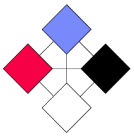
D6.3 Prototype and overview of data path optimizations and placement



DAPHNE

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

Version 1.0

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 Report and prototype of used data path optimization techniques and automatic data placement in hybrid memory and storage configurations. This deliverable describes the second version of the demonstrator of the Delilah computational storage prototype. The report also describes data path optimizations and placement in the context of the DAPHNE storage subsystem, including hardware acceleration on the data path.

D6.3 Prototype and overview of data path optimizations and placement			
WP6 – Computational Storage			
Type of document	D	Version	1.0
Dissemination level	PU	Project month	36
Lead partner	ITU		
Author(s)	Marcus Paradies, Philippe Bonnet, Constantin Pestka, Niclas Hedam, Alex Krause.		
Reviewer(s)	Ilin Tolovski, Patrick Damme		

Revision History

Version	Revisions and Comments	Author / Reviewer
V0.1	Structure of the document	Philippe Bonnet
V0.2	First Complete Draft	Marcus Paradies, Philippe Bonnet, Constantin Pestka, Niclas Hedam, Alex Krause
V1.0	Feedback from reviewers incorporated	Marcus Paradies, Philippe Bonnet, Constantin Pestka, Niclas Hedam, Alex Krause.

1. Introduction

In the context of DAPHNE, we are considering archival data sets (e.g., national or international data services, cloud-based storage services) that are staged on a HPC cluster or a data scientist laptop where the DAPHNE pipelines are run. We are focusing on data placement in connection with the execution of a DAPHNE pipeline. In this context, as described in deliverable D6.2, crucial issues with the DAPHNE run-time system are:

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 limitations, through careful, asynchronous data movement across storage tiers and by leveraging computational storage.

This deliverable reports on the demonstrator that illustrates the progress made with the Delilah prototype supporting code offload on computational storage (addressing issue (b) above). The design and implementation of Delilah were documented in an article "Delilah: eBPF code offload on Computational Storage" by Niclas Hedam, Morten Tychsen Clausen, Philippe Bonnet, Sangjin Lee and Ken Friis Larsen [1]. In this report, we report on additional work done to integrate data filtering on computational storage to minimize data movement. This is the main feature of the Delilah demonstrator.

Before, we describe the Delilah demonstrator, we discuss in depth asynchronous I/Os and Chunked I/Os, and their use in the context of DAPHNE. Asynchronous I/Os and Chunked I/Os are the cornerstone of an architecture where data can be loaded piecemeal without blocking DAPHNE operator pipelines (addressing issue (a) above). These are the key techniques we consider with respect to data path optimization and data placement in this report.

2. DAPHNE Asynchronous I/Os

Due to the rapidly growing performance of modern I/O devices, most notably SSDs, the overheads inherent in legacy I/O stacks have increasingly become a bottleneck. To tackle the performance overhead in legacy I/O APIs and storage stacks, a variety of novel I/O APIs have been proposed and implemented, most notably io_uring (linux) [2] and IO Rings (windows). Other APIs, such as the Storage Performance Development Kit (SPDK), reduce the overhead of the legacy I/O stack even further by bypassing the kernel I/O stack entirely [3]. One of the most important design goals of all modern I/O APIs is thus the elimination of most of the system calls required to perform I/O operations. This is due to the relative cost of system calls constantly increasing, as the performance of physical I/O devices, such as NVMe SSDs, have and continue to grow rapidly.

The most notable design commonality between these modern I/O APIs, however, is their asynchronous design in contrast to the blocking behavior of legacy I/O APIs (e.g., the POSIX related I/O operations). In the blocking design, control flow is completely taken away from an application upon performing I/O, if it only possesses one thread. The application is hence unable to make any further forward progress. A common pattern to mitigate this issue is to spawn many threads and rely on the OS scheduler to assign control flow back to another thread of the main application upon hitting an instance of blocking I/O on a given thread. Besides the chance for the potentially unwanted side effect of another process being scheduled instead of another thread of the application in question, this approach can cause potentially numerous context switches. These stem from the initial system call required for each I/O operation in blocking APIs, the scheduling to another thread, if an I/O operation is setup to be interrupt-based and not completed inline, the scheduling back to kernel space once the related interrupt arrives, as well as the context switches required for the thread creation, as well as spurious wake-ups.

Background on Asynchronous I/O

APIs based on asynchronous completion have the potential to alleviate all of the overheads. The system call-based submission and completion of I/O requests is replaced with two shared ring buffers, the submission queue (SQ) and completion queue (CQ). From a user-space perspective, I/O operations are performed by inserting *submission queue entries (SQEs)* to the SQ and then polling for *completion queue entries (CQEs)* in the CQ. The user-space thread retains complete agency over its control flow and gains the ability to prioritize certain objectives, such as optimizing for end-to-end latency for a specific set of I/O operations or to prioritize specific application logic over polling for completions.

Challenges and Design Decisions:

In-Line vs. Separated I/O Handling. Since the application logic can run concurrently to the execution of I/O tasks within the same thread, the user must decide whether both types of tasks should run in the same thread or whether application logic and I/O related tasks should be separated into different threads. While the inline execution of application logic can decrease the latency of application tasks that depend on the completion of I/O tasks, this leads to reduced *Quality of Service (QoS)* and a lower overall I/O throughput. Conversely, completely separating I/O tasks into dedicated threads requires adequate synchronization among those threads, which adds complexity and additional overhead.

Sharing of I/O API Instances. A related issue is the shared access to I/O API instances from multiple threads. Low-level I/O APIs, such as io_uring, utilize mostly acquire-release or even relaxed memory ordering for synchronization. While this enables high performance, it makes the shared access to an I/O API instance (e.g., to an SQ of a single I/O API instance) from

multiple user space threads not thread-safe without additional synchronization (e.g., by adding additional sequentially consistent memory barriers).

Due to memory ordering-based synchronization being famously difficult and error-prone, as well as the potential of severe congestion and thus performance implications, it is generally recommended to restrict access to a specific single thread, either for both SQ and CQ, or to two threads handling them separately. There are, however, valid use cases that make sharing I/O instances desirable. The first reason, depending on the configuration and workload, is the potentially large setup and constant overhead associated with each I/O instance. While there are many more and frequently optional contributions to these overheads, the most relevant are dedicated associated threads. These range from the user-space I/O thread itself, the potentially many dedicated, associated kernel threads for polling of the SQ and underlying hardware queues, as well as for performing work required due to the used part of the I/O stack (e.g., the file system). The significant cost of I/O instances implies that reducing their number to the minimum required will be beneficial for overall efficiency. Reducing the number of I/O instances to a minimum, however, implies that fewer application threads are available, if each thread possesses its exclusive I/O instance or that I/O instances must be shared, which yields synchronization challenges.

Out-of-order Request Handling. Another scheduling related issue that arises is due to the non-deterministic arrival order of CQEs. For the blocking I/O APIs the logic surrounding an I/O operation is organized in a linear, contiguous stream of operations. Each blocking I/O operation is simply treated as one of such operations. This keeps logically connected parts of the application also directly connected with regards to control flow. This reduces complexity and is a design pattern, software developers are very familiar with. However, due to the out-of-order arrival of CQEs, keeping this design pattern using the asynchronous I/O APIs is not possible. This issue is further exacerbated by the demand to handle arrived CQEs in a timely fashion to avoid reducing I/O throughput and the QoS of other involved I/O requests.

Asynchronous I/O in DAPHNE

Asynchronous I/Os have the potential to deliver significant performance and efficiency benefits. However, the aforementioned challenges show that effective usage of asynchronous I/O APIs in complex applications is non-trivial and error-prone. In the following we describe the design of asynchronous I/O in DAPHNE in the light of the previously mentioned challenges.

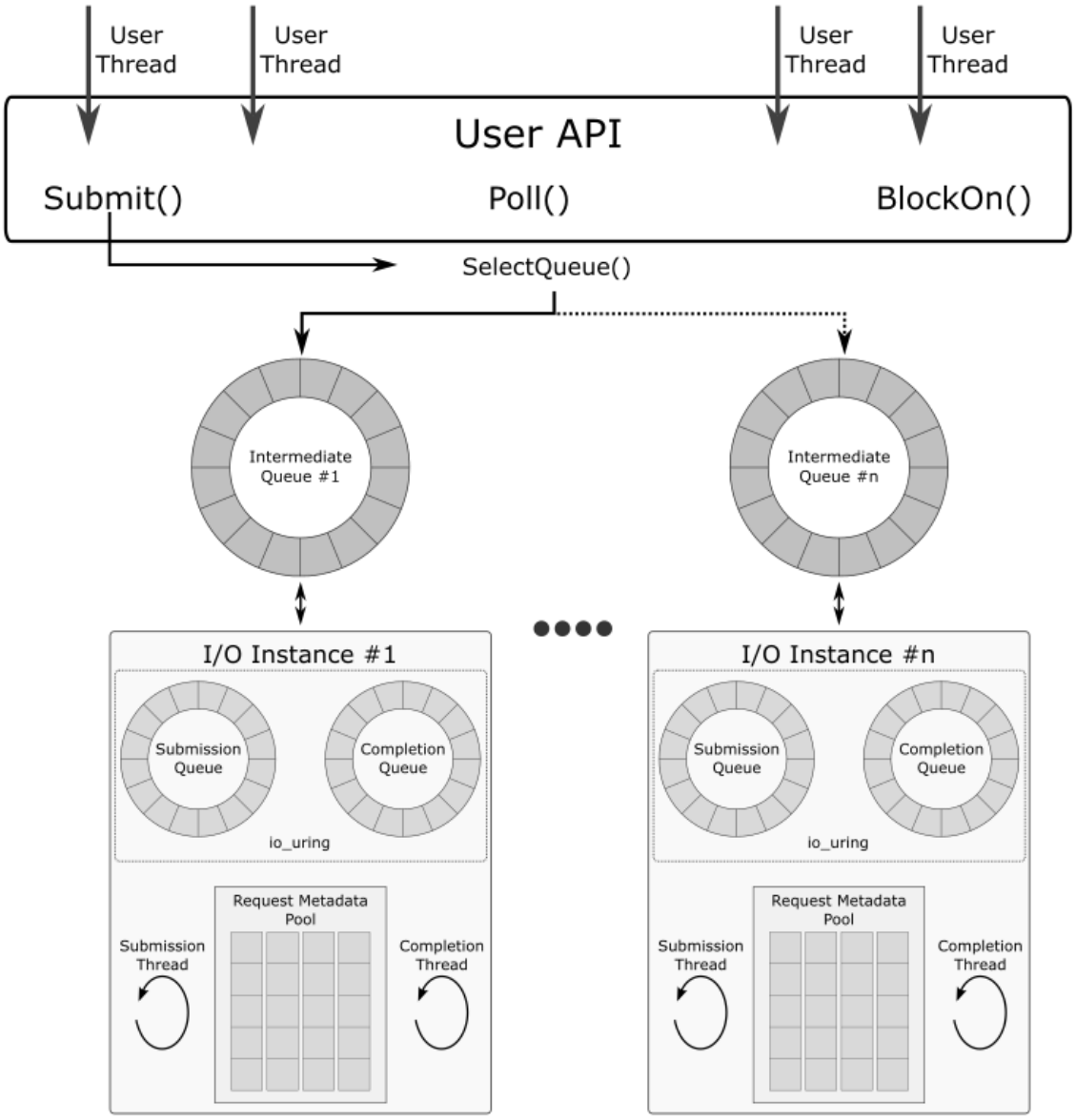


Figure 1: Architecture of the asynchronous I/O handling in DAPHNE

Figure 1 shows the overall architecture of the asynchronous I/O handling in DAPHNE and depicts the main components and their interactions. We choose to separate I/O tasks from other application logic. For this purpose, we utilize either one or two dedicated I/O threads per I/O instance. Each I/O instance encapsulates a single instance of one of the underlying low-level I/O APIs (e.g., io uring). To enable the degree of parallelism required to saturate one or multiple modern storage devices (e.g., high-performance NVMe SSDs), we employ a configurable number of I/O instances but disallow the direct sharing of the encapsulated I/O instances among threads. We disallow this shared access to avoid congestion on the CQ due to excessive polling from multiple threads. The ability to configure the number of I/O instances enables users to scale the resources spent on I/O tasks independently from resources spent on other tasks.

To exploit the parallelism enabled by multiple I/O instances, the requests must be distributed to said instances. As already stated, the direct access to, e.g., the SQ of an instance of io uring, is, per default, not safe for multiple user-space threads. The distribution of requests must either utilize additional memory barriers or utilize an additional intermediate data structure with more heavy-weight synchronization realized through locks or sequentially consistent atomics. In DAPHNE we employ one *submission staging queue* per I/O instance. A submission staging queue acts as an interface between, on the one side, an arbitrary number of user-space threads submitting requests, and on the other side, the dedicated thread of an I/O instance that is responsible for submission to the underlying low-level I/O API instance. The distribution of requests from the user API to the submission staging queues is performed in a round-robin manner. This offers both, a simple form of load balancing, as well as reduces congestion, as said queues are exclusive to a given I/O instance. This approach enables the Submit() function of the user API to be completely thread-safe. Users hence do not need to concern themselves with the non-trivial memory ordering-based synchronization of the internal underlying I/O instances.

The central object for the user regarding the results of an I/O operation are *request futures*, which are returned by the Submit() function. The user API provides the functions Poll() and BlockOn(), which take one or more of such requests futures as argument and enable the user to poll or block on the completion of specific requests. This is notably different to the behavior of the underlying low-level I/O API and is our design approach to handle the inconvenience of the out-of-order arrival of CQEs. As previously mentioned, a single I/O instance possess either one or two dedicated threads. In the case of two threads, one handles the submission and one the completion of requests. In the case of one thread, the thread handles both tasks. The submission executed by the respective I/O thread begins with the thread acquiring a batch of requests from its dedicated staging submission queue. It should be noted that the thread, before doing so, ensures that there are not too many requests currently already in-flight. This is done since, while overflowing the SQ or CQ ring buffer is safe in newer versions of io uring, it introduces some performance overhead, due to required allocations and other book-keeping in the kernel space but does not deliver any tangible benefits for our case. The thread then submits the acquired requests to the SQ. This involves the allocation of SQEs, filling in the SQEs based on the type of operation requested (e.g., a read) alongside required meta data (e.g., the file descriptor, read offset, and size) and finally submitting it to the SQ of the internal, low-level I/O instance of the underlying I/O API, such as io uring. To correctly handle the completion of a request, the mentioned meta data must be stored, alongside an identifier, as long as the request in question remains in-flight. The corresponding meta data objects are held in the *request meta data pool*, which utilizes a pre-allocated arena and acts as a slab allocator for the meta data objects. We use this approach here over a simpler and generally more efficient ring buffer, which cannot be used here due to the out-of-order arrival of CQEs and its comparatively lower cost than a general-purpose allocator such as malloc.

Once the execution of the requests has been completed, they arrive as CQEs in the CQ. At this point the generic part of the required processing is performed by the thread of the I/O instance dedicated to handling completions. This generic part is mostly concerned with checking return values and consequently propagating up the results to the request futures in the case of

success and failure, but also handling resubmissions in case of partially successful operations. This requires the associated metadata to be updated, as well as the request to be reinserted into the staging submission queue.

To summarize, asynchronous I/O within DAPHNE provides the following features and benefits:

- Parallelism required to saturate modern storage devices via the usage of multiple I/O instances.
- Maximum IOPS and QoS per I/O instance, due to two dedicated threads responsible for submission and completion, respectively.
- The ability to tune I/O resource consumption via adjusting the number of io_uring instances used.
- Easy-to-use user API only consisting of three functions: Submit(), Poll(), and BlockOn().
- Removes the need for familiarity with memory-ordering based synchronization via a fully thread-safe user API.
- Encapsulates many of the pitfalls and tedious tasks of the low-level I/O APIs (e.g, the need for resubmission of partially completed operations).
- Delivers a simple and effective form of load balancing via round-robin distribution among the used I/O instances.

3. Chunked I/Os

Splitting stored data into chunks is important for several reasons. First, it is a prerequisite for using a distributed file system such as HDFS [4]. Second, in cases chunks exhibit locality in space or time , accessing only relevant chunks is a means to work with only the relevant portions of a very large data sets. Third, chunks are a unit of data placement. A chunked data layout in storage provides an efficient partitioning scheme, which allows large data sets to be stored in (distributed) file systems and object storage systems efficiently.

In a file system context, each chunk is either stored as a separate file or as a well-defined portion of a file and and can be accessed independently. In an object storage system, the chunk index can be directly used as the object key to locate and retrieve an individual chunk from the storage system. To fully exploit available I/O parallelism and asynchronous I/O capabilities, in-memory data structures holding the data must support chunking as well.

In the rest of this Section, we describe chunked data layout and chunked I/Os in the context of a specific storage format, which is relevant for DAPHNE:: Zarr.

Zarr

Zarr[1] is an open specification of a storage format for multi-dimensional, typed arrays (also known as tensors). A focus of Zarr is the support for storage of large volumes of data using distributed storage systems (e.g., object stores, distributed file systems) and efficient handling of parallel I/O requests. Zarr data can be stored in any storage system that can be represented as a key/value store, which includes most POSIX-compliant file systems, cloud object storage systems, but also relational and document-oriented database management systems.

Each Zarr array has associated mandatory metadata (stored in JSON format), which allows for a correct interpretation of the stored data. The metadata description consists of mandatory fields, such as the *shape specification*, the *chunk specification*, the data type associated with the variables, a default *fill value*, and the *byte order*. Additionally, *compression* and *filter* specifications can be added to describe if and how the data stored was transformed/filtered and/or compressed upfront.

Zarr arrays are typically chunked (split into smaller pieces as atomic unit of data retrieval) using the *chunk specification*. A chunk is an optionally compressed sequence of bytes, which can be only properly interpreted with the help of the array metadata fields. The compressed sequence of bytes for each chunk is stored under a key formed from the index of the chunk within the grid of chunks representing the array. To form a string key for a chunk, the indices are converted to strings and concatenated with a delimiter character (default: ".") separating each index. For example, an array with shape (10000, 10000) and chunk shape (1000, 1000) is represented as 100 chunks laid out in a 10x10 grid. The chunk with indices (0, 0) provides data for rows 0-999 and columns 0-999 and is stored under the key "0.0" and so on.

Operating on Chunked Arrays in DAPHNE

The DAPHNE Run-time now supports chunks when storing/reading data from the binary DAPHNE storage format [2]. We have built initial support for a chunked tensor data structure, where individual chunks can be read from storage through the corresponding chunk indices and be loaded into main memory.

The availability of chunking is a necessary precondition for the introduction of a buffer manager, which handles frequently accessed chunks of data in a faster tier of the storage and memory subsystem of DAPHNE. As a second step, from a runtime perspective, existing execution kernels must be extended to fully support chunk-based data processing (as opposed to full-dataset loading and processing). Finally, a chunk-based data processing also requires changes to the DAPHNE compiler to reason about and optimize data access to large data sets, where only a small fraction of the overall data set is required to process the request.

[1] https://zarr.dev/

4. Delilah Demonstrator

The Delilah demonstrator is joint work between TU Dresden and IT University of Copenhagen.

Demonstrator Setup

The demonstrator focuses on the offload of portion of a filtering kernel in Delilah. The Delilah demonstrator is available in an accompanying zip file.

The demonstrator folder contains:

- Block Design: The FPGA block design used to deploy Delilah
- Petalinux: The Petalinux image used to deploy Delilah
- Host Driver: The host-side driver used to interact with Delilah
- Delilah Firmware: The firmware deployed to Daisy
- Filtering Application: A filtering application built using co-design between host and Delilah.

Delilah is built and targeted at the Daisy OpenSSD. In principle, it is possible to deploy Delilah on any Xilinx Zynq Ultrascale+ MPSoC. However, the block image will have to be tweaked to match the specifics of this device. For this demonstrator, we assume all deployments are targeted at the Daisy OpenSSD.

The demonstrator was tested with a host running Linux v5.9. The block design is built using Vivado 2019.1. The Petalinux image is built using Petalinux 2019.1.

The Petalinux folder is configured with the correct block design and precompiled with Delilah. Petalinux should be transferred to the Daisy OpenSSD via a SD card. This SD card should have at least 8 GB of space. 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.

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 Delilah has started up successfully after a few minutes, continue.

```
# delilah
```

Delilah is now running on the Daisy OpenSSD. Turn on the host that the Daisy OpenSSD is connected to. Go to the Host Driver folder.

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

You can verify if the driver is loaded correctly by checking if /dev/delilah0 has been created. Compile the evaluation suite by going to Filter Application on the host:

```
# mkdir build
# cd build
# cmake ..
# make
# ./delilah-tud-ssb
```

We will assume that all of the files located in the data directory are placed

on the Daisy's SSD. In our setup, They are located in the following folder:

/run/media/nvme0n1p1/tud

Depending on what experiments are enabled in the main function of src/ssb.cpp, you should see the corresponding performance numbers. For example, if you run the filtering application with fully offloaded operators on scale-factor 10, you will see the following output:

```
SSB 3.3 with Cache, offloaded Filter, Conversion and Aggregation, Bitmask, SF = 10 [JIT]
4964570us
```

Before we describe the different ways filtering is offloaded to DAPHNE, let us describe how DAPHNE has evolved since D6.2.

Delilah Prototype

Compared to D6.2 (and to the DaMoN paper [1]), we have modified the prototype in the following ways:

- Just-in-time (JIT) compilation of eBPF programs. Delilah can JIT compile an EBPF program or evaluate it (as in the original version).
- File access: We experimented with different ways to access data stored on SSD from Delilah. The key insight is that it is not possible to transfer data from SSD to the host via Delilah without a memory copy. The memory copy might be performed implicitly if we access data via a file system buffer, or explicitly if we bypass the file system buffer (with O_DIRECT). Indeed, it is not possible to transfer data directly from an SSD to the region of memory that is shared between Delilah and the host. This is because memory

regions associated to the SSD and shared with the host must be defined in incompatible ways in the device tree.

- Caching: Early experiments with filtering showed many configurations where it is beneficial to buffer data read from SSD across multiple offloaded functions. As a result, we cache the data read from the SSD and reuse it across offloaded eBPF function executions. Consequently, eBPF programs have two pointers as arguments: one to the context shared with the host for input and output, and one to the buffer where data obtained from the SSD are stored. This need for caching offsets the constraint of a memory copy described above.

We have an initial implementation of an hardware accelerated filtering function. At this point, this hardware-accelerated function is not fully integrated with Delilah. But this initial work gives us the perspective of an integration of some of the results from WP7 with the computational storage solution developed in WP6.

Performance Evaluation

We consider Delilah (a) with or without JIT compilation, (b) with caching (data is reused) or without explicit caching (data is repeatedly accessed) and (c) with comparison-based or bitmask based filtering. The workload is based on the Star Schema Benchmark (SSB) at scale 1 (24 MB per file), scale 10 (240 MB per file) and scale 21 (504 MB per file). We consider data at scale factor 1, 10 or 21, and we execute query 3.3:

```
SELECT
   C_CITY,
   S_CITY,
   toYear(LO_ORDERDATE) AS year,
   sum(LO_REVENUE) AS revenue
FROM lineorder_flat
WHERE (C_CITY = 'UNITED KI1' OR C_CITY = 'UNITED KI5') AND (S_CITY = 'UNITED KI1' OR S_CITY = 'UNITED KI5') AND year >= 1992 AND year <= 1997
GROUP BY
   C_CITY,
   S_CITY,
   year
ORDER BY
   year ASC,
   revenue DESC;
```

The table lineorder is stored columnswise in 4 files (1 for each attribute involved in the query).The execution of this query is decomposed into four sequential steps:

(1) Read the files

(2) Filter tuple ids that match the where clause

(3) Convert data from string to integer representation (toYear function)

(4) Aggregate by city by year

For this experiment, we code the four steps in C and compile them to eBPF bytecode. These eBPF functions can be executed either on the host or on computational storage. Figure 2 compares the following configurations:

- *Query3 3 cached agg on host VM* – Step1 on computational storage, steps 2-4 on host in interpreted mode at scale factor 1
- *Query3 3 JIT* - Offloads all operators in JIT mode without caching at scale factor 1
- Query3 3 cached 10 bitmask unrolled - Offloads all operators in JIT mode with caching *and bitmask unrolled filtering at scale 10*
- *Query3 3 cached agg on host JIT* - Step1 on computational storage, steps 2-4 on host in JIT mode at scale factor 1
- *Query3 3 cached 21* JIT - Step1 on computational storage, steps 2-4 on host in JIT mode at scale factor 21
- *Query3 3 VM* - Offloads all operators in interpreted mode at scale factor 1
- *Query3 3 cached VM* - Offloads all operators in interpreted mode with caching at scale factor 1
- *Query3 3 cached JIT* - Offloads all operators in JIT mode with caching at scale factor 1
- *Query3 3 cached 10 JIT* - Offloads all operators in JIT mode with caching at scale factor 10
- *Query3 3 cached 21 bitmask* - Offloads all operators in JIT mode with caching *and bitmask filtering at scale 21*
- *Query3 3 cached 10 bitmask* - Offloads all operators in JIT mode with caching *and bitmask filtering at scale 10*

At scale factor 1, we observe that execution in JIT mode is much more efficient than in interpreted mode, both on the host and on Delilah. We only consider execution in JIT mode at scale factors 10 and 21.

At scale factor 1, we observe that execution in JIT mode with caching on Delilah is comparable to execution on the host. This shows that code offload on Delilah can reduce CPU utilization on the host without performance penalty. We only consider execution with caching at scale 10 and 21.

At scale 10 and 21, we compare different ways to execute filtering on Delilah. At scale 10, there is little difference between the different implementations. At scale 21, bitmask filtering performs much better than tuple at a time comparison.

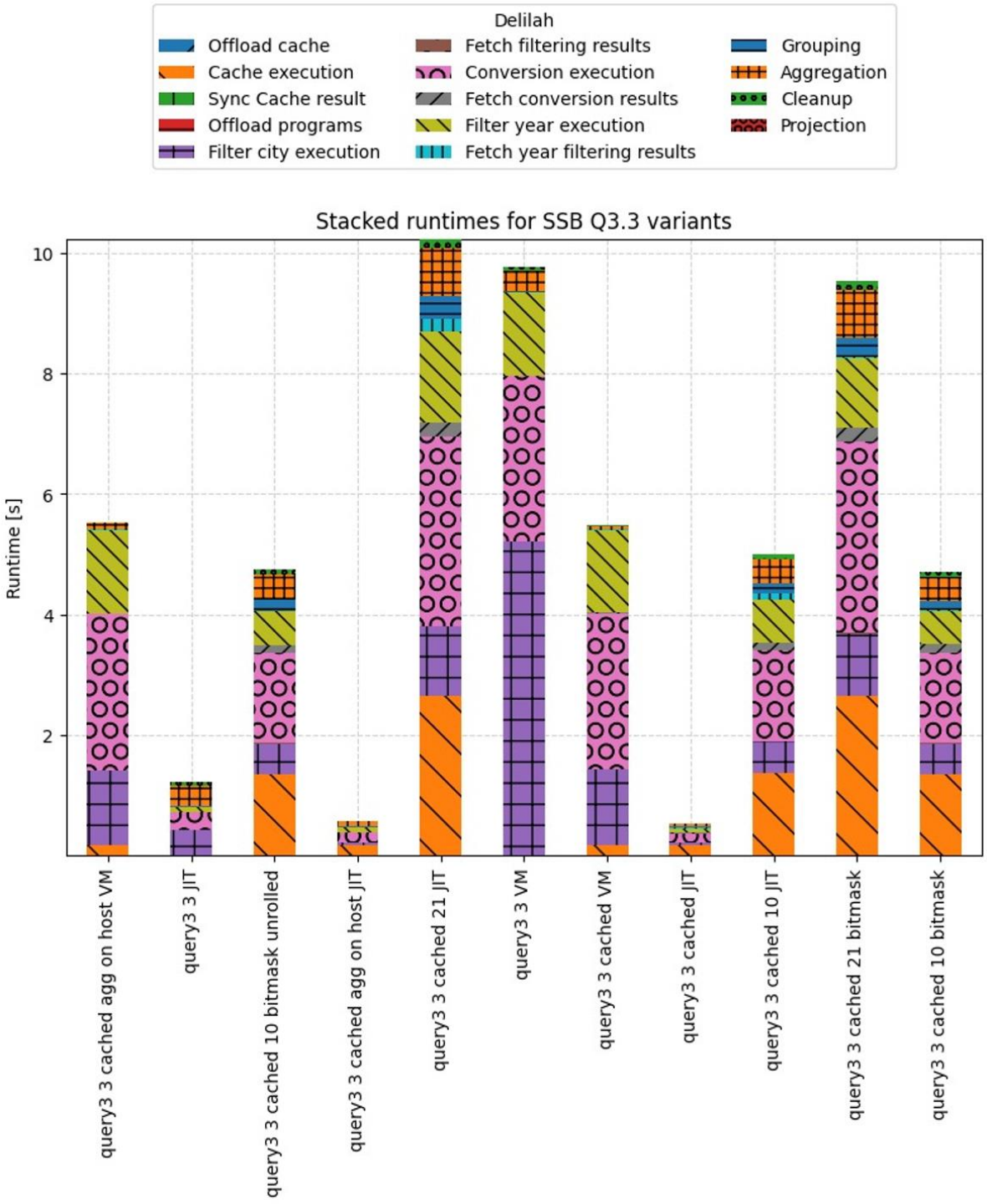


Figure 2: Comparing different eBPF code offload configurations

Conclusion

The second version of the Delilah demonstrator described in this report is used to execute portions of a relational query on computational storage. The report detailed asynchronous I/Os and chunked I/Os as the key building blocks for data path optimization and automatic data placement in DAPHNE. The next deliverable will integrate the relevant building blocks into an end-to-end computational storage components.

References

[1] Niclas Hedam, Morten Tychsen Clausen, Philippe Bonnet, Sangjin Lee, and Ken Friis Larsen. 2023. "Delilah: eBPF-offload on Computational Storage". In Proceedings of the 19th International Workshop on Data Management on New Hardware (DaMoN '23). Association for Computing Machinery, New York, NY, USA, 70–76. https://doi.org/10.1145/3592980.3595319

[2] Jens Axboe,. "What's New with Io_uring?" 2022. https://kernel.dk/axboe-kr2022.pdf.

[3] Simon Lund., Philippe Bonnet, Klaus B. A. Jensen, and Javier Gonzalez. 2022. "I/O Interface Independence with xNVMe." In *Proceedings of the 15th ACM International Conference on Systems and Storage*, 108–19. SYSTOR '22. New York, NY, USA: Association for Computing Machinery.

[4] Dimitrios Tsoumakos, Stratos Psomadakis, Constantinos Bitsakos, Aristotelis Vontzalidis, Mark Dokter, Patrick Damme. Improved DSL Runtime Prototype and Overview. DAPHNE Deliverable 4.3. November 2023.