Delilah: eBPF-offload on Computational Storage

Niclas Hedam
Morten Tychsen Clausen
Philippe Bonnet
IT University of Copenhagen
Denmark

Sangin Lee
University of Fribourg
Switzerland

Ken Friis Larsen
Department of Computer Science,
University of Copenhagen
Denmark

ABSTRACT

The idea of pushing computation to storage devices has been explored for decades, without widespread adoption so far. The definition of Computational Programs namespaces in NVMe (TP 4091) might be a breakthrough. The proposal defines device-specific programs, that are installed statically, and downloadable programs, offloaded from a host at run-time using eBPF. In this paper, we present the design and implementation of `Delilah`, the first public description of an actual computational storage device supporting eBPF-based code offload. We conduct experiments to evaluate the overhead of eBPF function execution in Delilah, and to explore design options. This study constitutes a baseline for future work.

ACM Reference Format:
Niclas Hedam, Morten Tychsen Clausen, Philippe Bonnet, Sangin Lee, and Ken Friis Larsen. 2023. Delilah: eBPF-offload on Computational Storage. In *19th International Workshop on Data Management on New Hardware (DaMoN '23), June 18–23, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3592980.3595319

1 INTRODUCTION

Computational storage [3] is the current incarnation of decades old ideas about active storage [19] and near-data processing [2]. According to SNIA[1], computational storage denotes architectures that provide computation *coupled to* storage, to offload host processing or reduce data movement [5]. The upcoming NVMe[2] standard for computational storage proposes extended Berkeley Packet Filter (eBPF) as a means to define and execute storage functions offloaded at run-time. Is eBPF well-suited for computational storage? How can it be used efficiently for data management? These questions should be tackled experimentally. In this paper, we present the design and implementation of `Delilah` , the first publicly described system supporting eBPF code offload on a real computational storage platform.

The BSD Packet Filter (BPF) was originally designed to evaluate boolean valued functions without copying packets inside a UNIX kernel [17]. Its modern incarnation, eBPF [20], has evolved into a vendor-neutral Instruction Set Architecture. It has become a means to execute user-defined functions inside the Linux kernel for performance monitoring [12], security monitoring [10] and more recently storage management [22]. eBPF is also used as an intermediate representation in the context of hardware accelerated packet filtering [6].

NVMe proposes to use eBPF for computational storage [4]. The upcoming TP4091 introduces a new I/O command set for computational programs. Computational programs are either downloadable functions, offloaded from the host at run-time, or device-defined functions, e.g., operating system images or FPGA bitstreams, that are installed before a device is deployed. eBPF is considered an example environment for the definition and execution of downloadable functions.

A downloadable eBPF functions may orchestrate calls to different device-specific functions and also perform data massaging. But, how can a host pass parameter values to a device-specific function using an eBPF program? How can a device-specific function return data to the host? Can downloadable and device-specific functions be executed efficiently? These are the questions we tackle in this paper.

To explore the potential and limitations of downloadable functions and computatinal storage in the context of data management systems, we built `Delilah`, a system supporting eBPF code offload on an actual computational storage device. In this paper, our contributions are the following:

- We survey the eBPF ecosystem;
- We describe the design and implementation of `Delilah` [3];
- We use `Delilah` to evaluate the overhead of executing eBPF functions and to explore design options.

2 BACKGROUND

2.1 eBPF Ecosystem

BPF stands for BSD (or Berkeley) Packet Filter. It was originally proposed in 1992 to perform user-defined packet filtering within a Unix kernel [17]. BPF defined a bytecode structure together with a virtual machine. In 2014, Alexei Starovoitov introduced extended BPF (eBPF), an adaptation of BPF for Linux and modern processors[4]. eBPF's capability to execute user-defined functions outside the boundaries of user-space has proven useful for a range of applications. This has led to the development of a rich ecosystem of tools and libraries. The eBPF foundation was established in August 2022[5] to organize the governance of this ecosystem. In this section, we only cover the aspects of eBPF that are relevant in the context of computational storage.

[1]Storage Network Industry Association, the trade association representing storage companies.
[2]Non-Volatile Memory Express, the consortium defining the industry-standard host-SSD interface specification.

[3]Code and experiments are available online at https://github.com/delilah-csp [13]
[4]https://lwn.net/Articles/598545/
[5]The eBPF foundation (http://ebpf.foundation) is a Linux Foundation Project.

DaMoN '23, June 18–23, 2023, Seattle, WA, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0191-7/23/06.
https://doi.org/10.1145/3592980.3595319

2.1.1 eBPF Programs. eBPF can be seen as a vendor-neutral Instruction Set Architecture[6]. An eBPF program is composed of a single main routine, located in memory in a *program slot* that can be read from the eBPF run-time environment. It is a sequence of 64-bit (and 128-bit) encoded instructions. Instructions are operations referencing registers (source and destination) or values (denoted as immediate). Each instruction is composed of:

- 8 bit opcode that represents an operation. Opcodes are grouped into classes based on the low 3 bits of the opcode. These classes include load, store, ALU, byteswap and branch instructions.
- 4 bit destination register (dst)
- 4 bit source register (src)
- 16 bit offset
- 32 bit immediate (imm)

ALU, byteswap and load/store operations manipulate data at 64/32/16/8 bits granularity. A wide 128-bit instruction encoding appends a 64 bit immediate value after a basic instruction.

There are eleven registers: r0 holds the return value for an eBPF program, r1-r5 hold the arguments for called functions, r6-r9 are callee-saved registers and r10 is a read-only register holding the stack frame pointer.

The call instruction uses imm as the index in a function pointer table maintained by the eBPF run-time environment.

The program counter is implicit. It cannot be manipulated explicitly by eBPF instructions. It is managed by the run-time environment based on the flow of execution. As a consequence, eBPF does not support indirect jumps, indirect function calls or jumps across eBPF programs[7].

eBPF instructions do not support floating point operations.

2.1.2 Compiler Support. Both gcc and clang/LLVM have an eBPF backend that generates eBPF programs from C programs[8].

Let us take a simple example that illustrates the compilation process and introduces some of the characteristics of eBPF that we will get back to later in the paper. Figure 1 is a simple C function that performs data massaging and calls an external function. More specifically, the simple function takes a pointer as an argument, casts it as a struct pointer (line 6, the struct is defined in lines 1-4), calls an external function (regfunc) using as a parameter the string component from the input struct (line 7), and appends an integer beyond the input struct in memory (lines 8 and 9). This integer is the product of the result of the external function with the integer element of the input struct.

Figure 2 is the corresponding eBPF program in LLVM assembly[9]. Register r1 contains the input pointer, which is first saved to a callee-saved register r6 (line 2), then incremented by the size of the integer n (4 bytes) so that it points to the s component of the input struct (line 3). Register r1 is now the input argument for the external function call (line 4). The integer value to which r6 points is then loaded into r1 (line 5). r1 is multiplied by the return value of the external function stored in r0 (line 6). Finally, r1 is stored at the first memory address beyond the input struct, i.e., the address pointed to by the contents of r6 + 104 (line 7). The return value of the eBPF program, 0, is assigned to r0 (line 7) and the eBPF program exits (line 8).

```
1  struct param {
2      int n;
3      char s[100];
4  };
5  int simple(void* ctx) {
6      struct param *p = (struct param *) ctx;
7      int v = regfunc(p->s);
8      ctx += sizeof(struct param);
9      * ((int*)ctx) = v*(p->n);
10     return 0;
11 }
```

Figure 1: A C program calling an external function (simple.c).

```
1  simple:
2      r6 = r1
3      r1 += 4
4      call regfunc
5      r1 = *(u32 *)(r6 + 0)
6      r1 *= r0
7      *(u32 *)(r6 + 104) = r1
8      r0 = 0
9      exit
```

Figure 2: The eBPF program generated from simple.c in LLVM assembly (clang -O1 -S -target bpf simple.c).

Note that the compiler does not take into account the characteristics of the eBPF run-time environment, which might only allow programs of a given size that verify predefined properties. For instance, the compiler can generate eBPF programs that contain unbounded loops.

2.1.3 Runtime Environments. There exists several software implementations of the eBPF ISA. Most prominently are: the eBPF subsystem within the Linux kernel, which consists of an interpreter and several JIT compilers along with a runtime environment; and uBPF which is a library that implements a virtual machine with a JIT designed to both in user-space and can be embedded to run in kernel-space.[10].

[6]The eBPF foundation now maintains a standard, currently eBPF Instruction Set Specification, v1.0: https://github.com/ebpffoundation/ebpf-docs.

[7]Within the Linux kernel, eBPF supports an additional operation: tail call. It relies on an additional function pointer map to execute jumps to other eBPF programs. (https://blog.cloudflare.com/assembly-within-bpf-tail-calls-on-x86-and-arm).

[8]There is no support in eBPF for variadic functions or polymorphic types. As a result, C++ programs cannot be compiled into eBPF.

[9]https://releases.llvm.org/15.0.0/docs/CodeGenerator.html

[10]uBPF is the core component of the eBPF for windows project (https://github.com/microsoft/ebpf-for-windows). Furthermore there is a Rust implementation similar to uBPF called rBPF (https://github.com/qmonnet/rbpf).

Recently, hardware implementations of the eBPF ISA have been proposed, most prominently hBPF[11] a project of the eBPF foundation, and Sephirot [6], a proprietary design specialized for packet processing.

A tool is now available to evaluate the conformance of an eBPF runtime to the Instruction Set Architecture defined by the eBPF foundation[12].

The code of external functions, that can be called from eBPF programs, is linked with the process that runs the run-time environment. When an external function is registered, its name and function pointer are added to the external function pointer map. A lookup based on an external function name returns its index in the function pointer map.

2.1.4 eBPF Lifecycle. We can distinguish the following phases in the life cycle of an eBPF program:

(1) *Produce bytecode* by compiling a C program;
(2) *Link bytecode* by replacing external functions with their index in the function pointer map, or replacing offsets into structs with their actual offset in memory;
(3) *Load bytecode* into the run-time environment;
(4) *Verify bytecode*, possibly rewriting it to be safe;
(5) *Execute bytecode* with a virtual machine or native assembly if it has been JIT'ed as part of phases 3 or 4.

The Linux eBPF verifier enforces that eBPF programs terminate without using too many resources and without leaking the contents of kernel memory. This verifier is in constant evolution. For intance, all loops were banned initially. This was too restrictive. Bounded loops have been supported since kernel 5.3. It is now over 15K lines of code[13]. In contrast, the uBPF loader merely checks that an eBPF program fits inside the allocated program slot. The eBPF for Windows project uses PREVAIL [11] together with uBPF.

2.1.5 Memory Management. We have so far mentioned, the stack and a program slot as regions of memory associated to the eBPF runtime environment. The size of the stack is a parameter of the environment, together with the maximum number of instructions in the program slot. In uBPF, they are set by default to 512B and 64K respectively.

In addition, mechanisms are provided to exchange data between the eBPF run-time environment and the outside world. In the Linux kernel, eBPF programs executed within the kernel can exchange data with user-space (or maintain state across executions) through key-value data structures denoted maps.

More importantly for us, the program that triggers the execution of an eBPF program usually passes a memory buffer as a context to that program. By convention, the first argument of an eBPF program is a pointer that can be used to refer to this context (as in Figure 1). Most programs using eBPF in the Linux kernel (e.g., socket filters, tc filters, XDP programs) and all uBPF programs respect that convention. With socket filters, for instance, the filter takes one argument, a pointer to an `__sk_buff` that has a field `data_end` is a pointer to the end of the context, so that its length can be computed.

[11]https://github.com/rprinz08/hBPF
[12]https://github.com/Alan-Jowett/bpf_conformance/
[13]https://github.com/torvalds/linux/blob/master/kernel/bpf/verifier.c

2.2 Computational Storage

The SNIA Computational Storage Architecture and Programming Model standard [5] distinguishes between (i) computational storage processors without storage (e.g., an accelerator card equipped with Eideticom NoLoad[14]), (ii) computational storage drives that encapsulate computational storage processor and device storage (e.g., Samsung SmartSSD [9]) and (iii) computational storage arrays that contain multiple (computational) storage devices (e.g., a flash array equipped with a DPU such as Fungible Storage Cluster[15]).

A computational storage processor is composed of: (i) computational storage engines (CSE), each able to execute functions in the context of a runtime environment, and (ii) memory, allocated from the device RAM, for a given function. The notion of computational storage engine abstracts various execution environments, such as OS image, container, FPGA bitstream or eBPF bytecode, which are given as examples.

We can dive a bit deeper into the relationship between host, computational storage processor and the underlying SSDs. We identify three possible architectures: (a) on-path over PCIe where the computational storage processor is a PCIe device for the host, and a PCIe root for the SSDs, (b) off-path over PCIe, where the computational storage processors and the SSDs are connected to the host in the same PCIe hierarchy domain so that they can communicate via peer-to-peer Direct Memory Accesses (p2pDMA), and (c) Smart NIC, where the computational storage processor is part of a NIC which is connected to SSDs via PCIe.

Regardless of the architecture, computational storage increases the cost of the storage subsystem. This cost is offset if computational storage makes it possible to (i) significantly reduce the use of core or RAM resources on the host (e.g., [8]), or to (ii) achieve application speedup through hardware acceleration of significant storage functions (e.g., [15]).

3 DELILAH

We designed and implemented `Delilah` to experiment with eBPF offload on an actual computational storage device. To the best of our knowledge, this is the first public description of such a system.

3.1 Requirements

We use the OpenSSD Daisy platform[16] as computational storage device. Daisy is the latest generation of OpenSSD prototypes, designed by Prof. Song and his team at Hanyang University [14]. The prototype is commercially available through CRZ Technology.

Daisy is an accelerator card, equipped with 2x100GE and PCIe Gen3x16 connectors, a Zynq Ultrascale+ Multiprocessor System on a Chip (MPSoC) and a backplane interface for connecting two M.2 SSDs. In the rest of this paper, we consider Daisy in the context of an on-path/PCIe architecture.

The Zynq Ultrascale+ MPSoC is a heterogeneous multiprocessing platform combining hardware acceleration on FPGA with the flexibility of ARM cores running embedded Linux. The ARM processor (Cortex A-53, denoted PS) only has access to peripherals

[14]https://www.eideticom.com/products.html
[15]https://www.fungible.com/product/nvme-over-tcp-fungible-storage-cluster/
[16]https://www.crz-tech.com/crz/article/Daisy/

(PCIe, RAM DIMMs, M.2 SSDs) through the FPGA (denoted PL). The boot image of the Daisy should contain:

- *A PL image*: An FPGA bitstream generated from a block design that may be used on generic IP (e.g., Xilinx IP for a DMA engine), or specific accelerators defined in a hardware description language. The Vivado toolchain is used to synthesize this bitstream.
- *A PS image*: Embedded Linux with specific programs cross compiled for the ARM cores.

The main requirement is that `Delilah` *should enable the offload of eBPF programs from a host onto Daisy. These programs should call registered functions that access stored data on the M.2 SSDs.*

It is necessary to verify that code offloaded from the host is not malicious. In particular, it is crucial to ensure that downloadable functions do not exploit the eBPF run-time environment to corrupt or deny access to stored data. This requires verification of the eBPF virtual machine or JIT compiler. Recent work has made significant progress on that front [18, 21]. In addition, registered functions and eBPF programs should be verified. While the Linux eBPF verifier is an obvious reference, its strict limitations (e.g., banning unbounded loops) are not well-suited for our purpose. Backward jumps are essential for data massaging, which makes termination analysis much harder. At the same time, termination of eBPF programs is important so that the device is not locked up by an infinite loop. We leave the verification of registered functions and offloaded eBPF programs as a topic for future work.

3.2 Design

At its core, `Delilah` implements a host-controller transport protocol through a driver module in the host kernel and a controller on the device. Figure 3 shows the architecture of `Delilah` (in green) and the components it relies on (in grey). In the rest of this Section, we detail the design and implementation of `Delilah` .

3.2.1 `Delilah` *Controller.* In Section 2.1.3, we identified three eBPF run-time environments: the Linux kernel eBPF run-time environment, the uBPF run-time environment in user-space and the eBPF hardware processor.

A thorough exploration of the use of the Linux kernel eBPF run-time environment or of a hardware-based eBPF run-time environment for computational storage are topics for future work. The Linux kernel eBPF run-time environment is the most robust and mature. It might be well-suited on a Smart NIC where data is transferred through the network and storage stacks inside the kernel. eBPF hardware processing should be well-suited on FPGA accelerator cards, specially in the context of off-path architectures.

In the rest of the paper, we focus on computational storage with the uBPF run-time environment, which gives us most flexibility for exploring the design space. The `Delilah` controller, the uBPF virtual machines and registered functions are part of the PS image. Note that hardware-accelerated functions are part of the PL image.

The RAM associated with a uBPF run-time environment is used to store an eBPF program, to maintain the state of the virtual machine (e.g., the map of pointers to external functions), and the context which is used to pass input to an eBPF program and collect its output (as we saw in Section 2.1).

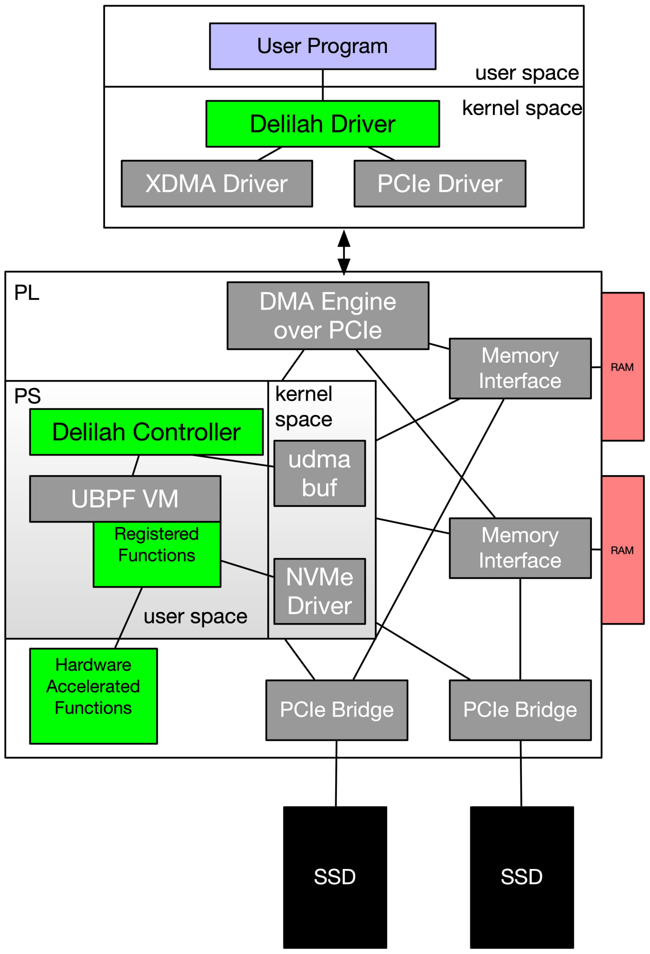

Figure 3: The `Delilah` architecture includes a host driver, a device controller and registered functions in user-space(PS) or hardware accelerated (PL).

The host-device protocol organizes how the host transfers programs and data to the device memory. It extends the Eid-Hermes protocol[17]. With PCIe, two mechanisms can be used to write data: (i) transfer through DMA or (ii) direct transfer through the memory-mapped region used for configuration (a so-called BAR aperture). Writes through the BAR aperture are usually reserved for coordination between host and device. This is how the host signals the device that a function should be executed. We use DMA to transfer programs and data.

3.2.2 Block Design. The `Delilah` controller must access Daisy peripherals (PCIe connection to the host and to M.2 SSDs and RAM DIMMS) through the PL. This requires a block design that defines a DMA engine over PCIe (for communication with the host), memory interfaces (for accessing RAM) and PCIe bridges (for communicating with the M.2 SSDs). Because of lack of space,

[17]https://github.com/Eideticom/eid-hermes/blob/master/specs/eid-hermes-theory-of-operation.md

we do not get into the details of our block design or discuss the interesting trade-offs that it involves.

3.2.3 `Delilah` *Driver.*

The main issue when designing the driver is to avoid overhead. We designed the `Delilah` so that it can be called directly with asynchronous commands, from programs running in user-space.

We defined the driver as a char device that defines uring commands. They are similar to ioctls but are asynchronous and can only be used with io_uring submission and completion queues [1].

The driver defines commands for transferring an eBPF program from the host to the device memory, for writing program arguments on the device memory, for triggering the execution of an eBPF program and for reading the result from device memory.

3.3 Implementation

We implemented the `Delilah` block design with Vivado 2019.1. The `Delilah` controller and the registered functions associated with the uBPF environment are programmed in C and cross compiled with gcc 12.1 for PetaLinux 2019.1, the embedded operating system running on the ARM processor (PS). The `Delilah` driver is also implemented in C and compiled with gcc 12.1. All code is available online [13].

3.3.1 *DMA Buffers.*

DMA is the usual mechanism to transfer data over PCIe. We rely on DMA transfers via XDMA[18] XDMA is a DMA engine over PCIe defined by Xilinx. We use the Xilinx IP in our block design and our driver uses the XDMA Linux driver.

The regions of the device memory that can be read and written from the host via DMA are defined as DMA buffers in XDMA. We rely on *user-space mappable DMA buffers*[19] to access them from the `Delilah` controller in user-space.

We define reserved memory in the PetaLinux device tree. The Linux contiguous memory allocator is used to allocate a contiguous DMA buffer that is then made available to the uBPF virtual machines. The DMA buffer is cached in the Zynq PS.

There is no means in `Delilah` to directly DMA data from the host into the ARM cores caches on the device. Such a feature is supported by Xeon processors (DDIO[20]) and via *cache stashing* on ARM processors based on the Dynamiq architecture, which is not the case for the ARM Cortex A-53, embedded on the Zynq Ultrascale+ MPSoC.

The cache is flushed before a program is loaded in the uBPF virtual machine to make sure that it accesses the contents that has been DMAed to the program slot and not old cached contents. Likewise, the cache is invalidated before and after program executed, so that (i) data DMAed by the host can be read by the eBPF program and (ii) data written by the eBPF program is flushed to memory and can then be DMAed to the host.

3.3.2 *Interrupts.*

In order to generate an interrupt, the PCIe DMA / Bridge IP must receive a signal on a given pin. In order to send this signal, the `Delilah` controller relies on a GPIO IP block and its associated driver, which is part of the Linux kernel.

[18]https://github.com/Xilinx/dma_ip_drivers/tree/master/XDMA/linux-kernel
[19]https://github.com/ikwzm/udmabuf
[20]https://www.intel.de/content/www/de/de/io/data-direct-i-o-technology-brief.html

The `Delilah` driver registers a callback in XDMA that is associated with the MSI-X interrupt. This callback triggers the uring completion.

4 EVALUATION

Our evaluation is essentially a sanity check of the host-device protocol. We quantify the overhead associated to offloading eBPF programs to identify potential problems in our implementation and identify potential optimizations.

4.1 Experimental Framework

We focus on latency as a metric for our experiments.

We run experiments on the Daisy platform equipped with a single SSD (Samsung EVO970). The Daisy is connected to a host via PCIe Gen3 x16. The host is equipped with Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz with 4 cores and 32 GB DDR3. It runs Ubuntu 22.04 (Linux 6.2.6, liburing 0ce8a73f) and clang 14.0 is used to compile eBPF programs.

A development server (12th Gen Intel(R) Core(TM) i7-12700KF with 12 cores and 32 GB DDR4), running Ubuntu 16.04 (Linux 4.15.0), is used to compile the block design, to offload FPGA bitstream and OS image to the Daisy platform before experiments [13].

4.2 Experimental Results

To evaluate the latency of eBPF code offload in `Delilah`, we rely on a simple eBPF program that takes as argument a file name and calls a registered function. that reads that file in the data slot. The files contain data sets of various sizes.

Figure 4 shows the end-to-end latency (in ms) associated with the four `Delilah` driver commands for (i) writing an eBPF program to a program slot on the device, (ii) writing the program's arguments in a data slot, (iii) executing the program and reading the result, when the program calls a registered function that reads a file whose size varies from 1KB to 100 MB.

Surprisingly, we observe that the latency for executing the program has a high floor of approximately 15 ms, regardless of the size of the data read by the registered function. As expected, the latency for writing program and data to the device is low and the latency for reading the output data is proportional to the size of the data transferred.

While program and data read/write are performed via DMA from the host, program execution entails a command sent by the host to the device via BAR0 (see Section 6.2.3) and the execution of the offloaded in uBPF with a registered function. We compare the baselines from Figure 4 with (a) the end-to-end latency of executing the eBPF program without registered function, and (b) the latency of executing eBPF program and registered function on the `Delilah` controller.

We break down the latency when offloading the eBPF program reading a 1KB file. The overhead on the host driver and the latency of executing eBPF and registered function are negligible. This is very positive. Even the latency associated to interrupt handling is small. Most of the time is spent invalidating and flushing the cache associated to DMA buffers. This time is proportional to the size of the cache, not to the amount of data.

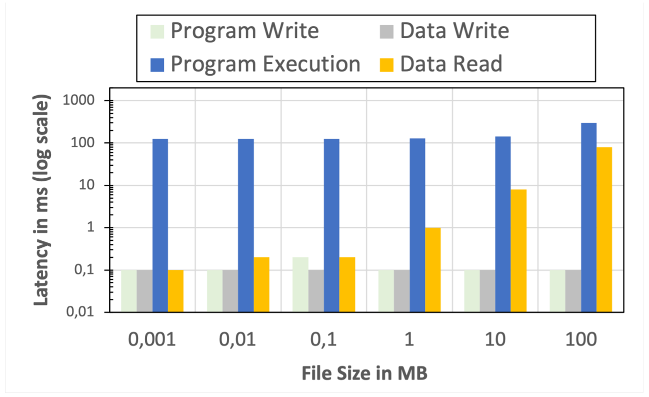

Figure 4: End-to-end latency (in ms) associated with the four `Delilah` driver commands.

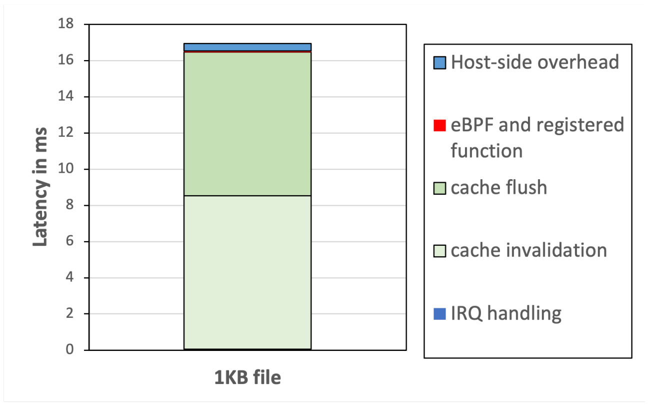

Figure 5: Breakdown of the latency (in ms) for end-to-end execution for a file size of 1 KB.

The cost of flushing the entire cache (as described in 3.3.1) is unacceptable. As an optimization, we introduce selective cache flushing, a mechanism that makes it possible for the host to communicate to the device to declare how many bytes must be invalidated both for reads and writes when sending the execution command. This state is easily maintained by the host, and the controller defaults to flusing the entire cache is this information is not provided.

Figure 6 shows the end-to-end latency for the 4 `Delilah` driver commands with selective cache flushing. Now, we observe that execution time is proportional to the amount of data processed, as it should be.

In Section 3.2.1, we mentionned the two mechanisms that exist for data transfer over PCIe: DMA and transfer over the BAR aperture. BAR has been used to write small log updates [16] while DMA is the default to read or write data pages. Let us compare transfer the latency cost for writing data with both mechanisms. We experimented with transfer sizes ranging from 1B to 100 MB. Figure 7 shows the result we obtained.

The key observation is that BAR is faster when the amount of data is small. BAR writes are fastest until data is 1 KB. DMA is superior when transfer size is greater than 10 KB. For us, it would

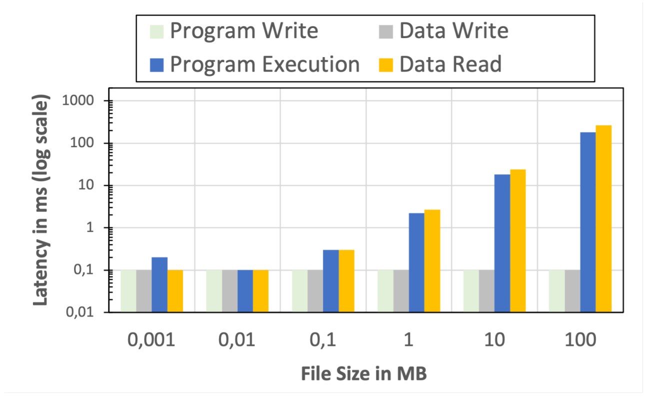

Figure 6: End-to-end latency (in ms) associated with the four `Delilah` driver commands using selective cache flushing.

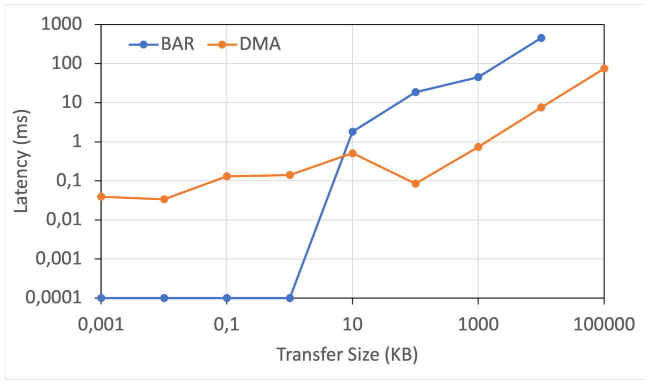

Figure 7: CMB vs. DMA transfer latency.

make sense to use BAR to write programs and use DMA to read and write data.

Here are directions for future work. First, we should focus on minimizing latency, e.g., using the uBPF JIT compiler rather than the interpreter. There might be other coherence issues that arise then. Second, we should complement the latency analysis presented in this paper with a throughput analysis (possibly altering the block design). Third, we should quantify the speedups that can be achieved with hardware accelerated functions, called from eBPF programs. Finally, we should evaluate `Delilah` for eBPF code offload in the context of a real-world data system. In particular, we need to evaluate how well-suited eBPF is for filtering, data massaging and feature engineering.

5 CONCLUSION

We presented the design and implementation of `Delilah`, the first publicly described system supporting eBPF code offload on a real computational storage platform, OpenSSD Daisy. Our design is representative of a class of computational storage processors, implementing an on-path/PCIe architecture. Much work remains to be done to explore how computational storage and eBPF code offload will benefit data-intensive systems.

ACKNOWLEDGMENTS

This work is developed in the context of the Daphne project [7] funded by the European Union's Horizon 2020 Research and Innovation Programme (grant agreement No 957407/Daphne).

REFERENCES

[1] Jens Axboe. 2022. What's new with io_uring? https://kernel.dk/axboe-kr2022

[2] R. Balasubramonian, J. Chang, T. Manning, J. H. Moreno, R. Murphy, R. Nair, and S. Swanson. 2014. Near-Data Processing: Insights from a MICRO-46 Workshop. *IEEE Micro* 34, 4 (2014), 36–42. https://doi.org/10.1109/MM.2014.55

[3] Antonio Barbalace and Jaeyoung Do. 2021. Computational Storage: Where Are We Today?. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2021/papers/cidr2021_paper29.pdf

[4] Stephen Bates and Kim Malone. 2022. NVMe Computational Storage – An update on the standard. https://www.youtube.com/watch?v=iTEgnqtwW44

[5] Stephen Bates, David Slik, Nick Adams, Scott Shadley, Bill Martin, and Arnold Jones. 2022. Computational Storage Architecture and Programming Model. https://www.snia.org/sites/default/files/technical-work/computational/release/SNIA-Computational-Storage-Architecture-and-Programming-Model-1.0.pdf

[6] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. 2022. hXDP: Efficient software packet processing on FPGA NICs. *Commun. ACM* 65, 8 (July 2022), 92–100. https://doi.org/10.1145/3543668

[7] Patrick Damme. 2022. DAPHNE: An Open and Extensible System Infrastructure for Integrated Data Analysis Pipelines. In *CIDR 2022, Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 4-7, 2022, Online Proceedings.*

[8] Jaeyoung Do, Ivan Luiz Picoli, Philippe Bonnet, and David B. Lomet. 2021. Better Database Cost/Performance via Batched I/O on Programmable SSD. *VLDB Journal* 30, 3 (2021).

[9] Samsung Electronics. 2022. Samsung Electronics Develops Second-Generation SmartSSD Computational Storage Drive with Upgraded Processing Functionality. https://semiconductor.samsung.com/emea/newsroom/news/samsung-electronics-develops-second-generation-smartssd-computational-storage-drive-with-upgraded-processing-functionality

[10] Guillaume Fournier, Sylvain Afchain, and Sylvain Baubeau. 2021. Runtime Security Monitoring with eBPF. In *17th SSTIC Symposium sur la Sécurité des Technologies de l'Information et de la Communication.*

[11] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A. Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. 2019. Simple and precise static analysis of untrusted Linux kernel extensions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 1069–1084. https://doi.org/10.1145/3314221.3314590

[12] Brendan Gregg. 2019. *BPF Performance Tools* (1 ed.). Addison-Wesley Professional.

[13] Niclas Hedam. 2023. Delilah. https://github.com/delilah-csp.

[14] Jaewook Kwak, Sangjin Lee, Kibin Park, Jinwoo Jeong, and Yong Ho Song. 2020. Cosmos+ OpenSSD: Rapid Prototype for Flash Storage Systems. *ACM Trans. Storage* 16, 3 (2020).

[15] Miryeong Kwon, Donghyun Gouk, Sangwon Lee, and Myoungsoo Jung. 2022. Hardware/Software Co-Programmable Framework for Computational SSDs to Accelerate Deep Learning Service on Large-Scale Graphs. 147–164. https://www.usenix.org/conference/fast22/presentation/kwon

[16] Sangjin Lee, Alberto Lerner, André Ryser, Kibin Park, Chanyoung Jeon, Jinsub Park, Yong Ho Song, and Philippe Cudré-Mauroux. 2022. X-SSD: A Storage System with Native Support for Database Logging and Replication. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 988–1002. https://doi.org/10.1145/3514221.3526188

[17] Steven McCanne and Van Jacobson. 1993. The BSD packet filter: a new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings (USENIX'93)*. USENIX Association, USA, 2.

[18] Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi Wang. 2020. Specification and verification in the field: applying formal methods to BPF just-in-time compilers in the Linux kernel. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, 41–61.

[19] Erik Riedel, Christos Faloutsos, Garth A. Gibson, and David Nagle. 2001. Active Disks for Large-Scale Data Processing. *Computer* 34, 6 (June 2001), 68–74. https://doi.org/10.1109/2.928624

[20] Alexei Starovoitov. 2014. LKML: Alexei Starovoitov: [PATCH net-next] extended BPF. https://lkml.org/lkml/2013/9/30/627

[21] Shenghao Yuan, Frédéric Besson, Jean-Pierre Talpin, Samuel Hym, Koen Zandberg, and Emmanuel Baccelli. 2022. End-to-End Mechanized Proof of an eBPF Virtual Machine for Micro-controllers. In *Computer Aided Verification: 34th International Conference, CAV 2022, Haifa, Israel, August 7–10, 2022, Proceedings, Part II*. Springer-Verlag, Berlin, Heidelberg, 293–316. https://doi.org/10.1007/978-3-031-13188-2_15

[22] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, and Asaf Cidon. 2022. XRP: In-Kernel Storage Functions with eBPF. 375–393. https://www.usenix.org/conference/osdi22/presentation/zhong