IT UNIVERSITY OF COPENHAGEN

Delilah: eBPF-offload on Computational Storage

Niclas Hedam[1], Morten Tychsen Clausen[1], Philippe Bonnet[1], Sangjin Lee[2], Ken Friis Larsen[3]

IT University of Copenhagen, Denmark[1]
University of Fribourg, Switzerland[2]
University of Copenhagen, Denmark[3]

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

Computational Storage

- Decades old idea of active storage and near-data processing.
  - Operate on data where it is stored, rather than moving it to the CPU.
  - Historically, one device per application; every device has its own interface.
- Potential breakthrough due to standardisation.
  - SNIA has standardized an architecture:
    - Computational Storage Processor vs. Drives execute functions in the context of I/Os
    - Device-specific / Downloadable functions
    - eBPF proposed as vendor-neutral Instruction Set for Downloadable functions.
  - NVMe also proposes eBPF for computational storage in TP 4091.
    - Caveat: The standard is still not public.

eBPF on Computational Storage

A downloadable eBPF function may orchestrate calls to different device-specific functions, perform operations and data massaging.

Open Questions:

1. **How to execute eBPF-based downloadable functions on computational storage?**
2. **Can eBPF-based computational storage functions be executed efficiently?**

Contributions

Delilah is the first public prototype of an actual computational storage device supporting eBPF-based code offload.

1. Survey of eBPF Ecosystem
2. Design of Delilah
3. Evaluation of Host-Device Protocol

Related Work

- First generation of commercial computational storage was based on proprietary interfaces.
  - Most promintently: ScaleFlux, NGD, Samsung SmartSSD.
- ZCSD is a computational storage platform with eBPF offload on Zoned namespaces, **emulated in QEMU**.
- Eid-Hermes project defines a protocol for eBPF offload over PCIe, **emulated in QEMU**.
- Wilcox and Litz propose an architecture for code offload on computational storage, **emulated in QEMU**.

eBPF

- eBPF is a vendor-neutral Instruction Set Architecture.
  - Governed by the eBPF Foundation (https://ebpf.foundation/)
- An eBPF program is a sequence of 64-bit encoded instructions.
  - 8 bit opcode
  - 4 bit destination register (dst)
  - 4 bit source register (src)
  - 16 bit offset
  - 32 bit immediate (imm)
  - Most instructions do not use all fields

Instruction	Opcode	Mnemonic
Function call	0x85	call imm
dst += imm	0x07	add dst, imm
PC += off if dst == imm	0x15	jeq dst, imm, +off

eBPF Lifecycle

1. Produce bytecode by compiling a C program

```c
struct param {
    int n;
    char s[100];
};
int simple(void* ctx) {
    struct param *p = (struct param *) ctx;
    int v = regfunc(p->s);
    ctx += sizeof(struct param);
    * ((int*)ctx) = v*(p->n);
    return 0;
}
```

```
simple:
    r6 = r1
    r1 += 4
    call regfunc
    r1 = *(u32 *)(r6 + 0)
    r1 *= r0
    *(u32 *)(r6 + 104) = r1
    r0 = 0
    exit
```

2. Link bytecode (external functions, structs into offset)
3. Load bytecode into the run-time environment
4. Verify bytecode, possibly rewriting it to be safe
5. Execute bytecode

eBPF Run-Time Environments

- eBPF has multiple existing runtimes
  - Within the Linux kernel – execution of user-space programs in kernel for monitoring/profiling
  - User-space - **uBPF** and rbpf
  - On FPGA - hBPF
- Delilah relies on uBPF
  - Interface is load/unload/execute.
  - Provides programs with a contiguous memory buffer.
    - Buffer holds data and state at runtime.
  - eBPF programs access device-specific functions that are registered within the uBPF virtual machine.
    - We call these registered functions.

eBPF: Known Challenges

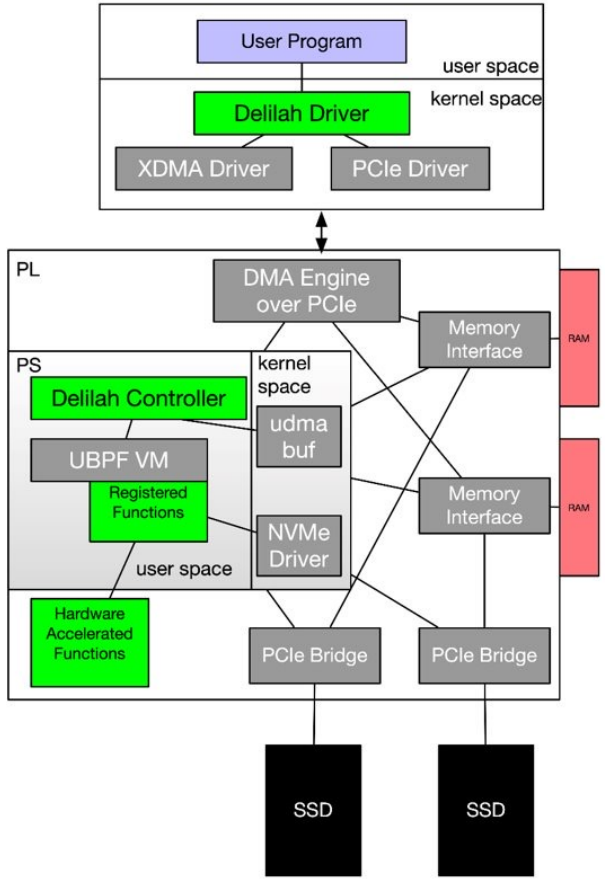
- Limitations of eBPF
  - No floating-point arithmetic
  - Limited number of registers
  - General-Purpose ISA

- Limitations of eBPF compiler/run-time
  - Limited stack size in clang compiler
    - Fixed in clang-17

Delilah Requirements

- Delilah have several main requirements.
  - It must run on the Daisy OpenSSD.
    - I.e. Delilah must have a controller which manages one or more uBPF VMs.
    - Delilah must be able to access the host and other peripherals via the FPGA programmable logic.
  - It must be able to execute eBPF-based downloadable functions.
  - It must access underlying storage mediums via registered functions.

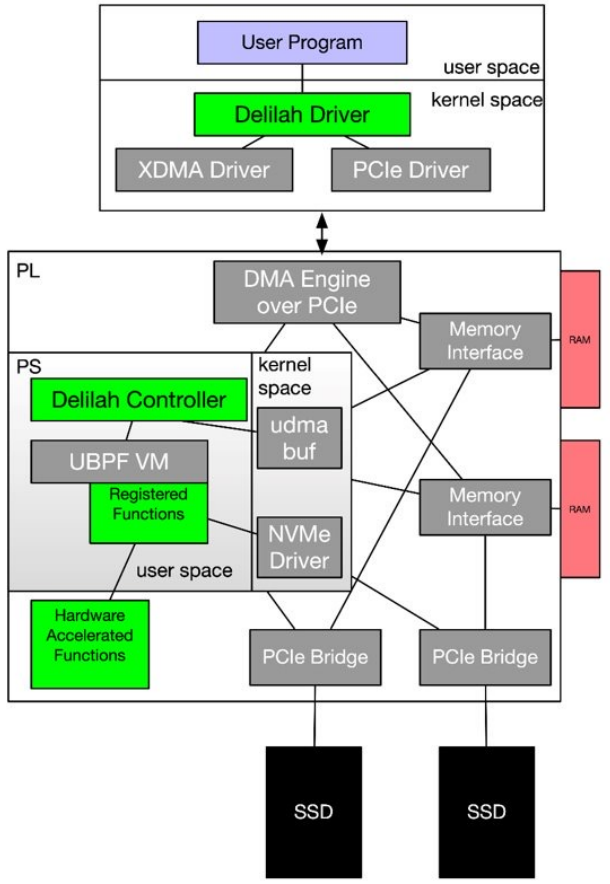
Delilah Architecture



- Delilah is divided into several subsystems
  - PCIe/XDMA driver on host
  - Controller in processing system
  - Peripheral organisation through programming logic

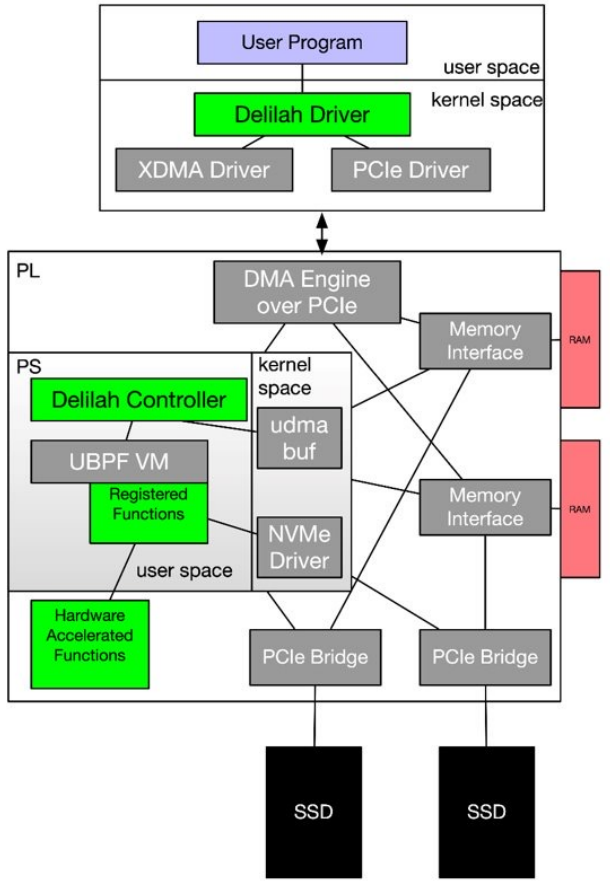
Delilah Design - Controller

- The controller runs within the ARMv8 and is in charge of:
  - Managing uBPF virtual machines.
  - Listening for submitted commands in the Base Address Register (BAR).
  - Initialisation of support systems.
- The controller and protocol is synchronous.
  - It’s up to the driver to manage queues and schedule execution.



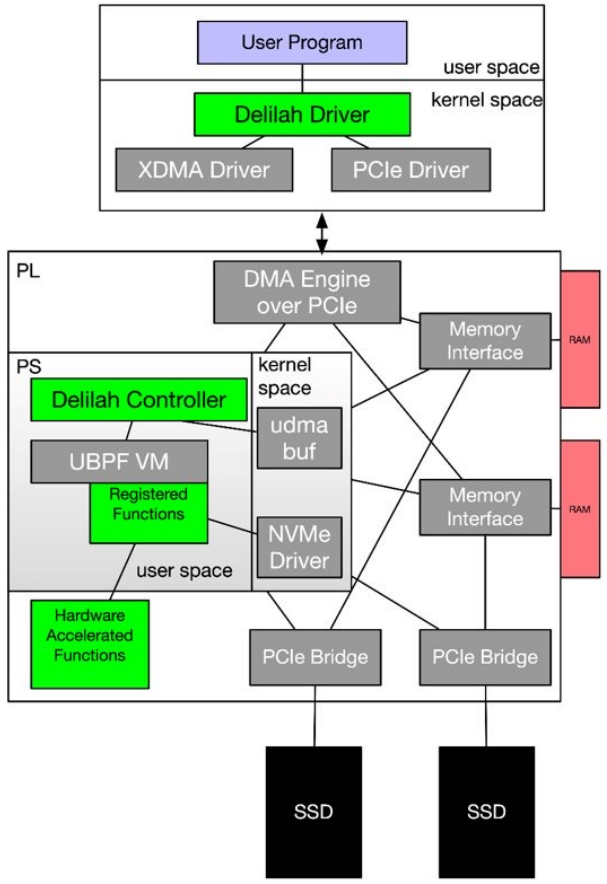
Delilah Design - Controller

- The controller exposes two types of buffers to the host.
  - 16 x 4 MB program slots, which holds eBPF instructions to be executed on demand.
  - 12 x 1 GB data slots, which holds state and data.
- Buffers are stored on physical addresses on Daisy's DDR4.



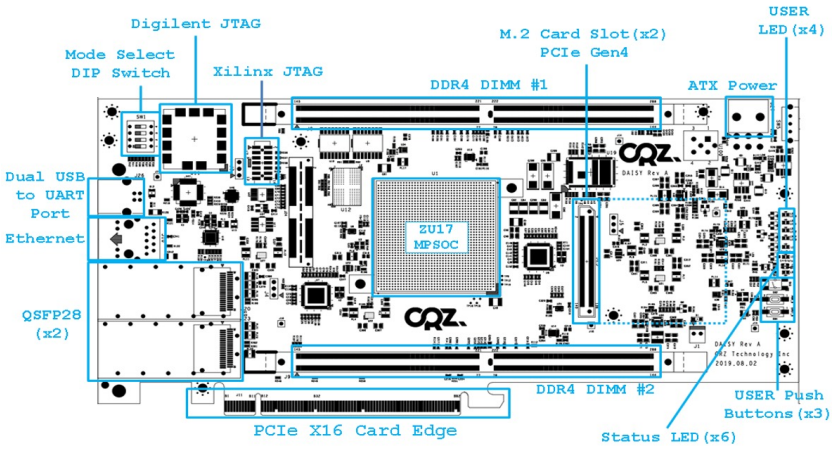
Delilah Design – Driver

- The driver is exposes Delilah devices as a character device.
  - The character device only accepts io_uring commands.
- The driver acts as an extension of XDMA, the Xilinx/AMD DMA engine.
- The driver queues I/Os via kernel work queues and handles them with the FIFO principle.
- Since programs may be coupled to specific data slots, it cannot be queued.
  - The application must handle scheduling.



Delilah Computational Storage Processor

- Daisy is equipped with a Multi Processor System on a Chip: Ultrascale+ FPGA associated with an ARM V8 processor.
- All peripherals are connected to the FPGA.
  - 2x DDR4 DIMM slots.
  - 2x 100G QSFP28.
  - 2x PCIe3 M.2 slots.
  - 1x PCIe3x16 connector.



Images by CRZ Technology – http://mangoboard.com/main/view.asp?idx=1056&cate1=9&cate2=25&cate3=

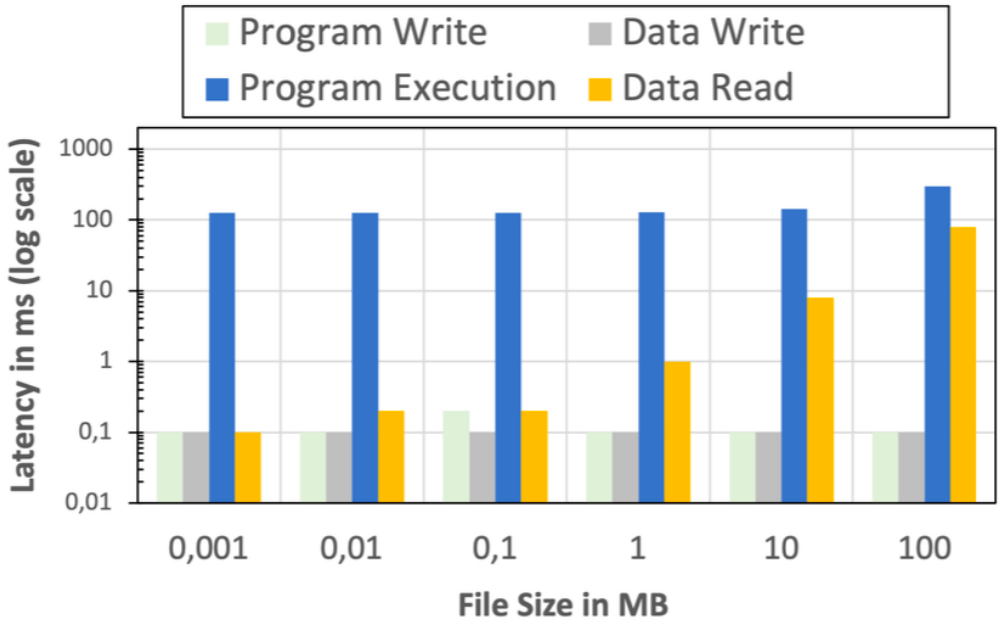
Delilah Implementation

- The controller is written in C as a Petalinux application.
  - Petalinux is the Yocto-based SDK for running an operating system on MPSoCs.
  - We use Petalinux 2019.1.
- The controller is connected to peripherals via a block design
  - The block design is targeted solely at the Daisy OpenSSD and enables Petalinux to access RAM, SSDs, PCIe, ..
  - We use Vivado 2019.1.
- Data/program slots mapped using udmabuf.
  - Udmabuf is responsible for mapping physical addresses to virtual addresses and managing low-level subsystems like the CPU cache.
  - Without udmabuf, Delilah would need its own kernel-space module.
- Interrupts to host triggered via GPIO.
  - The GPIO subsystem in Petalinux is used to trigger interrupts in the DMA engine.

Experimental Framework

- Experiments run the Daisy with a single SSD (Samsung EVO970).
  - Connected to a host via PCIe3.
    - Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz with 4 cores and 32 GB DDR3.
    - Ubuntu 22.04 (Linux 6.2.6, liburing 0ce8a73f), clang 14.0.
  - Connected to a development server via JTAG.
    - 12th Gen Intel(R) Core(TM) i7-12700KF with 12 cores and 32 GB DDR4.
    - Ubuntu 16.04 (Linux 4.15.0).
- To experiment with eBPF and Delilah, we measure the latency of reads to the underlying SSD to the host via Delilah.
  - Reveals bottlenecks and potential problems with the architecture.
  - Workload size is 1KB to 100 MB.

End-to-End Latency

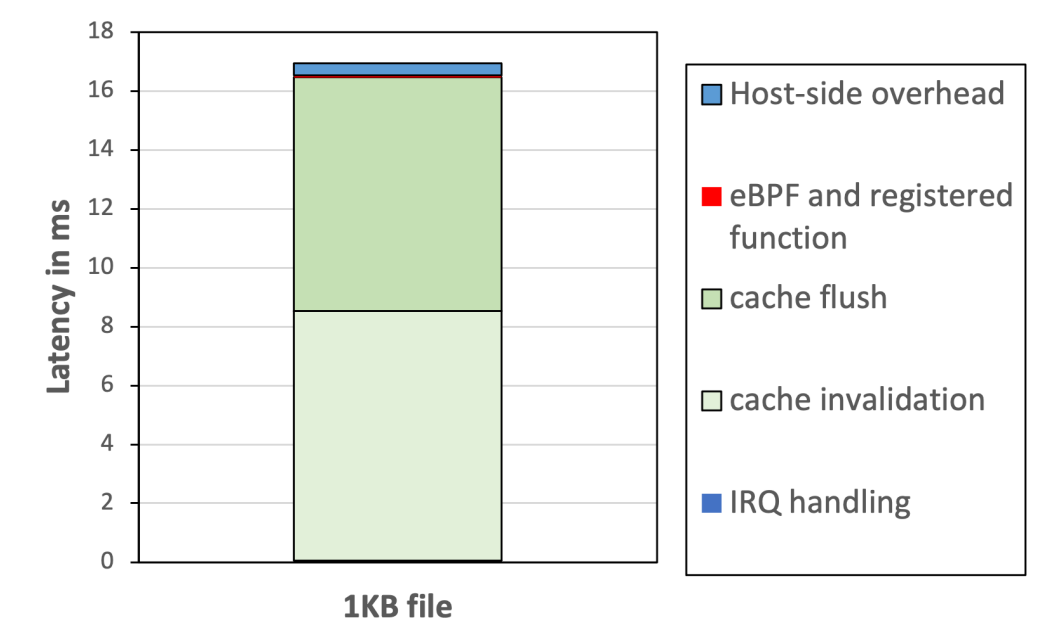


- We expect that program execution and data reads scale with file size.
- We observe that data reads scale with the size of the transfer.
- However, Program execution remains constant until ~50 MB.

Why ?

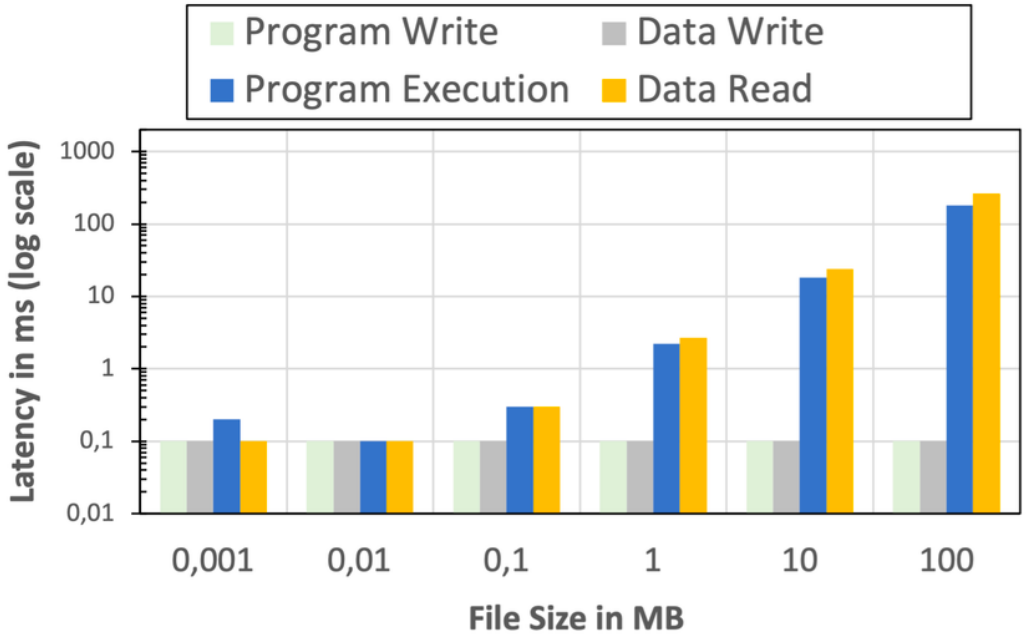
CPU Cache Coherency

- 95% of program execution time is spent synchronizing CPU and DMA buffers.
  - The actual execution of the eBPF programs takes a few micro-seconds.



Selective Cache Invalidation

- We introduce a mechanism to only invalidate relevant parts of the cache.
  - When executing, the host can define how much of the cache must be flushed or invalidated.
- With this optimisation, execution time is proportional to file size.



Lessons Learnt

- Maintaining coherence of DMA regions is a key issue.
- The host-controller protocol must support identification of registered functions.
  - I.e. what registered functions are available? What are their signatures? How do eBPF programs reference these functions?
- A tight coupling of program and data slots is necessary to support eBPF offload in the context of computational storage.
  - Contents of program slots are static (not changing after offload), while data slots are dynamic. Since data slots may hold program state, it is necessary in some circumstances to couple specific program slots and data slots together.
- How to organize support for parallelism and state management (including concurrency control and recovery) within registered function is an open issue.

Future/On-going Work

1. Improving Delilah performance and functionalities
   - Focus on memory latency/throughput

2. Integrating Delilah and database systems
   - Collaboration with TU Dresden Database Group

3. Benchmarking computational storage

Conclusion

- TP 4091 might be the breakthrough of computational storage
  - SNIA and NVMe proposes eBPF as offload mechanism
- eBPF is a light-weight vendor-neutral Instruction Set Architecture
- We designed and implemented Delilah, the first publicly described eBPF CSP
  - We experiment with Delilah on top of the Daisy OpenSSD
- Experiments reveal importance of DMA cache coherency
  - Selective Cache Invalidation as potential solution
- Future work includes integration with database systems and benchmarking