pipeline API (C++) for explicit near-storage compute. As the data is being transferred from the host to the NVMe file, it is encrypted by the intermediary FPGA.

the file system operations in the C standard library and almost directly translates to the necessary callbacks to provide a Fuse user-space file system driver.

In contrast, a more flexible operator pipeline API for C++ supports using NVMe files as the inputs and outputs for multi-stage operator pipelines (see Listing 4). Other supported data interfaces are host memory, card DRAM and benchmarking data sources and sinks.

Using the manifest that is included in the FPGA overlay image, an `OperatorFactory` object discovers available operators. Once an operator reference is obtained from the `OperatorFactory`, necessary configuration options can be passed to the FPGA. Finally, the `Pipeline` object assembles the overlay configuration and invokes FPGA processing.

4.3 Proxy Executables for Unix Pipe Orchestration

The Fuse driver exposes proxy executables in the Linux file system under the names of the available operators. Users can connect the `stdin` and `stdout` file descriptors of these proxies arbitrarily. Internally, processing is delegated to the FPGA via the file system driver process.

Upon startup, each proxy process collects command line parameters and relevant environment variables. If a regular file is connected to the standard file descriptors, as indicated by `/proc/self/fd/{0,1}`, its path is compared with the closest file system mount point to the operator proxy file path in order to detect Metal FS-internal files.

In case the standard file descriptor resolves to a path of the format `pipe:[<id>]`, the *procfs* is linearly traversed to find a process with a matching standard file descriptor. If the found process executable path points to another operator proxy, its process id is associated with the operator input or output, respectively.

The communication between proxies and the file system driver is handled through a Unix domain socket that resides at a known location in the Fuse file system. Initially, the proxy (client) announces itself along with the previously collected data to the file system (server) using a pair of request and response Protobuf [12] messages. If necessary, the proxy and driver processes subsequently establish a shared memory region for double-buffered and copy-free exchange of payload data in chunks of 64 MiB. The necessary signaling for exchanging payload data is driven through another pair of Protobuf messages.

Once the file system driver has received messages from all proxy processes that were referred to by their process ids, it configures the operators according to the supplied command line options and invokes an operator pipeline that matches the Unix Pipe expression specified by the user. In the process, it handles the following special cases:

Two or more proxy processes connected to each other are translated into a multi-stage operator pipeline, thereby bypassing the actual Unix Pipes and therefore any involvement of the host CPU. Moreover, if proxy processes are connected to files residing within Metal FS, the driver instantiates a near-storage compute pipeline.

5 Evaluation

The evaluation is divided into a case study which integrates an existing accelerator implementation into Metal FS as well as performance measurements of the system components.

5.1 Case Study: Snappy Decompression

The goal of the Metal FS overlay architecture is to provide a programming model that suits a variety of near-memory compute use cases. Therefore, we investigated if existing stream-oriented accelerator implementations can be fitted into the operator interface by exemplarily adapting the Snappy Decompressor implementation by Fang et al. [9].

Snappy is a fast compression codec originally implemented by Google [11]. As part of a near-memory compute pipeline, it can help in alleviating input I/O bottlenecks. Fang's hardware implementation of Snappy is provided as a Snap action

and targets a CAPI 2-capable AlphaData ADM-PCIE-9V3 FPGA card at a clock rate of 250 MHz.

While the core decompressor logic consists of System Verilog code, the adaptation to the action interface required by Snap is implemented in VHDL. Internally, the decompressor provides 64 byte wide streaming interfaces for input and output data as well as register interfaces for receiving the compressed and decompressed data sizes as additional inputs to the algorithm. This structure fits well into Metal FS since it closely resembles the required operator interface.

By replacing the Snap wrapper, we adapted the implementation to work as a Metal FS operator. The new wrapper forwards the input and output data streams and exposes the payload size parameters as operator configuration options. Compared to the wrapper required for the Snap action interface (1560 lines of code), it is lightweight with only 329 lines of code. We synthesized a valid overlay image containing the Snappy Decompression Operator and also tested it in simulation where it showed identical performance compared to the original version.

This example shows that existing FPGA accelerator designs can be adapted to Metal FS with little effort, both for HLS-based implementations as well as HDL projects. Porting an algorithm to the Metal FS model provides value through re-use of data source and sink implementations including NVMe on supported FPGA cards, composability with other Metal FS operators and an easy interface for integration of the accelerated function into software or use on the command line.

5.2 Performance Evaluation

The remainder of this section presents benchmarking results that were obtained on a 20-core OpenPower8 S824L system with 1 TB RAM running Ubuntu Linux 16.04. Near-memory compute resources are provided by a Nallatech 250S card [20] that combines a Xilinx Kintex UltraScale XCKU060 FPGA with two Samsung M.2 NVMe SSD sticks[2] and supports the CAPI 1 interface.

All FPGA designs are clocked at 250 MHz. Although the system has four NUMA nodes with five 3.42 GHz cores each, the benchmark measurements are constrained to a single node to avoid the effects of remote memory accesses.

We evaluate the following performance characteristics: Firstly, the FPGA overlay architecture is investigated with regards to consumption of FPGA resources and data throughput. Secondly, we measure the throughput of a near-storage compute pipeline and provide an estimate of the possible energy savings compared to a CPU implementation. Finally, we benchmark the overhead of orchestrating operator pipelines using Unix pipes.

[2]Samsung part number MZ1LV96-HCJH

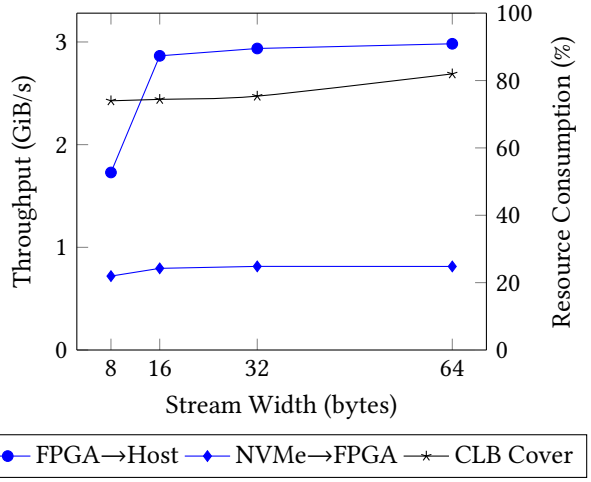

Figure 6. Throughput and resource consumption of overlay images with different stream widths. All images contain three placeholder operators. Throughput was measured with a 1 GiB data transfer. The translation of file offsets into physical blocks is part of the NVMe measurements.

5.2.1 Overlay Throughput and Resource Consumption

When configuring the data stream width of a Metal FS FPGA overlay image, users must consider the following tradeoff: A narrow stream width can limit the possible data throughput of the accelerator. On the other hand, I/O components take up more FPGA resources when using wider streams.

To evaluate this tradeoff for the CAPI 1-capable 250S card, we measured the data throughput and resource consumption of images with different stream widths. These images contain only the Metal FS overlay as well as a simple passthrough operator, that directly binds its input stream to its output stream. The throughput measurements are obtained by determining the runtime of the `Pipeline::run` invocation (see subsection 4.2) for a transfer volume of 1 GiB.

The CAPI 1 interface operates on top of PCIe 3.0 and the 250S card uses 8 lanes, resulting in a theoretical maximum bandwidth of 7.9 GB/s. Nevertheless, previous measurements have shown that CAPI 1 supports an effective bandwidth between 2.8 GB/s [38] and 3.5 GB/s [6].

The host transfer results (see Figure 6) show that the overlay is able to saturate the interface bandwidth at 3 GiB/s with a stream width of 16 bytes or more. At this point, the stream offers a bandwidth of $16\ \text{B} \cdot 250\ \text{MHz} = 4\ \text{GiB/s}$.

For NVMe transfers, a read throughput of 800 MiB/s is achieved in all configurations, as even the 8 byte stream supports a bandwidth of 2 GiB/s. The SSDs on the 250S card support a sequential bandwidth of 1000 MB/s (read) and 870 MB/s (write) [24]. These results show that, given a sufficient stream width, the overlay design itself imposes no major bottleneck on the data transfer performance. On the

other hand, it depends on the performance characteristics of the underlying CAPI SNAP environment.

The resource consumption of the reference design (see Figure 6) ranges between 75% and 80% of the configurable logic blocks (*CLB*) on the Xilinx XCKU060 FPGA. With a negligible resource footprint of the passthrough operator, the Metal FS overlay leaves only a relatively small share of the logic resources available to user operator logic. Nevertheless it is interesting, that over a wide range of stream widths between 8 B and 64 B, the resource consumption changes only by about 5%. This result suggests that the majority of hardware resources are not consumed by the Metal FS stream infrastructure. A finer analysis shows that the Power Service Layer (PSL) uses 23% and the Snap framework components 34% of the CLBs. Together, these components provide the Axi interfaces to host memory as well as the DDR4 and NVMe memories on the 250S card.

In contrast, the Metal FS I/O subsystem and stream switch contribute another 20% to 33% to the total CLB utilization, depending on the stream width. It is important to note that these percentages are not additive, as they denote a CLB cover: A CLB can be subdivided for use by more than one component of the image. Thus, a single CLB may be counted multiple times to the utilizations of different components, which explains the higher sum of individual percentages of about 90% in contrast to the actual 80% total utilization.

In newer generations of the CAPI protocol, the PSL is more lightweight so that more resources are available for user-defined logic. It should also be noted that the XCKU060 FPGA in the testbed is relatively small compared to most of the other FPGAs supported by Snap.

5.2.2 Near-Storage Compute Pipeline

For evaluating the performance and energy efficiency of Metal FS near-storage compute pipelines, we investigate the throughput of a filter and aggregation pipeline that processes a contiguous array of 8-byte integer values residing on NVMe storage. As part of the aggregation step, the sum, minimum, maximum and count of values are computed. This type of operation could be implemented as part of a columnar database, which computes column statistics for query plan optimization.

The FPGA overlay used for this benchmark has a stream width of 16 bytes and contains separate operators for filtering and aggregation. Both operators are implemented in Vivado HLS and are designed to fully saturate the stream bandwidth.

We performed measurements on 1 GiB of data which contains a uniform random distribution of integer values. Figure 7 shows the end-to-end throughput of the filtering operation as well as the combined filtering and aggregation operation.

When combining both operators, the operation processes data at the speed of the NVMe interface. In the filter-only case, more data needs to be transferred over the CAPI interface, which incurs additional communication latencies that have a minor effect on the throughput. Compared to a traditional CPU-based implementation, near-memory computing does not increase the throughput in our scenario since the bottleneck remains loading data from the NVMe SSD.

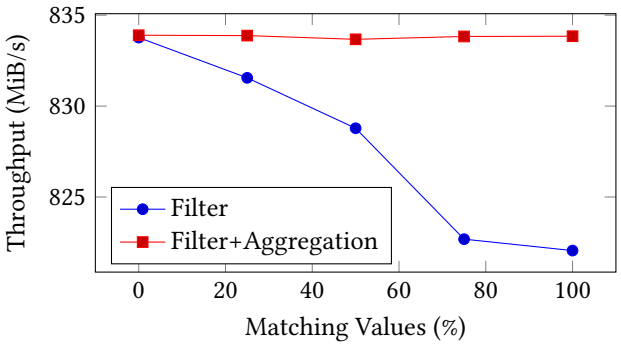

Figure 7. Throughput measurements for filtering and aggregating 1 GiB of NVMe data on the FPGA.

In a second step, we provide an estimation of the energy consumption of the operation performed on the near-storage FPGA in contrast to a traditional CPU implementation. In a large-scale server system, it is difficult to attribute the measured overall energy consumption to individual system components that perform a certain computation. Instead, to make a more general statement about the orders of consumed energy in the different implementations, we base the estimations on specifications published by the hardware vendors.

To be able to estimate the energy savings from employing near-storage compute resources, the maximum possible throughput of the FPGA kernel has to be compared to an equivalent CPU implementation. For the energy estimation we will assume that the power consumption of the FPGA is directly proportional to the data throughput, which is typically limited by the NVMe bandwidth. In order to determine the maximum throughput of the FPGA, we use the benchmarking data source and -sink to generate data in place, thereby excluding CAPI from the measurement.

As for the single-threaded CPU implementation, we again evaluate both the filter-only operation as well as a fused filter and aggregation, i.e. the intermediate set of filtered values is not materialized in memory. The software implementation is not manually tuned and is compiled using GCC 7 at optimization level 3 for the Power8 system.

Figure 8 shows that the data throughput on the FPGA is nearly always constant regardless of the filter selectivity. In contrast, the throughput of the CPU implementation depends heavily on the share of matching values, which can be explained by less effective branch prediction optimizations.

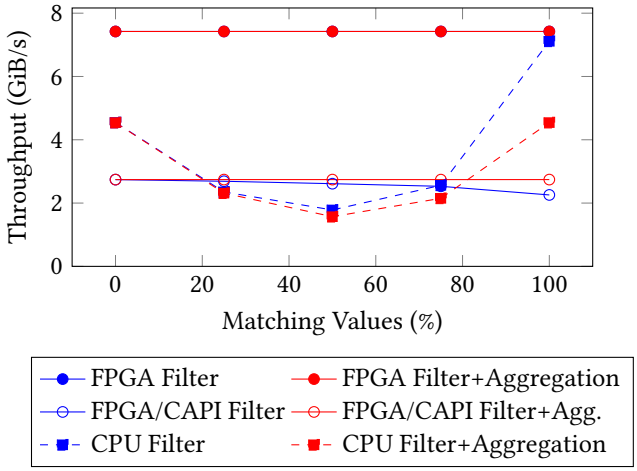

Figure 8. Throughput of CPU and FPGA based implementations operating on 1 GiB of random data. The FPGA measurements operate on data that is generated on the FPGA itself; they overlap at 7.4 GiB/s. For reference, the measurement of the FPGA accessing host memory through the CAPI interface is depicted as well.

Table 1. Estimation of energy consumption of the filtering operation in different scenarios.

Energy Consumption	CPU busy + No FPGA	CPU idle + FPGA	No CPU + FPGA
CPU	8.84 J	3.95 J	-
FPGA	-	2.10 J	2.10 J
Total	8.84 J	6.05 J	2.10 J

Between 20 to 80% of matching values, offloading the computation to the FPGA through CAPI improves the performance of the operation. With newer generations of CAPI, also the theoretical FPGA throughput of 7.4 GiB/s can be saturated by the host memory link [30].

We now take the measured processing times of the combined filter and aggregation operation as the basis for the energy estimation in the following. Aiming to consider an average case, we assume a 75% share of matching values (CPU: 465 ms, FPGA: 135 ms).

We assume the power consumption of the FPGA design at maximum throughput to be 15 W as reported by Xilinx Vivado and of a single POWER8 core[3] to be 19 W. The IBM Systems Energy Estimator [15] specifies a difference in total power consumption for the system in idle vs. fully utilized state of 210 W, which translates into power savings of 10.5 W per core. Table 1 shows the resulting energy estimates for different scenarios.

[3]According to the total TDP of the processor (part number 00UL864) [14]

The left column shows the baseline energy consumption in the traditional CPU implementation. If the system is underutilized, i.e. offloading the computation leads to the CPU idling, the overall energy consumption already decreases since the savings from frequency scaling overcompensate the additional energy consumption by the FPGA (middle column). If we further assume that the additional FPGA capacity allows to use a system with fewer CPU cores, we can eliminate the entire energy consumption of a core (right column), which results in an energy consumption of less than a quarter of the CPU baseline.

Provided that the number of CPU cores in computer systems is continuously growing over time, the potential of energy savings through near-storage computing can be realized by using less powerful CPU cores to achieve the same system capacity, as demonstrated by Becher et al. [4].

5.2.3 UNIX Pipes for Accelerated Computing

As one of the main contributions to improve accessibility of near-storage compute, Metal FS proposes to use standard shell expressions including the pipe operator to combine FPGA compute kernels with traditional software programs. In this section we investigate whether processes can exchange data over a UNIX Pipe fast enough to benefit from accelerated near-storage computing and if the increase in productivity justifies the overhead added by this abstraction.

For the experiments we use an overlay image with a passthrough operator and a stream width of 64 bytes. Additionally, a software utility generates 1 GiB of payload data to be used as the input for the pipeline. During the tests, we put the passthrough operators in profiling mode and compare the throughput measured by the Performance Monitor built into the overlay with the effective throughput on the host measured by the pipe viewer (*pv*) utility. Listing 5 shows the shell expressions used for the evaluation. Since the payload data crosses a pipe at most twice for any operator pipeline, the number of chained operators has no effect on the measurements.

The results in Table 2 show that UNIX Pipes are fundamentally suited to provide sufficient data throughput to saturate the FPGA accelerator connected via CAPI 1. If the payload traverses a CPU-FPGA boundary twice, the total overhead is 14%. In the near-storage compute scenarios, where data passes through a pipe only once, the overhead amounts to only 6 to 11%.

5.3 Discussion

Through UNIX pipes, Metal FS offers near-storage compute semantics to shell-scripting users, suggesting that awareness of near-storage compute may accelerate everyday scripting tasks. While this certainly depends on the use case, our stance is that the operating system should offer users an accessible interface to the available near-storage compute capabilities of the system.

```
# Host to FPGA
cpu_datagen 1Gi | pv  \
    | ./operators/passthrough --profile \
    > /dev/null

# FPGA to Host
./operators/datagen --size 1073741824  \
    | ./operators/passthrough --profile  \
    | pv > /dev/null

# Host to FPGA to Host
cpu_datagen 1Gi  \
    | ./operators/passthrough --profile  \
    | pv > /dev/null
```

Listing 5. Shell expressions used for determining the overhead of Unix Pipes. The `datagen` operator represents the benchmarking data source on the FPGA and connecting the passthrough output to `/dev/null` translates to discarding the output directly on the FPGA.

Table 2. Data rates measured on the FPGA compared to the end-to-end Unix Pipe throughput for different combinations of data sources and sinks during a 1 GiB data transfer.

Source	Host			FPGA		
Sink	FPGA GiB/s	Pipe GiB/s	%	FPGA GiB/s	Pipe GiB/s	%
Host	1.81	1.56	86%	2.55	2.28	89%
FPGA	2.57	2.41	94%	-		

In a broader sense, we can also consider application developers as users, who utilize a standardized set of operating system APIs. It is an open question whether near-storage compute semantics necessarily have to be a part of the OS API surface or if the operating system scheduler should take care of near-storage compute tasks internally.

The hardware and OS internals together with the OS API and the applications built on top of it resemble an hourglass model [5]: While the underlying hardware resources as well as the applications built on top permit a large degree of variance, the OS API should be stable and minimal. Minimality means to provide as few facilities as possible without omitting such facilities that are required or benificial for the applications using the API. Based on this model, the above question may be translated to ask whether there are applications that benefit from knowledge and control over data locality and placement of computations.

In current parallel data processing platforms and distributed filesystems, such as Hadoop and Apache Spark, there already is an application-level notion of horizontal data partitioning and locality in terms of where shards of data reside in a homogeneous set of worker nodes. Research from the high-performance computing domain also indicates that algorithms need to be aware of node-internal data motions [1] as well as other hierarchical aspects such as discrete data representations and tailored floating point precisions to achieve the necessary energy efficiency in the future [17]. These insights combine to suggest that there are applications that benefit from operating system APIs exposing the near-storage computing characteristics of the underlying system. A broad study on which particular types of applications can benefit in this way from near-storage computing infrastructure using Metal FS is planned as future work.

Metal FS Operators can be suitable primitives as part of a minimal operating system API with near-storage compute semantics. Since the interface does not express specifics about the underlying accelerator, the concept of pre-defined, runtime-composable compute kernels on the data path as implemented by Metal FS can be mapped to a variety of system architectures: It may support different FPGA accelerator platforms and interconnect technologies, as well as additional accelerator classes such as GPUs or domain-specific compute units. This opens up the near-storage operator concept to a wide range of platforms and application domains.

6 Conclusion

We have proposed Metal FS as a framework for creating applications that leverage FPGA+NVMe devices as near-storage compute accelerators. It is designed to reduce the effort to target heterogenous hardware during application development. This is achieved by providing a portable model and execution environment for FPGA compute kernels, a coherent build process for composing FPGA overlay images as well as an approachable programming and user interface based on established operating system concepts.

We have shown that with Unix Pipes the benefits of near-storage computing can be made accessible through a straightforward programming interface. Measurements have demonstrated that the provided abstractions are very lightweight and can support a wide variety of application scenarios. Finally, the Metal FS interface has proven valuable in parallel programming education, especially for inexperienced users.

Acknowledgements

We thank the anonymous reviewers and our shepherd, Tim Harris, for their valuable feedback. Also, we thank everyone at IBM who supported us during the project, including but not limited to: Jörg-Stephan Vogt, Frank Haverkamp, Bruno Mesnet, Alexandre Castellane, Sven Boekholt, Thomas Fuchs, Sven Peyer and Nicolas Mäding.

References

[1] Rabab Al-Omairy, Guillermo Miranda, Hatem Ltaief, Rosa M Badia, Xavier Martorell, Jesus Labarta, and David Keyes. 2015. Dense matrix computations on numa architectures with distance-aware work stealing. *Supercomputing Frontiers and Innovations* 2, 1 (2015), 49–72.

[2] Arm Limited. 2019. AMBA AXI and ACE Protocol Specification. (Data Sheet).

[3] Nils Asmussen, Marcus Völp, Benedikt Nöthen, Hermann Härtig, and Gerhard Fettweis. 2016. M3: A Hardware/Operating-System Co-Design to Tame Heterogeneous Manycores. *SIGARCH Computer Architecture News* 44, 2 (March 2016), 189–203.

[4] Andreas Becher, Florian Bauer, Daniel Ziener, and Jurgen Teich. 2014. Energy-aware SQL query acceleration through FPGA-based dynamic partial reconfiguration. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE.

[5] Micah Beck. 2019. On the Hourglass Model. *Commun. ACM* 62, 7 (June 2019), 48–57.

[6] Alexandre Castellane and Bruno Mesnet. 2017. SNAP Framework built on Power CAPI technology. (Presentation).

[7] Jaeyoung Do, Yang-Suk Kee, Jignesh M Patel, Chanik Park, Kwanghyun Park, and David J DeWitt. 2013. Query processing on Smart SSDs: Opportunities and Challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 1221–1230.

[8] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2012. Dark Silicon and the End of Multicore Scaling. *IEEE Micro* 32, 3 (May 2012), 122–134.

[9] Jian Fang, Jianyu Chen, Jinho Lee, Zaid Al-Ars, and H. Peter Hofstee. 2019. A Fine-Grained Parallel Snappy Decompressor for FPGAs Using a Relaxed Execution Model. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE.

[10] Jian Fang, Yvo T. B. Mulder, Jan Hidders, Jinho Lee, and H. Peter Hofstee. 2019. In-Memory Database Acceleration on FPGAs: A Survey. *The VLDB Journal* (Oct. 2019).

[11] Google. 2017. Snappy: A fast compressor/decompressor. (Website). https://github.com/google/snappy

[12] Google. 2019. Protocol Buffers. (Website). https://developers.google.com/protocol-buffers

[13] Boncheol Gu, Andre S Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, et al. 2016. Biscuit: A framework for Near-Data Processing of Big Data Workloads. *ACM SIGARCH Computer Architecture News* 44, 3 (June 2016), 153–165.

[14] IBM Corporation. 2016. POWER8 Processor Datasheet for the Single-Chip Module. (Data Sheet).

[15] IBM Corporation. 2019. IBM Systems Energy Estimator. (Website). http://see.au-syd.mybluemix.net/see/EnergyEstimator

[16] Abhishek Kumar Jain, Suhaib A. Fahmy, and Douglas L. Maskell. 2015. Efficient Overlay Architecture Based on DSP Blocks. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE.

[17] David E. Keyes, Hatem Ltaief, and George Turkiyyah. 2020. Hierarchical Algorithms on Hierarchical Architectures. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 378, 2166 (Jan. 2020), 20190055.

[18] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. 2016. NVMeDirect: A User-space I/O Framework for Application-specific Optimization on NVMe SSDs. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*. USENIX Association, Denver, CO.

[19] Dirk Koch, Christian Beckhoff, and Guy G. F. Lemieux. 2013. An Efficient FPGA Overlay for Portable Custom Instruction Set Extensions. In *2013 23rd International Conference on Field programmable Logic and Applications*. IEEE.

[20] Nallatech. 2017. 250S Hardware Reference Guide. (Data Sheet).

[21] OpenPOWER Accelerator Work Group. 2017. CAPI Storage, Network, and Analytics Programming (SNAP) Framework. (Website). https://developer.ibm.com/linuxonpower/capi/snap/

[22] Max Plauth. 2018. MetalFS: Near-Storage Operators for CAPI SNAP. (Presentation).

[23] Zhenyuan Ruan, Tong He, and Jason Cong. 2019. INSIDER: Designing In-Storage Computing System for Emerging High-Performance Drive. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 379–394.

[24] Samsung. 2016. PM953 NVMe PCIe SSD. (Data Sheet).

[25] Rober Schmid, Max Plauth, Lukas Wenzel, Felix Eberhardt, and Andreas Polze. 2019. Orchestrating Near-Data FPGA Accelerators Using Unix Pipes. In *2019 Seventh International Symposium on Computing and Networking Workshops (CANDARW)*. 125–128.

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

[27] Debendra Das Sharma. 2019. Compute Express Link. (White Paper).

[28] Deshanand P. Singh, Tomasz S. Czajkowski, and Andrew Ling. 2013. Harnessing the Power of FPGAs Using Altera's OpenCL Compiler. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '13)*. ACM, New York, NY, USA, 5–6.

[29] Gagandeep Singh, Lorenzo Chelini, Stefano Corda, Ahsan Javed Awan, Sander Stuijk, Roel Jordans, Henk Corporaal, and Albert-Jan Boonstra. 2019. Near-Memory Computing: Past, Present, and Future. *Microprocessors and Microsystems* 71 (Nov. 2019), 102868.

[30] Myron Slota. 2018. OpenCAPI Technology. (Presentation).

[31] Hayden Kwok-Hay So and Robert Brodersen. 2006. Improving Usability of FPGA-Based Reconfigurable Computers Through Operating System Support. In *2006 International Conference on Field Programmable Logic and Applications*. IEEE.

[32] Hayden Kwok-Hay So and Cheng Liu. 2016. FPGA Overlays. In *FPGAs for Software Programmers*. Springer, 285–305.

[33] Athanasios Stratikopoulos, Christos Kotselidis, John Goodacre, and Mikel Luján. 2018. FastPath: Towards Wire-Speed NVMe SSDs. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. 170–1707.

[34] Jeffrey Stuechel, Bart Blaner, Charles R. Johns, and Michael S. Siegel. 2015. CAPI: A Coherent Accelerator Processor Interface. *IBM Journal of Research and Development* 59, 1 (Jan. 2015), 7:1–7:7.

[35] Jeffrey Stuechel, William J. Starke, John D. Irish, L. Baba Arimilli, Daniel M. Dreps, Bart Blaner, Curt Wollbrink, and Brian Allison. 2018. IBM POWER9 Opens Up a New Era of Acceleration Enablement: OpenCAPI. *IBM Journal of Research and Development* 62, 4/5 (July 2018), 8:1–8:8.

[36] Symas Corporation. 2019. Lightning Memory-Mapped Database. (Website). https://symas.com/lmdb/

[37] Anuj Vaishnav, Khoa Dang Pham, and Dirk Koch. 2018. A Survey on FPGA Virtualization. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE.

[38] Lukas Wenzel. 2019. *Operating System Abstractions for FPGA Accelerator Designs*. Master's thesis. Hasso Plattner Institute for Digital Engineering, University of Potsdam.

[39] Louis Woods, Jens Teubner, and Gustavo Alonso. 2013. Less Watts, More Performance: An Intelligent Storage Engine for Data Appliances. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 1073–1076.