D6.2 Prototype and Overview of Managed Storage Tiers and Near-Data Processing

DAPHNE

Integrated Data Analysis Pipelines for Large-Scale Data Management, HPC, and Machine Learning

Version 1.2

PUBLIC

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 957407.

Document Description

Report on state-of-the-art techniques for computational storage, near-data processing, and potential side effects in the context of the I/O hierarchy, as well as an overview of automatically determining the capabilities of a storage configuration.

D6.2 Prototype and Overview of Managed Storage Tiers and Near-Data Processing			
WP6 – Computational Storage			
Type of document	D	Version	1.2
Dissemination level	PU	Project month	24
Lead partner	ITU		
Author(s)	Philippe Bonnet (ITU)		
Contributors	all		

Revision History

Version	Item	Comment	Author / Reviewer
V1.0	Structure of the document		Philippe Bonnet
V1.1	First draft		Philippe Bonnet, Marcus Paradies, Niclas Hedam
V1.2	Complete version	Based on feedback from reviewers P.Damme and I.Tolovski	Philippe Bonnet, Marcus Paradies, Niclas Hedam

Executive Summary

This deliverable describes the design of the Delilah prototype for offloading eBPF code on a computational storage platform. We detail its implementation on the Daisy computational storage platform and discuss its initial performance evaluation.

1 Introduction

The current DAPHNE prototype relies on the traditional file abstraction to access stored data through DAPHNE scripts. In the context of task 6.2, we implemented I/O kernels[1] that support, read, and write operations where a data object (e.g., a frame or a matrix) is read or written in its entirety from/to file. In terms of data representation, the current DAPHNE prototype supports common data formats for frames and matrices (csv, parquet, matrix market) as well as the DAPHNE binary data format [2]. The prototype integrates existing access libraries (arrow and parquet).

This design makes it possible to manipulate stored data sets with DAPHNE scripts. However, this initial design has a few limitations:

a) DAPHNE scripts only work with data sets that can be entirely loaded in memory.
b) The DAPHNE run-time does not optimize performance or minimize data movement when accessing stored data.

Tasks T6.3, T6.4 and T6.5 are designed to address these limitatons, in the rest of the project, through careful, asynchronous data movement across storage tiers and by leveraging computational storage.

This deliverable is a report associated to a demonstrator. It focuses on the Delilah prototype that we have designed and implemented to support code offload on computational storage, a necessary first step towards integrating computational storage in DAPHNE.

Before we describe Delilah, let us first situate this work in the context of the storage tiers that are relevant for DAPHNE.

2 Storage Tiers in DAPHNE

In Deliverable 6.1, we identified three storage tiers that are typical in both cloud systems and high-performance computing (HPC) systems where DAPHNE can be deployed: performance, capacity, and archival tiers. We also identified a range of different abstractions that are used to expose storage to programmers.

Without loss of generality, we narrow the storage tiers that we consider in DAPHNE to:

1. Tier 1: High-performance NVMe Solid-State Drives (SSDs) or SSD arrays that are accessed via xNVMe [3] (see D6.1), and
2. Tier 2: Performance, capacity and archival tiers, composed of high-capacity SSDs, Hard-Disk Drives (HDDs) and tapes that are accessed via a (parallel) file system.

These two tiers are present on modern HPC systems (e.g., Frontier [4] that distinguishes between local NVMe drives, a centralized Lustre file system equipped with performance and capacity tiers and an archival system exposed though HPSS) [5] and on cloud systems (e.g.,using S3FS [6] to mount S3 buckets as a file system).

Note that some HPC systems are equipped with Tier 2 but not Tier 1 (e.g., LUMI [7] where three storage systems are available via a file abstraction using Lustre or Ceph). Those systems cannot leverage all the benefits of high-performance SSDs and they cannot leverage computational storage out of the box.

Explicit tier management and data path optimization across Tier 1, Tier 2 and memory is needed to optimize for performance (tasks T6.4 and T6.5). Near data processing focuses on computational storage devices that are part of tier 1. Indeed, computational storage is being standardized in the context of NVMe [8] (Task T6.3).

Implicit tier management takes place within Tier 2. This complexity is hidden from the DAPHNE run-time that relies on files to access data from this tier. Let us illustrate implicit tier management using the German Satellite Data Archive as an example. The main storage system of the German Satellite Data Archive is based on a hierarchical storage system solution consisting of two tiers: a capacity tier and a performance tier. The capacity tier is a robotic tape library. Each tape drive can hold between 6 and 30 TB of data (depending on the compression level applied). Sequential read/write speed per drive is between 300 MB/s and 900 MB/s (again, dependent on the drive type and compression level enabled). In total there are two geo-replicated libraries at two locations in Germany that are used to handle potential natural hazards, load balancing, and overall improvement of access latencies. Per library, there is a number of tape drives attached, for the German Satellite Data Archive this is up to 64 drives.

Files get staged (written) to a performance tier based on hard-disk drives (HDD), which also serves as a caching layer to quickly serve repeating data requests. Both tiers are connected via a storage-area network and operated via a hierarchical storage management (HSM) software, which exposes the data through a virtual file system abstraction.

In this example, the file abstraction already incorporated in the DAPHNE prototype makes it possible to write DAPHNE scripts that access the DLR satellite images with minimal modifications, assuming that the nodes where DAPHNE runs mount DLR's hierarchical storage management system via the Linux virtual file system

In summary:

- The current DAPHNE prototype supports a simple form of data movement between tier 2 and memory based on synchronous reading/writing of an entire file.
- Improvements to the prototype resulting from T6.4 and T6.5 will include (i) block-based, asynchronous access to files in tier 2, (ii) asynchronous access to blocks in tier 1, (iii) cost-based staging of blocks across tiers.

3 Near-Data Processing with the Delilah Prototype

We consider near-data processing using computational storage in the context of Tier 1. The NVMe standard for computational storage is expected for early 2023. This standard will be aligned with Storage Network Industry Association (SNIA)'s architecture and programming model for computational storage [9], published in August 2022. In particular, it will support the offload of extended Berkeley Packet Filter (eBPF) bytecode to computational storage.

We described the role of eBPF in D6.1. In order to experiment with eBPF code offload, we designed and implemented the Delilah prototype. In the rest of this section, we describe the Delilah demonstrator. Its integration with the DAPHNE run-time remains an issue for future work.

3.1 Background

The goal of the Delilah prototype is to support eBPF code offload on a computational storage device attached to a host via PCIe. The Delilah prototype implements the Eid-Hermes protocol [10], defined by Eideticom, on the Daisy platform, manufactured by a company namedCRZ.

3.1.1 Daisy Platform

The Daisy OpenSSD is a storage system built with 2x100GE and PCIe Gen3x16 connectors, a Zynq Ultrascale+ MPSoC, and a backplane interface for connecting to two M.2 SSDs. [11] Daisy is the latest iteration of the OpenSSD prototypes developed by Prof. Song and his team at theENC Lab, Hanyang University[12].

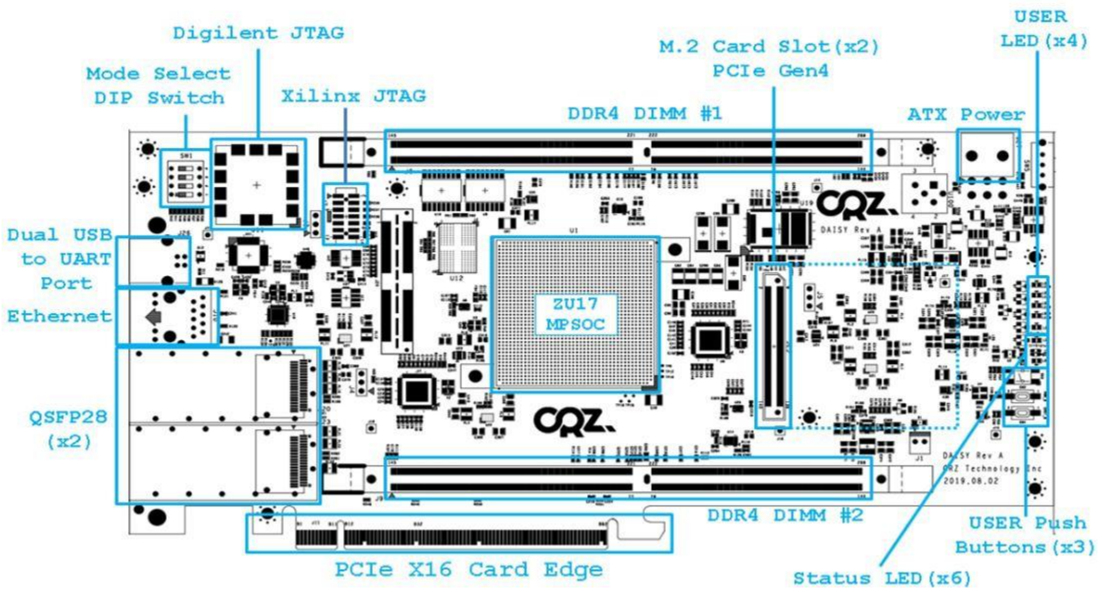

Figure 1 A diagram showing the physical components of the Daisy OpenSSD. From http://crztech.iptime.org:8080/Release/OpenSSD/Daisy-OpenSSD/Doc/DAISY_UserGuide_Rev1.1.pdf

The Daisy OpenSSD offers multiple host interfaces. For this demonstrator, we focus solely on the PCIe Gen3 X16 interface.

3.1.2 Peripheral Component Interconnect Express (PCIe)

PCIe is a transactional layered network protocol with requests and responses. [13] It is layered on top of a packet-based data link protocol and physical connections organised as a set of lanes. A lane is a pair of unidirectional, serial, point-to-point connections.

PCIe connections will have a root complex connected directly or via switches to one or more endpoints. PCIe endpoints have a vendor ID and a device ID. PCIe devices may have one or many memory-mapped memory spaces, defined via the Base Address Registers (BARs). These regions reside on the device but are accessible directly fromthe host memory.

3.1.3 Direct Memory Accesses (DMA)

It is possible, via the PCIe protocol, to move large chunks of data from the host to the device and back independently of the host CPU. These operations are managed by the DMA-controller in the PCIe Root Complex at the hardware level. On the Daisy board, DMA operations are set up and executed by XDMA. DMA is conducted via a set of write channels, called H2C, and a set of reading channels, called C2H.

It should be noted that DMA is the most performant way to transfer large amounts of data over PCIe.

3.1.4 Zynq UltraScale+ MPSoC

The Zynq Ultrascale+ is a heterogeneous multiprocessing platform with an ARM processor and an FPGA hardware accelerator (it is labelled UG1085) [14]. The ARM processor or Processing System (PS) can run an entire operating system and thus run any eBPF program on demand. The FPGA component or Programming Logic (PL) allows hardware elements to be wired efficiently for the computational storage use-case. Zynq Ultrascale+ MPSoCs come with multiple high-performance PS-PL ports to connect an operating system in PS to peripherals in PL. While the most efficient ports expose AXI interfaces, the chip can also expose general purpose ports like GPIO and USB.

Xilinx MPSoC uses isolation to separate subsystems. [15] As such, IPs and components with memory capabilities will be assigned an aperture, which is a memory region with its own permissions and address bits.

Figure 10-1 of UG1085, the reference manual for the Zynq UltraScale+, shows the various apertures. The largest aperture can contain addressing for 256 GB via 40 bits, while the smallest can only address 1 MB.

3.1.5 Daisy Programming Environment

As mentioned in the previous section, the Daisy OpenSSD comes with a Xilinx Zynq Ultrascale+ FPGA component. This means that the SSD must be programmed using the Xilinx toolchain.

The toolchain includes the Vivado suite for managing the hardware aspects and the Petalinux SDK for managing the operating system.

When programming FPGAs, one must describe the desired hardware layout. The hardware layout is visualised as Block Diagrams and stored as Hardware Design Files (HDFs). The Block Designs show the wiring of physical components on the device and any intermediate IPs between the components. IPs in the context of Xilinx are pre-developed virtual components often used as middleware between components that cannot communicate out of the box.

We will describe some of the typical IPs in this section.

3.1.5.1 Advanced eXtensible Interface (AXI)

AXI is an interface specification that defines the interface of IP blocks. [16] In essence, AXI allows various IPs to connect over a generalpurpose connection. In a Xilinx block design, it is the preferred way to connect IPs together, as it abstracts the details of the connections away.

For example, when connecting PS to any of the complex FPGA peripherals, one must use one of the PS-PL ports described in 2.1.2. However, the PS-PL ports on the Zynq are not directly compatible with the AXI port on the peripherals. To connect PS to peripherals or to connect peripherals together directly, one must use an AXI Interconnect or an AXI SmartConnect. The two connectors serve the same purpose, but the Interconnect type allows more fine-grained configurations over SmartConnects. We use Interconnects in our design to allow tweaking of the configurations when necessary.

3.1.5.2 PCIe DMA/Bridge – XDMA

Xilinx's DMA intellectual property component (XDMA IP) offers a way to expose the FPGA as either a PCIe Root Complex or PCIe Endpoint. When an XDMA IP is in bridge-mode, XDMA will function as PCIe Root Complex to an underlying PCIe Slave device. [17] Alternatively, putting XDMA in DMA-mode will expose the device as a PCIe Slave device with DMA capabilities to a host-machine. [18]

Although possible, it is not required to connect an XDMA component to the board's processing unit. For example, when in DMA-mode, the XDMA IP only has to have one or more AXI backends. It is then possible for the host to issue I/Os to any of the apertures in the AXI backends.

The IP have many different configurable properties. Most notably are the number of PCIe lanes, the link speed, and the BARs.

XDMA allows the FPGA to raise interrupts over PCIe. It supports up to 32 legacy, MSI and MSI-X interrupts. However, half of the interrupts are reserved for DMA operations. Interrupts are raised by an interrupt handshake, where a component in the block design sends a signal over a predetermined pin to notify the host. The component must keep signaling the pin until the host has acknowledged and serviced the interrupt.

3.1.5.3 Memory Interface Generator (MIG)

MIG is an IP for exposing a DIMM slot as a memory aperture via AXI.

The reason MIGs are needed to access the DIMM-memory is simple; Modern DIMM sticks have 288 pins, which have to be wired correctly to the underlying processing unit or peripheral. Xilinx provide MIGs to expose these 288 pins as a single AXI connection. Without the MIG IP, hardware developers would have to connect each of the 288 pins and implement a driver to communicate with the memory stick.

MIGs are configured by providing the attributes of the memory sticks mounted in the slot. This includes, for example, I/O latencies, voltages and information about columns, banks and ranks.

3.1.5.4 General-Purpose IO (GPIO)

The AXI GPIO IP provides a way to expose single pins to PL or outside of the FPGA. When used in conjunction with Petalinux, each pin is exposed as a pseudo-file in /sys/class/gpio. All GPIO chips connected to Petalinux will have a base, which indicates the identifier of the first pin on the GPIO chip. The base of a chip is assigned by the operating system at boot-time and can be deducted using various hints like the static memory address from PL.

Configuring the pin in Petalinux is simple. First, one must export it, which tells the Linux driver to make the pin available to userspace. Afterwards, one must set the direction to out unless the pin is considered read-only.

After setting up the pin, one can easily change the pin by writing either 0 or 1 to the value pseudo-file of the pin.

Below is an example of a successfully configured GPIO-pin, which is immediately changed to1.

```
$ cd /sys/class/gpio

$ echo 504 > export

$ echo out > gpio504/direction

$ echo 1 > gpio504/value
```

3.1.5.5 Petalinux

Petalinux is an SDK for generating and compiling a lightweight Linux-based operating system for running on the Zynq MPSoC. The SDK takes a Hardware Description File (HDF) as input and generates an operating system with the necessary system files, including the device tree and drivers. As Linux powers Petalinux, it is possible to enable many of the drivers and modules of a typical Linux system, including support for various drive types (fx. NVMe, Open-Channel SSDs) and software packages (fx. GCC, GDB, Python). Furthermore, developers provide their packages and modules, which are cross-compiled to run on the MPSoC.

The Petalinux SDK provides a tool to generate Hello World programs and modules for Petalinux with build configurations for languages like C and C++. This provides an excellent starting point for developing and integrating software into a Petalinux instance.

Petalinux compiles applications and libraries on the development machine, which does not necessarily run the same architecture as the FPGA. As such, Petalinux makes use of cross-compilation tools. Under the hood, Petalinux compiles software using Yocto, a collaborative open-source project. The goal of Yocto is to develop tools and processes that enable the generation and compilation of Linux distributions for embedded and IoT systems. [19] The tools and processes are independent of the underlying architecture of the embedded hardware.

The way cross-platform compilation practically looks in Petalinux is simple. A software package provides a simple Makefile, which gives no configurations on architecture. Developers can compile and test code on the development machine without regard to the FPGA. When the code is to be compiled through Yocto for the FPGA, Yocto appends a set of architectural configurations to the compiler, including the CROSS TARGET setting. As such, the cross-compilation is entirely abstracted away from the software developer. It should be noted that this project is based on Petalinux version 2019.1.

3.1.5.6 Vivado

Vivado is a software suite for synthesising and analysing HDLs with additional features for system on chip development and high-level synthesis.

Vivado helps automate the design phase of FPGAs by automatically connecting IPs that have similar interfaces. For example, if a Zynq IP is added and a MIG is added, Vivado will suggest connecting these two and generate the intermediate AXI Interconnect IPs. Due to this, the main task of an FPGA designer using Vivado is understanding which IPs are appropriate for the application and defining applicable constraints.

Constraints in the context of Vivado are a set of rules for how the block design is wired to external hardware. As this depends on the FPGA and its capabilities, it cannot be automated. In some limited circumstances, Vivado can infer constraints, for example, if related constraints are defined. Many boards come with a guide or specification on which port various hardware components are wired.

It should be noted that for the during of this project, we will work with Vivado version 2019.1.

3.1.6 User-Space eBPF (uBPF)

eBPF bytecode is either interpreted through a virtual machine or translated into assembly via a just-in-time compiler. The Linux kernel contains both. In the context of computational storage, we are interested in executing eBPF code on the computational storage platform, which is generated on the host. We are thus interested in an interpreter or just-in-time compiler in user-space. This is why we are considering uBPF. The uBPF VM is a RISC register machine based on eleven 64-bit registers, one stack pointer, an implicit program counter and a fixed-size stack [20] [21]. The virtual machine has access to several registered functions, often used for BPF helpers. However, in principle uBPF allows registering any function to the VM.

The uBPF VM accepts either a buffer with eBPF instructions or an eBPF ELF file. The VM loader can parse the segment header table and sections to extract the program and references to registered functions. On a computational storage device, we aim to reuse the standard uBPF design with only minor modifications. The only required modification is to allow the call operation to invoke any function and not just BPF helpers.

The uBPF VM exposes a simple interface with three operations. First, a load function that takes a program as a parameter and prepares the uBPF engine for execution. Second, an unload function that resets the state of an uBPF engine. Lastly, an execute function executes the loaded program with respect to a provided memory buffer.

When an eBPF program is executed within the uBPF virtual machine, a return value can be returned from eBPF. It corresponds to a value stored in register 0 of the VM. Put differently, an eBPF program does not return blocks of data. It may return a form of pointer to the memory managed by the uBPF virtual machine or an identifier for a resource managed outside the uBPF virtual machine. A protocol would be needed to make it possible for an external program to get access to memory managed by the uBPF virtual machine. More generally, access to resources associated to the Computational Storage Processor, such as memory, direct-attached SSDs or other peripherals, must be mediated through external functions. If those resources are shared across several virtual machines, then the registered functions should provide concurrency guarantees.

In the context of computational storage, eBPF bytecode is shipped from the host to the computational storage device that should load the bytecode onto the virtual machine (or JIT-compile the bytecode) and execute it. There is a need for a protocol that exposes the uBPF virtual machine functionalities to the host. Eid-Hermes defines such a protocol.

3.1.7 Eid-Hermes

Eid-Hermes is a host-controller transport protocol, which supports the offload of eBPF code generated on a host and executed on a device. Eid-Hermes is an Eideticom-led project. [22]

Eid-Hermes basically exposes the load/unload/execute interface of uBPF to the host. One of the key aspects of Eid-Hermes is that it focuses on the transport of data as well as programs. Programs are loaded onto a uBPF virtual machine. Data is passed as an argument when the program is executed. More specifically, Eid-Hermes makes use of program slots and data slots as abstractions of memory buffers. The number of slots and their respective sizes are exposed to the host during enumeration. Neither program nor data slots exist in the context of uBPF and as such, it is the responsibility of the device to pass along appropriate pointers to uBPF. The pointer of program slots are given to uBPF when loading a program [23], while data slots are passed as argument when executing a program [24].

In the context of computational storage, Eideticom's choice of adding data slots to the protocol opens a range of possibilities. First, it allows data to be transferred to the device alongside programs. Second, it is possible to daisy-chain programs to use the same data without transferring the data to the device and back.

Eid-Hermes makes use of three Base Address Registers BARs. First, BAR0 holds the command registers and the state of Eid-Hermes, including the capabilities of the device. BAR2 holds configurations related to XDMA. BAR4 holds the program and data slots of uBPF. It should be noted that the program and data slots are populated using DMAs. As such, an implementing device does not need to expose the slots as a BAR since the host CPU never issues reads or writes directly to the memory regions.

The protocol uses XDMA, a Xilinx DMA engine that was described in an earlier section, to transfer programs and data to the device. Commands are transferred and executed by writing to the Eid-Hermes command register on BAR0.

3.1.7.1 Commands

The lifecycle of a command looks like this:

1. The user sends a command request to the Eid-Hermes driver.
2. The driver copies the command request to the BAR0 command registers.
3. The driver writes the engine start bit of BAR0 to start command execution.
4. Upon completion, the device writes the command response (16 bytes) to the BAR0 command registers and raises an MSI-X interrupt

The driver then reads the command response and returns it to the user.

In the current release of the Eid-Hermes protocol, only three commands exist; Request Slot (0x0), Execute Program (0x80) and Release Slot (0x01). The current version of the driver does not implement 0x0 and 0x1 as slot management is handled on the host.

3.1.7.2 Theory of Operation

The high-level overview of the Eid-Hermes protocol can be boiled down to four specific steps.

1. The host discovers the capabilities of the device. The capabilities include the number of slots and engines, including their physical location on the device memory.
2. The host initiates DMA from the host to the appropriate program and data slots of the device. At the end of this step, a program slot will hold the offloaded program, and a data slot will hold the initial data of the program.
3. The host initiates the execution of a program by setting the engine start bit to 1. The engine start bit is set to 0, and the engine finished bit to 1 upon completion.
4. The host initiates a transfer of the resulting data back to the host.

3.1.7.3 Linux Quick EMUlator (QEMU)

Eideticom has developed a host driver for Eid-Hermes, based on the DMA library from Xilinx (XDMA). Eideticom has also developed a QEMU version of the Eid-Hermes protocol. The QEMU version simulates how physical Eid-Hermes devices work and interact and executes programs in a uBPF virtual machine.

The Eid-Hermes QEMU implementation is designed to be self-contained. This means that although Eid-Hermes is based on XDMA, the QEMU version does not include or reference XDMA. Instead, the Eideticom implements only the capabilities of XDMA used in Eid-Hermes. A problematic aspect of this is that Eideticom cannot give any guarantees that their emulated version of XDMA behaves the same way as a physical XDMA device.

For example, the QEMU version contains at least one instance where the QEMU device acts differently from a physical device. The Eid-Hermes driver does not set the user interrupt mask correctly, which the QEMU completely disregards. This can present issues if the software is tested on QEMU since the software may behave differently on a physical device.

The project is multi-threaded, and there exist multiple threads per DMA channel and one thread per uBPF engine. Therefore, the implementation relies on mutual exclusion locks and semaphores to function.

3.2 Delilah Block Design

Our Delilah prototype is software run on the ARM core of the Zynq MPSoC (introduced in Section 3.1.4). To run on the Daisy board, our prototype requires that the ARM cores (PS domain) are connected to the host via PCIe and that they are connected to M.2 SSDs. This requires that the FPGA (PL domain) is programmed to provide this connectivity. We had to build this basic functionality as it did not exist on the Daisy board.

To this end, we selected and configured existing Xilinx IPs, defined for managing PCIe DMAs, memory management and data transfers within the Zynq Ultrascale+.

Our starting point in the exploration of a functional block design is the example designs built by the manufacturer of the Daisy. In particular, the Daisy M.2 PCIe + MIG design, [25] which provides a layout including most of the necessary components. In their design, PL consist of XDMA and MIGs. However, they do not make use of the PS domain.

With the manufacturer's example design as point of departure, we explore what is given and what must be added to the design. The design contains three XDMA IPs; Two in bridge mode for connecting to the M.2 slots on the Daisy, and one in DMA mode for connecting to the host over PCIe. The XDMA IPs are connected to external clocking pins, ensuring that communication between the Zynq and SSD or host, respectively, does not result in timing issues.

The two XDMA IPs in bridge-mode are configured with factory-tested settings provided by the manufacturer. As such, we will not explain in detail the configurations. Most importantly, they are configured to use PCIe 3.0 with 4 PCIe lanes. They are connected to the M.2 backplane of the Daisy and managed by the NVMe driver residing in PS.

The XDMA IP in DMA-mode has BRAM as backend without any connection to the Zynq. This means that the PS domain have no way of accessing anything transferred to the device. It also has no BAR0 defined. As such, we must expand the block design here.

We can either connect the Zynq to the BRAM or alternatively, change the backend to be the Zynq. The first option presents limitations on capacity, while the second requires us to reserve a memory space within PS, which limits the memory available for the operating system and other processes. In our design, we choose to go with the latter approach, as the Linux kernel does not require much memory and since Delilah will be the only major process running.

First, we configure the XDMA IP with an AXI connection to the Zynq, such that the host can issue DMA transfers to the Zynq's internal memory. Second, we expose another memory segment within the Zynq via AXI-Lite. This memory segment holds the 0th BAR, which contains the Eid-Hermes-related configurations and command registers. We choose to accept the configured values from the manufacturer which is set to maximise the full potential of the PCIe connection.

Furthermore, the given block design does not contain functionality to raise interrupts on the host. We add an AXI GPIO IP, which we can manage from PS to communicate with the interrupt pins of the XDMA IP.

The resulting block design can be seen in Figure 3 and Figure 4. Beside the XDMA IPs and our proposed additions, the block design contains a set of intermediate and support IPs defined and configured by the manufacturer. For example, two GPIO IPs provide the M.2 SSDs with 3.3V power and a set of Utility Vector Logic IPs sets the PERST pins on the M.2 as per the PCIe specification. Processor System Reset IPs ensures that asynchronous external reset input is synchronised with the internal clocks in the block design. Lastly, Utility Buffers, Concats and Virtual I/Os help alter and modify signals according to the underlying specifications. Most of the support and intermediate IPs have been configured and tested by the manufacturer as an example of how to configure the Daisy.

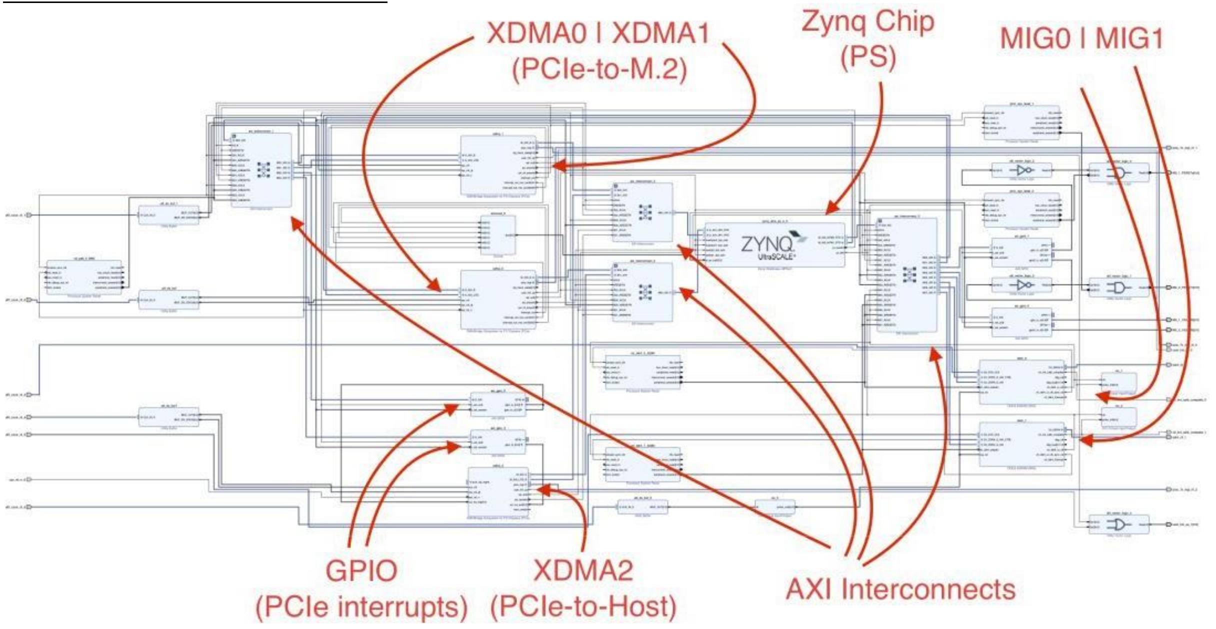

Figure 2 A full view of the Block Design of Delilah. As the diagram is very small, captions have been added to show important IPs.

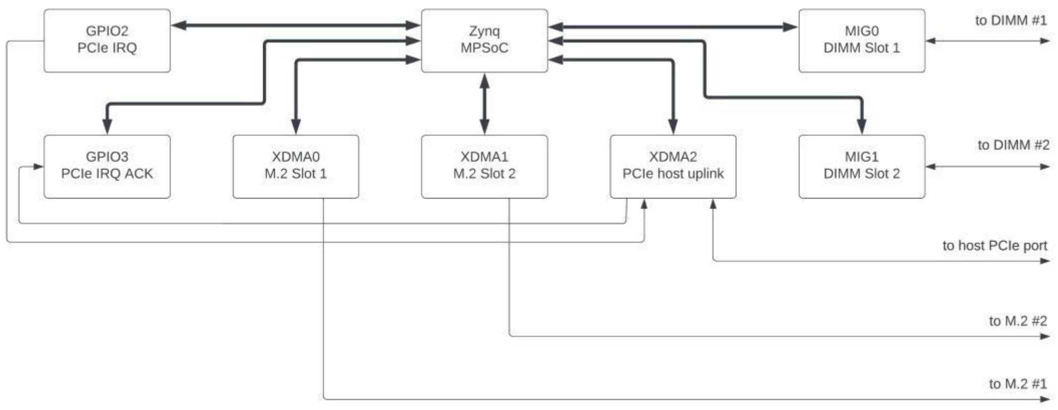

Figure 3 An abstract overview of the Delilah Block Design showing the most important IPs. The bold connections are AXI connections, while the thin connections are one or more singular wires. Intermediate and support IPs have been abstracted away for simplicity.

Lastly, we have two MIGs that enable us to use Daisy's two DIMM slots equipped with two 32 GB DDR sticks. They are, however, not being used in the proof-of-concept release, as Eid-

Hermes only supports DMA to 32-bit addresses. DIMM slots connected to Zynq UltraScale+ chips are assigned memory addresses in the 40-bit apertures and thus cannot be referenced by Eid-Hermes.

We have upstreamed an Eid-Hermes contribution to enable referencing of 64-bit addresses. Since this was a breaking change, it required an increase of the Eid-Hermes version counter (EHVER).

The block design proposed in this section is generic and will be useful for other Daisy users. We are making it available at https://github.com/delilah-csp/delilah-bd.

3.3 Delilah Software

Our design relies on several layers of software components.

First, we have Petalinux configured with mostly default settings. The most important changed settings are the boot configurations to match the specifications of the Daisy. We have provided the block design from the previous section to the Petalinux generator, ensuring that all applicable drivers and device-tree information are present.

Second, we have the user-space device-side driver containing the Delilah. Since uBPF, as of 2022, is released under an Apache 2.0 license, we choose to integrate the uBPF VM directly into the Delilah. This is to avoid compiling and integrating certain components which are not ready for the ARM architecture, such as the JIT compiler.

Delilah performs several key operations in sequence during startup, which we describe in the list below.

1. Open the memory region containing program and data slots using /de-v/mem and mmap.
2. Open the memory region containing BAR0 using /dev/mem and mmap.
3. Open and configure the GPIO pins for raising interrupts, including exporting the pins to userspace and setting the direction to out.
4. Expose Hermes configurations on BAR0, including the physical addresses of the program and data slots and their respective sizes.
5. Create four uBPF engines and register external functions, if applicable.
6. Spawn four threads to manage command execution of each engine.

After the six steps are completed, the startup finishes and the Delilah runs in each of the engine threads. The sequence follows the flow described below.

1. If a command has been received, continue, else wait a single microsecond and try again.
   - Find the appropriate command handler from the opcode.
     i. If 0x00, allocate a free slot from the pool of slots.
     ii. If 0x01, put the given slot back to the pool of slots.
     iii. If 0x80, follow the instructions below.
        1. Load the program slot into the engine.
        2. Execute the program with respect to the given data slot.

3. Unload the program.
4. Move return values or error codes to the response registers.

- Set the engine completion bit.
- Raise the interrupt of the respective engine.

3.3.1 Design

In summary, we introduce Delilah: a fully functional computational storage platform based on Eid-Hermes on the Daisy OpenSSD. To the best of our knowledge, this is the first such device. While the Eid-Hermes framework is specified and a host driver is implemented, no device actually exists to implement the protocol.

On the host side, clang for GCC is used to compile C programs into eBPF bytecode. This operation is similar to compiling eBPF programs for the kernel with two major differences. First, programs do not go through the meticulous verification process, as they are never loaded into the kernel. Second, programs are given two parameters in the main function; A pointer to a contiguous memory buffer holding state and data and an unsigned integer holding the size of the buffer.

The Daisy is mounted in a PCIe slot on the host, where the host serves as PCIe root complex and the Daisy as the slave. The host loads the Eid-Hermes driver and enumerates the Daisy, which is then exposed as a device on /dev/hermes0. The driver may now load the precompiled program and ship it to the Daisy using XDMA.

XDMA is used for transferring programs and data to predefined slots on the Daisy. Commands follow the Eid-Hermes specification, which requires commands to be written to BAR0. BAR0 serves as the main point of entry for Eid-Hermes. BAR2 holds the configurations for PCIe and DMA, which is managed entirely by XDMA.

On the device side, we have three distinct software packages. First, XDMA handles communication with the host and provides DMA capabilities. Second, we have one or more uBPF virtual machines in charge of executing BPF code. Lastly, we have Delilah.

The Delilah is responsible for setting up the uBPF virtual machines, spawning worker threads and exposing Eid-Hermes-related configurations to the host. Delilah also is responsible for receiving and executing commands and programs. It is furthermore responsible for notifying the host when programs are finished executing.

We implement an extension to uBPF called registered functions, which are a set of functions made available to eBPF programs for managing Delilah resources, including the underlying SSDs. We expect to either use a file system or make use of xNVMe [26] to access SSDs, e.g., NVMe ZNS SSDs, as raw devices. Defining registered functions for a given system is a topic for future work.

3.3.2 Implementation

In this section, we will describe the implementation details of the project.

3.3.2.1 Build Process

Compiling from scratch is a long-running process that usually takes several hours.

However, after the first compilation, some artefacts may be cached for later use.

The first step of compiling the block design is initiating the Out-of-Context synthesis. The OoC step will generate and compile functional blocks for each IP to be reused later. It is comparable to compiling individual files without linking them. The goal is to avoid recomputing the whole IP after minor changes.

The next two steps are synthesis and implementation. Here the IPs are first converted to schematics, which are then later converted into a bitstream. During implementation, the constraints come into play since the compiler now ensures that all wires are connected according to the constraints. Implementation is the most time-consuming phase since it cannot be cached. This is the case since a single change of wire may cause the whole layout to be regenerated.

When the implementation phase has finished, and the resulting bitstream is exported, all hardware components are compiled. This is a milestone in the build flow, as the hardware will not have to be recompiled unless the block design or constraints change. This means that recompiling changes in software does not require a hardware recompile, and thus the time required to compile is decreased tenfold.

All the steps above are executed in Vivado. The coming steps are conducted in the Petalinux SDK, which is command-line only.

First, we provide the resulting bitstream to Petalinux using the configuration tool. This updates the device tree and boot configurations to contain the newest hardware. During the implementation of Delilah, we have previously observed caching problems where previous hardware was deployed after updating the bit-stream. We manually invoke the mrproper build step, which cleans the cache of Petalinux. Compilation without caching takes roughly 15 minutes.

```
$ petalinux-config --get-hw-description ../delilah-hw/ps.sdk/

$ petalinux-build -x mrproper
```

The manufacturer of the Daisy has provided a convenience script for compiling Petalinux. This script manages the artefacts that Petalinux yields, which may be several files. In essence, the convenience script can be boiled down to the following:

```
FSBL=zynqmp_fsbl.elf

PMUFW=pmufw.elf

FPGA=system.bit

ATF=bl31.elf

UBOOT=u-boot.elf

$ petalinux-build

$ petalinux-package --boot --fsbl ${FSBL} --pmufw ${PMUFW} --fpga ${FPGA} --atf ${ATF} -u-boot ${UBOOT} --force
```

The convenience script furthermore moves all artefacts to known locations to ease the deployment. Another convenience script helps format and create an SD card with the artefacts. In essence, it creates a boot partition and an operating system partition.

3.3.2.2 Poll mode

After Delilah has pushed Eid-Hermes configurations to BAR0, it will spawn four threads to listen for commands in each of the four command registers. There exists no interrupt system from host to Daisy, meaning that Delilah works in poll mode.

When idle, Delilah polls the command register once a micro-second. We choose this constantto ensure responsiveness while not wasting all system resources on polling.

3.3.2.3 Interrupts

XDMA and Eid-Hermes make use of interrupts to notify the host when an op-eration has terminated. XDMA supports interrupts of the legacy type, MSI and MSI-X, while Eid-Hermes

only supports MSI-X. Furthermore, XDMA can be set to work in poll mode, thus eliminating the need for interrupts. The poll mode may be faster for small DMAs as the context switches related to interrupts are very costly.

After the Eid-Hermes driver has finished initialising, there exist two interrupts per DMA channel and one per engine. XDMA reserves the right to allocate 16 vectors and guarantees up to 16 v̄ectōrs for user-defined interruptible events. XDMA interrupts are handled by XDMA internally, while Delilah handles interrupts for Eid-Hermes. Eid-Hermes expects an interrupt after command or program execution.

Interrupts are triggered, as described earlier, by setting the respective bit of usr irq req to 1.We keep the bit one until the engine is triggered again. This ensures that the bit is one at least until the interrupt has been serviced and cleared by the host, as the Xilinx documentation requires.

Changes to the interrupt pin invoke a forced sleeping period of 1 microsecond to ensure the XDMA subsystem observes the change on the pin in case the program terminates immediately. It is possible to optimise the forced sleep period away and combine it with the sleep/poll mechanism, which is considered to be future work for now.

While implementing the interrupt mechanism above, we discovered an error in Eid-Hermes. XDMA uses IRQ masking to enable the host to silence irrelevant interrupts. By default, all interrupts are silenced, meaning that Eid-Hermes must inform XDMA of its ability to handle all applicable interrupts.

Eid-Hermes delegates the responsibility of the DMA interrupts to the XDMA driver, which correctly notifies the subsystem. However, Eid-Hermes never notifies the subsystem of its ability to handle the user interrupts, causing them to never be received and serviced on the host.

We manually implemented a temporary patch in Eid-Hermes, notifying the subsystem correctly, which resolved the bug. We expect our patch to be submitted to Eideticom when it has been tested and verified thoroughly.

3.3.2.4 Limitations on registered functions

The current implementation of registered functions in Delilah has several major limitations. Programs to be offloaded are compiled on the host and may not have access to Delilah at compile time. As such, there is no way for the compiler to know if an invoked function will exist on the device at runtime. If we assume an asynchronous mechanism for sharing device capabilities (for example, by installing a header file into Linux), we still run into problems if the device is updated and changes its registered functions. Furthermore, it is currently unknown how to handle programs that were successfully offloaded but invoke an unknown registered function, as eBPF does not support exceptions. Lastly, it is an open question whether or not registered functions must be idempotent and without side effects. If side effects are introduced, state management becomes yet another challenge to tackle.

We can summarise the challenges above into four distinct questions:

- How does the compiler know, at compile-time, which registered functions are available on a device even if the device is yet to be attached?
- How does the compiler handle versioning of registered functions, i.e. if a function signature changes?
- How can Delilah propagate back to the host that an invalid registered function was invoked?
- How can Delilah manage the state of registered functions, and how should state scope be managed (i.e. state alive for the duration of the program and state alive for the duration of the device)?

These questions are very similar to all SSDs or, more generally, PCIe device capabilities. NVMe provides admin functionality to determine capabilities. We expect that some of the questions can be answered using a similar mechanism.

3.4 Preliminary Evaluation

To determine the latency of our computational storage platform in various scenarios, we enable two simple registered functions in Delilah. One function, let us call it A, reads the entirety of a file to a predetermined buffer. Another function, B, acts as a mapping table from logical filenames to absolute paths on the filesystem.

We develop two programs to make use of the functions above. Program 1 makes use of function A to read a file. Program 2 makes use of B to translate a logical filename to an absolute path and then function A to read it.

All programs are seeded with two parameters; A file name, which can either be logical or absolute and a file size. The buffer to be returned to the host is equal to the size parameter. We attach an SSD on one of Daisy's M.2 slots, format it and create an EXT filesystem. On this filesystem, we create six files with random ASCII characters; 1kb, 10kb, 100kb, 1mb, 10mb and 100mb.

File Size (B)	Latency Arguments Write (seconds)	**Latency Program Execution (seconds)**	Latency Result Read (seconds)
1.000	0.000120	**0.023129**	0.000251
10.000	0.000157	**0.016728**	0.000229
100.000	0.000145	**0.017051**	0.000822
1.000.000	0.000287	**0.019210**	0.003907
10.000.000	0.000078	**0.038675**	0.033861
100.000.000	0.000087	**0.232203**	0.401262

Above is an overview of the results for the first program. We see that very little time is spent transferring state of a program to the device. We can also observe that the execution time of function A is constant for the first 1 MB. This means that the first ~20ms of execution time is overhead and is spent on preparing the VM, executing the program and triggering the host-side IRQ. We also see that the execution time increases less than the number of bytes.

File Size (B)	Latency Arguments Write (seconds)	**Latency Program Execution (seconds)**	Latency Result Read (seconds)
1.000	0.000068	**0.016452**	0.000124
10.000	0.000122	**0.016456**	0.000150
100.000	0.000060	**0.016551**	0.000294
1.000.000	0.000057	**0.017696**	0.004221
10.000.000	0.000107	**0.029570**	0.040555
100.000.000	0.000077	**0.142703**	0.339767

Above is an overview of the results of the second program, which adds a mapping element to the program. We see similar numbers indicating that mapping between logical file paths and absolute file paths causes no significant overhead. Furthermore, we see a lot of latency variance.

4 Delilah Demonstrator

This Delilah demonstrator contains the following components:

- BlockDesign: The FPGA block design used to deploy Delilah on the Daisy platform
- Petalinux: The Petalinux image used to deploy Delilah on Daisy's ARM processor
- Eid-Hermes: The Linux-based host-side driver used to interact with Delilah
- Evaluation: A small evaluation script used to evaluate simple eBPF function execution.

The Delilah demonstrator is packaged as a 10GB compressed file. This file is available on request.

4.1 System requirements

Delilah is built and targeted at the Daisy OpenSSD platfrm. The host we used for the demonstrator ran Linux v5.9.

The block design is built using Vivado 2019.1. Petalinux image is built using Petalinux 2019.1

4.2 Installation

The Petalinux folder is configured with the correct block design and precompiled with Delilah.To deploy to an SD card, run the following commands:

```
# cd Petalinux/project-spec/image#

./create-sdcard.sh
```

Then follow the instructions to write to your SD card. The SD card should have at least 8 GBof space.

Plug your Daisy OpenSSD into your host and power it using the external power adapter with the SD card plugged into it. Mind that the Daisy OpenSSD must be in SD mode. You can set the Daisy OpenSSd in SD mode by setting the DIP switch to ON-OFF-OFF-OFF. Attach one or more NVMe M.2 SSDs. We use Samsung EVO for the demonstration. Connect your Daisy OpenSSD through the USB/JTAG port to a third machine. It cannot be the same machine as the Daisy is connected to via PCIe. On this machine, connect to it via Minicom.

```
# sudo minicom -D /dev/ttyUSBxx
```

Turn the Daisy on using the switch. You should now see Petalinux boot in Minicom. When reaching a login prompt, log in using root and root.

Mount the NVMe

device.# mkdir

/media/nvme

```
# mount /dev/nvme0n1p1 /media/nvme/
```

The NVMe device should have the files from the Evaluation folder on it. To verify, run the ls command. It should look like this if done right:

```
root@daisy:~# ls -la /media/nvme/

total 108672

drwxr-xr-x 3 root root      4096 Sep  9 12:07 .

drwxr-xr-x 3 root root      4096 Oct 12 15:49 ..

-rw-r--r-- 1 root root      1000 Jun 15 12:38 100.txt

-rw-r--r-- 1 root root     10000 Jun 15 12:38 1000.txt

-rw-r--r-- 1 root root    100000 Jun 15 12:39 10000.txt

-rw-r--r-- 1 root root  1000000 Jun 15 12:39 100000.txt

-rw-r--r-- 1 root root  10000000 Jun 15 12:39 1000000.txt
```

```
    -rw-r--r-- 1 root root 100000000 Jun 15 12:40 10000000.txt
    drwx------ 2 root root     16384  May 18 12:34
```

lost+foundTo start Delilah, run Delilah through Minicom.

```
# delilah
```

Delilah is now running on the Daisy OpenSSD. Turn on the host that the Daisy OpenSSD is connected to. Go to the Eid-Hermes folder.

```
# cd src/driver
# make
# sudo make install
```

You can verify if the driver is loaded correctly by checking if /dev/hermes0 has been created.

Compile the evaluation suite by going to Evaluation on the host.

```
# cmake .
# make
# ./delilah-eval
```

You should now see the reading performance from 1 KB to 100 MB. The evaluation is done both using an absolute path and a logical path.

Our results show that we can read 100 MB of data to a local buffer on the Daisy OpenSSD in 0.142703 seconds and transfer it to the host in 0.339767 seconds.

As a sanity check, we see that reading 100 MB to /dev/null takes 0.069s seconds outside of Delilah. The increased latency of Delilah comes from orchestrating the execution of the read operation.

```
root@daisy:~# time cat /media/nvme/10000000.txt > /dev/null
real    0m0.069s
user    0m0.001s
sys     0m0.068s
```

5 Conclusion and Future Work

This report describes the Delilah demonstrator, which supports eBPF offload on the Daisy computational storage platform. The demonstrator is functional and its performance shows no obvious shortcomings.

Future work includes (i) a thorough performance evaluation of the Delilah prototype and most importantly (ii) its integration with the DAPHNE run-time, which requires the implementation of new IO kernels and the definition of use cases that will make it possible to explore the overall performance impact of eBPF functions offload on DAPHNE performance. In the Introduction, we identified two limitations of DAPHNE's current IO kernels: (a) datasets must fit in memory and (b) there are no optimizations of data movement. New IO kernels that leverage computational storage will address these issues by supporting access to portions of a dataset while offloading portions of processing from the CPU in a way that requires significantly less data to be transferred or leads to better performance on the host. An example of processing that can be offloaded to improve performance on the host and reduce transfers is the formatting from on-disk format to in-memory format. Another example, related to WP7, is the quantization in the DLR use case. A third example, could be a second order function such as a parameter server for mini-batch training deployed on a storage hub, associated with several computational storage devices that compute gradient descent upgrades over stored data.

6 References

[1] https://github.com/DAPHNE-eu/DAPHNE/tree/main/src/runtime/local/io

[2] https://github.com/DAPHNE-eu/DAPHNE/blob/main/doc/BinaryFormat.md

[3] https://xnvme.io/

[4] https://www.olcf.ornl.gov/frontier/

[5] https://www.hpss-collaboration.org/interfaces.shtml

[6] https://github.com/s3fs-fuse/s3fs-fuse

[7] https://www.lumi-supercomputer.eu/lumis-full-system-architecture-revealed/

[8] https://sniapmcs.org/conference/presentations/sessions/nvme-computational-storage-update-standard

[9] https://www.snia.org/standards/technology-standards-software/standards-portfolio/computational-storage-architecture-and

[10] https://github.com/Eideticom/eid-hermes

[11] http://crztech.iptime.org:8080/Release/OpenSSD/Daisy-OpenSSD/Doc/DAISY_UserGuide_Rev1.1.pdf

[12] https://site.enc.hanyang.ac.kr/home/aboutenc

[13] https://DAPHNE-eu.eu/wp-content/uploads/2021/11/DAPHNE_D6.1_Design-Space-IO-Hierarchy-1.pdf

[14] https://docs.xilinx.com/r/en-US/ug1085-zynq-ultrascale-trm/Zynq-UltraScale-Device-Technical-Reference-Manual

[15] https://docs.xilinx.com/v/u/en-US/xapp1320-isolation-methods

[16] https://developer.arm.com/documentation/102202/0300/AXI-protocol-overview

[17] https://docs.xilinx.com/v/u/en-US/pg194-axi-bridge-pcie-gen3

[18] https://docs.xilinx.com/r/en-US/pg195-pcie-dma/Introduction

[19] https://www.yoctoproject.org

[20] https://DAPHNE-eu.eu/wp-content/uploads/2021/11/DAPHNE_D6.1_Design-Space-IO-Hierarchy-1.pdf

[21] https://github.com/iovisor/ubpf

[22] https://github.com/Eideticom/eid-hermes

[23] int ubpf load(struct ubpf vm *vm, const void *code, uint32 t code len, char **errmsg);
[24] int ubpf exec(const struct ubpf vm *vm, void *mem, size t mem len, uint64 t* bpf return value);
[25] https://github.com/CRZ-Technology/OpenSSD-OpenChannelSSD/tree/main/Daisy/M.2_MIG_PCIe/Daisy_M.2_PCIe_MIG_201901_20210413

[26] https://github.com/OpenMPDK/xNVMe