Near-Storage Processing in FaaS with *Funclets*

Alan Nair
The University of Edinburgh
Edinburgh, UK
alan.nair@ed.ac.uk

Raven Szewczyk
The University of Edinburgh
Edinburgh, UK
raven.szewczyk@ed.ac.uk

Donald Jennings
The University of Edinburgh
Edinburgh, UK
d.d.jennings@outlook.com

Antonio Barbalace
The University of Edinburgh
Edinburgh, UK
antonio.barbalace@ed.ac.uk

Abstract

Serverless computing has disrupted how computation is performed in the Cloud. The ability to write Functions, and not care about infrastructure brings many benefits, including significantly lower deployment costs, improved developer workflow, scalability, resilience, and resource utilization. However, being stateless and not tied to specific machines, Functions need to access Cloud storage services to access data, which may require crossing the entire data center network incurring high overheads. Existing solutions either provide database APIs running on the storage servers to perform the data-intensive operations locally, or deploy entire Functions on storage servers. The former approach can not perform arbitrary computations locally on storage servers. The latter violates the principle of compute-storage disaggregation, resulting in poor scaling. We observe that allowing on-the-fly migration of I/O-intensive parts of Functions to storage nodes achieves both objectives. We propose a FaaS runtime that runs on both the compute servers *and* the servers running the storage services which introduces an efficient migration mechanism for Functions across machines to move I/O-intensive parts of Functions to the relevant storage node, with minimal code changes. This allows Functions to perform arbitrary computations on storage nodes, benefiting from the locality of data, without sacrificing the scalability offered by compute-storage disaggregation. We implement our approach in a state-of-the-art FaaS runtime and show that it improves latency and throughput in bandwidth-constrained FaaS workloads while making better utilization of idle CPU cycles on storage servers.

CCS Concepts

• **Computer systems organization → Cloud computing**.

Keywords

Serverless, Function-as-a-Service, Cloud Computing

ACM Reference Format:
Alan Nair, Raven Szewczyk, Donald Jennings, and Antonio Barbalace. 2024. Near-Storage Processing in FaaS with *Funclets*. In *25th International Middleware Conference (MIDDLEWARE '24), December 2–6, 2024, Hong Kong, Hong Kong*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3652892.3700755

1 Introduction

Serverless computing has disrupted the way applications are deployed in the Cloud. In particular, the Function-as-a-Service (FaaS) paradigm is increasingly gaining popularity. In FaaS, developers deploy their applications as a collection of event-driven stateless code-units called 'Functions', without caring about the infrastructure they are deployed on. Independently of the specific FaaS offering, a Function is a program – a collection of functions[1] written in a programming language supported by a FaaS runtime system. The Cloud Service Provider fully manages the infrastructure, and users are billed as per the resources they use, which scales up and down elastically based on demand (i.e., giving the illusion of infinite available resources). Hence, from the user's point of view, FaaS reduces costs, improves scalability, and improves developer workflow, while also providing resilience, security, and performance guarantees as maintenance and scheduling are taken over by the Cloud Service Provider. The FaaS offering is plenty and diverse, which includes big hyper-scalers like AWS Lambda, Azure Functions, Google Cloud Functions, Huawei Cloud Functions, and Cloudflare Workers, as well as smaller players based on open-source software.

A key enabler of the elastic scaling offered by the FaaS model is the statelessness of Functions, which facilitates the Cloud Service Providers to disaggregate compute from storage. Hyperscalers build their data centers with disaggregated compute and storage nodes, because this provides several advantages, such as independent scaling and technology upgrades. Functions, which run into a FaaS runtime, are deployed exclusively on compute nodes. When Functions need to access Cloud storage, they communicate across the data center network to interact with the storage stack. Often the result of a stateful computation would be just a fraction of the total data moved. Thus, such communication adds heavy bandwidth and latency overheads to the FaaS processing [23].

Lesson from the Database Community. A similar problem has been observed and solved in the context of distributed databases

[1]To avoid confusion we will refer to FaaS Functions as Functions (capitalized) and the programming language concept with the same name as functions (lower case).

in data centers by offering limited APIs to execute standard database queries on the storage nodes. For example, AWS added the *S3Select* functionality to its S3 object storage [4], which enables the execution of limited SQL queries on storage nodes for JSON and CSV data. This approach solves the problem for a given domain but does not allow the user to perform arbitrary stateful computations on the storage nodes.

Approaches adopted in Prior Work. Existing solutions have proposed co-locating the stored state with the Functions [26, 34, 41]. While these approaches solve the issue related to the overheads of data transfer, they have limitations. **First**, separating the I/O-intensive parts of a Function and the non-I/O-intensive parts into two separate Functions is not always advisable, as it may contradict the requirements of the application logic, and add redundant operations to the Function's workflow. **Second**, deploying full Functions on storage servers can cause storage node CPUs to experience heavy load as they become a contested resource. As storage technologies get faster, I/O stops being the primary performance bottleneck on storage servers, and CPU cycles become a more precious resource. Since the data on storage nodes may also be used by external applications, storage requests sent by them will compete for finite CPU cycles with compute-heavy Function code, resulting in serious performance degradation. **Third**, this approach re-aggregates compute with storage. Consequently, Cloud Service Providers that adopt these approaches will suffer limitations on scalability and technology upgrades.

We observe that the problems with the existing solutions that co-locate compute and storage can be solved if we adhere to the following principles:

- Only the I/O-intensive part of a Function should be co-located with stored data, rather than an entire Function. This ensures that the CPU/memory load on storage servers from non-I/O code is minimal.
- The decision of whether to run Function code on the storage server should be made *dynamically* based on factors known at runtime, such as the current server load, the size of the data transaction, and the hotness of the accessed data object. This ensures that the principle of compute-storage disaggregation is not violated, as Functions are still deployed on conventional compute servers and data is still stored on conventional storage servers. The only difference from the conventional scenario is that now the users and Cloud Service Providers have the option of executing stateful Function code by shipping *code to data* rather than *data to code.* Whether to take the former route or the latter route is to be decided per-invocation.

Introducing the Funclet. In this paper, *we propose an architecture to **offload part(s) of FaaS Functions** into storage nodes in a disaggregated data center*, which applies to Functions written in any programming language – thus, generic, instead of domain-specific. The developer will mark sections of Function code for offloading with a very simple markup. We call these code segments "Funclets" which are I/O-intensive blocks of code. The Cloud Service Provider can then decide, on a per-invocation basis in a dynamic fashion, whether to offload any given Funclet. In contrast to Functions, Funclets are smaller, I/O-bound mostly, and deployed on-the-fly. This makes Funclets more effective at utilizing the untapped CPU cycles on storage nodes than entire Functions.

We implemented a prototype of our architecture based on WebAssembly to support Functions written in different programming languages and heterogeneous CPUs, which we evaluated on multiple FaaS workloads using a realistic cluster with compute and storage servers. We show that it significantly improves latency and throughput in bandwidth-constrained FaaS workloads without constraining the computational resources of the storage server.

1.1 Contributions

Our **key contributions** are:

(1) We introduce an architecture to offload parts of FaaS Functions, called "Funclets", to storage nodes in data centers – practically, wherever the data is persisted. Consequently, Funclets become **the new smallest unit of code** that can be invoked remotely in the cloud – until now, it was Function.
(2) We implement the architecture in a state-of-the-art WebAssembly FaaS runtime, including new mechanisms to reduce the overheads of state migration.
(3) We evaluate our architecture with microbenchmarks and real-world Functions first on a 3-server setup, and then on a realistic (small) data center cluster.

The rest of the paper is organized as follows. In section 2 we describe the key technologies involved in our work. In section 3 we motivate our approach with a realistic example of the problem and a solution sketch. In section 4 we describe the high-level design of our work. In section 5 we explain the implementation details. In section 6 we present the results of our evaluation. We then discuss some related work in section 7.

2 Background

WebAssembly [16] (WASM). is a recent technology that is an evolution of previous attempts (asm.js [7], NaCl [27]) to allow for embedding high-performance code in the sandboxed and heterogeneous environment of web browsers. Additionally, it has been used in many other environments, such as plug-in systems for traditional desktop applications, and as a portable binary format for cloud providers. Cloudflare Workers allow WebAssembly modules to run on their serverless infrastructure [35] since 2018.

A WASM program typically consists of a module that contains or references external structures for its memory space, function pointer tables, and globals. These mutable structures change during the module's execution and must be captured during snapshots to enable the migration of live WASM processes.

Modern Datacenters. disaggregate compute from storage by running workloads on designated compute clusters and storing the data in large remote storage pools. These remote storage pools can store up to petabytes of data [11, 12, 17] and are managed by storage servers running storage services. This disaggregation allows datacenter service providers to reduce costs by using less powerful CPUs for storage servers and by employing distributed and replicated storage to increase availability and fault tolerance. Further, disaggregation allows applications to be deployed on any node in the compute cluster, without awareness of where the associated data resides. Applications on the compute cluster interact with storage via storage APIs.

3 Motivation

Consider a Function that contains an I/O-intensive operation, as illustrated in Figure 1a. The I/O-intensive operation operates on remotely stored data, which must be requested from the corresponding storage server before the operation can begin. The operation performed on the data is computationally simple, like substring-matching or vector multiplication followed by reduction. The result of the operation is typically much smaller in size than the data transferred. This scenario is typical of many real-world FaaS workloads, such as analytics and monitoring systems (DevOps). The data to be transferred can be up to hundreds of megabytes in size [23]. This results in high Function latency due to long network transfer delays, and high network bandwidth consumption. Further, Cloud Service Providers often throttle network bandwidth to ensure that no single Function hogs all the available bandwidth, worsening the situation.

Figure 1b illustrates our approach. The I/O-intensive operation is offloaded to the storage server, with a lightweight snapshot of the Function's memory state. The I/O-intensive computation is performed locally, and the snapshot is updated with the result. This updated snapshot is returned to the compute node where the Function's state is restored, and execution resumes. Since the overall data transferred over the network is much smaller than the data processed by the I/O-intensive code, the function completes more quickly and consumes fewer network resources throughout its lifetime.

Our solution targets workloads deployed within the FaaS model, including search operations, counting, statistics, serialization/deserialization, and other forms of data preprocessing. These operations consume significant I/O bandwidth in the system, often performing synchronous operations sequentially, which leads to unnecessary waiting times in programs. We aim to move these routines closer to the stored data transparently to the user, for example, by wrapping the existing I/O routines in a function call. Thus, existing software can be easily adapted to benefit from near-data processing without requiring extensive code rewrites for the target storage hardware architecture.

4 Design

Our approach to bringing compute closer to storage involves offloading storage-intensive parts of the Function code, called *Funclets*, to the storage nodes. The computations are performed locally on the stored data, and the result is returned to the source compute node to resume Function execution. Our solution, in the context of Function-as-a-Service in a disaggregated data center, is based on the following ***design principles***.

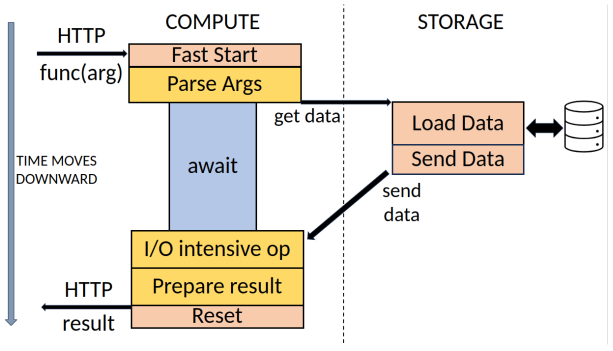
- **Practical:** Demarcation of a Funclet within a Function must be simple with minimal (if any) changes to the code of the application so that it is easy for developers to update their Functions to make use of our solution.
- **Generic:** The solution must not be tied to a single programming language or application domain.
- **Portable:** The scheme should operate on any hardware, since the storage and compute nodes may have different architectural specifications. An emphasis on portability also empowers us to take advantage of hardware accelerators such as smartNICs or smartSSDs.
- **Dynamic:** The decision of whether or not to offload a particular Funclet on a Function invocation can be taken at runtime based on considerations such as the current server load, the size of the data transaction, and the hotness of the data object.
- **Secure:** All the isolation and security guarantees provided by the FaaS runtime on the compute node must also be preserved on the storage nodes where an offloaded Funclet executes.
- **Lightweight:** The overheads imposed by our solution, such as the startup time of a migrated Function segment, the memory overhead of the runtime, and so on, should be minimal.

4.1 Overall Architecture

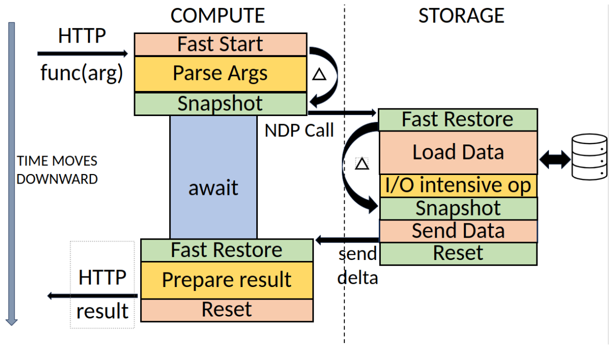
We propose an architecture in which FaaS runtimes run not just on compute nodes but also on storage nodes (Figure 2). Since we do not expect to run Functions that communicate with other services in the same or another data center, FaaS runtimes on storage nodes are limited in functionality in our architecture. Moreover, Functions' CPU utilization on storage nodes is monitored and capped, to avoid potential Denial of Services affecting existing services on the storage nodes. Although we primarily target the storage layer in disaggregated data centers, we believe that the same architecture may directly apply to any distributed store.

The FaaS runtimes on compute and storage nodes both communicate with the storage service by using the same storage API (e.g., object, block, or file). However, on the former, it is network communication, while in the latter it is intra-machine communication which offers higher bandwidth and lower latency, without consuming any network resources. A FaaS runtime includes an endpoint for Function invocation (such as API gateway in Amazon, or Ingress in Knative). A load balancer receives requests and submits them to a FaaS runtime running on a compute node. FaaS runtimes on storage nodes only accept requests coming from other FaaS runtimes, principally the ones running on compute nodes. Therefore, the in-storage FaaS runtime does not accept generic requests for Function execution, but only requests for Function offloading from compute nodes (cf. "runtime to runtime API" in Figure 2).

We do not alter the interfaces used to access data. They are executed on the storage nodes locally rather than via the usual RPC (over HTTP) interface. This means that our system does not modify the data consistency and replication schemes. It maintains all the invariants that were present in the existing storage layer. Replication does add the question of which node to run the code on, which is solved by the usual load-balancing strategies. Some forms of erasure coding might require the storage nodes to communicate on data read to exchange parity information in order to validate stored data, this process would remain unchanged in our solution.



(a) Convention FaaS workflow ships data to code



(b) Our work ships code to data

Figure 1: Comparing Conventional FaaS approach with Our approach

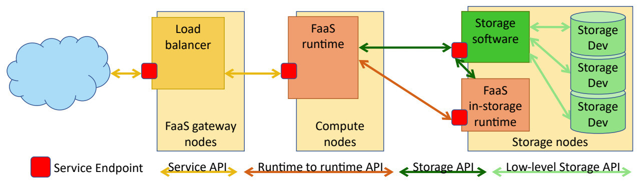


Figure 2: Overall architecture of our in-storage FaaS.

4.2 Offloading Architecture

Our architecture enables a Funclet to be sent along with its context to be executed on the in-storage FaaS runtime. This gives the FaaS developer the flexibility to choose part(s) of their Function as candidates for running on the storage nodes, rather than an entire Function which may include parts that are not storage-intensive. Further, the decision of whether to offload a particular Funclet is made dynamically on a per-invocation basis, enabling the Cloud Service Provider to adopt different Funclet-offloading policies based on factors known at runtime such as the current server load, the size of the data transaction, and the hotness of the data object.

This offloading architecture allows the FaaS developers to reap the benefits of near-storage processing, which include lower bandwidth costs and faster overall runtime which also improves the end-user experience. It also enables the Cloud Service Provider to streamline resource usage across compute and storage nodes, achieve effective load balancing during peak load times, and utilize unused CPU cycles on storage nodes. This brings down the service provider's TCO (Total Cost of Operation). Additionally, the configurability of the offloading ratio allows Cloud Service Providers to cater serverless offerings with a broader range of SLOs (Service Level Objectives), providing customers with more choices that tradeoff service cost for better performance guarantees.

5 Implementation

Data centers are no longer dominated by x86 CPUs [1, 2] and storage node vendors are introducing ARM-based servers and SmartNICs/SmartSSDs [6, 24, 28]. In view of this trend towards increasing heterogeneity of data center hardware, we base this work on WebAssembly [18], which is ISA-agnostic and supports multiple programming languages – instead of other solutions that are constrained to a single programming language, including [8, 10, 22].

We implemented our design atop the Faasm runtime [34]. We chose Faasm, as it is aligned with several of our design principles, being *generic*, *portable*, *secure*, and *lightweight*. Faasm uses the WAVM runtime [36] and includes various services and service endpoints in a single application. Our modifications comprise around 5000 lines of code, including bug fixes and performance tweaks, and are minimal to the core design of Faasm. Additionally, we submitted about 2000 lines of code as patches, which were integrated into Faasm. All our code is publicly available under an open-source license [2].

The Faasm runtime provides a mostly POSIX-compatible FaaS runtime via *wasi*. It uses WebAssembly "faaslets" as the unit of isolation, rather than system processes, VMs or containers. Proto-faaslets are also created, which are essentially pre-initialized WebAssembly modules. This allows for very fast startup and tear-down of Functions by simply cloning the proto-faaslet state. We use these proto-faaslets to provide a consistent view of the Function/Faaslet memory state at the compute and storage nodes (See Section 5.3). Faaslets from different users always run in isolated threads, allowing the operator to control relative scheduling windows according to the desired fairness policy. Faasm can run across different nodes, sharing the load between them automatically. In our work, communication between nodes happens over an NNG[19] network message passing layer, following Faasm's existing implementation with a few custom message types added.

5.1 Storage Access Interface and Backend

We employ Ceph [37] as our distributed replicated storage backend. Additionally, we enhance the Faasm runtime by introducing the concept of a storage node, enabling local file access as Ceph objects via WebAssembly and HTTP APIs. This includes WASI-compliant Key-Value Store APIs to access stored objects. The `get` function loads the stored object corresponding to the given key, into a memory buffer, while the `put` function stores a given key-value pair to the storage backend. These APIs have been implemented in C++ and

[2]https://github.com/auto-ndp/faasm

Python. Listing 1 shows an example of how the `get` and `put` functions from the C++ API are used (the corresponding functions are `__faasmndp_getMmap` and `__faasmndp_put` respectively). When running on a compute node, the requested file chunks are fetched from the storage node and kept in memory while file operations are in progress. On a storage node, operations are directly applied to an open file through the operating system.

Ceph uses RADOS [38] as a storage service. Each storage node runs an Object Storage Daemon (OSD) which manages data stored on that server and coordinates with other OSDs to orchestrate object placement. In the case of replicated storage, multiple replicas of an object are distributed across multiple OSDs, of which one is designated as the primary OSD which serves all requests to that object.

```

int global1;

int wasmFunc()
{
    ...
    uint8_t* data = __faasmndp_getMmap(
        keyPtr, keyLen, offset, bufsize, &fetchedLen);
    ...
    __faasmndp_put(
        keyPtr, keyLen, objPtr, objLen);
    ...
    global = result;
    return 0;
}

int main()
{
    ...
    if (__faasmndp_storageCallAndAwait(
        objPtr, objLen, wasmFunc
    ) != 0)
        return 1;
    ...
}
```

Listing 1: Funclet C++ API usage example. wasmFunc is a Funclet that accesses a stored object.

5.2 Funclet Offloading

Function invocations are performed via HTTP POST requests. We enabled a special invocation parameter as a form data object in the POST request, allowing the user to specify whether the invoked function should offload any of its enclosed funclets. This fulfills our design requirement of being *dynamic*, as it allows the user or Cloud Service Provider to decide at runtime, whether the Funclets should be offloaded. Cloud Service Providers can build their Function invocation APIs that wrap around our bare-bones invocation framework that uses HTTP POST requests. This way, the Cloud Service Providers can decide if and how to expose the Funclet-offloading capability.

We provide a new WASM system call that can be used as a wrapper around offloadable Funclet calls. This system call takes the function pointer of the offloadable Funclet and the key to the stored object as parameters. To adapt existing Function code for use with our work, the developer must replace the calls to offloadable code segments with our WASM system call. This aligns our work with the design principle of being *practical*.

In the C++ API, this function is called `__faasmndp_storageCallAndAwait`. Its usage is shown via example in Listing 1. The function to be offloaded cannot take any arguments and must return 0 (integer) on successful exit, or a non-zero integer on failure to indicate error type. Since the Function context is migrated along with the Funclet, global variables can pass data between the Funclet and the rest of the Function.

This WASM system call uses the key of the stored object accessed by the Funclet to identify the storage node which serves as the offloading target for the Funclet. If the object is replicated across multiple storage servers, then the one that runs the primary OSD for that object (see Section 5.1) is identified as the offloading target. A Funclet invocation request is created, with context and state information, including the Function name, Funclet pointer, Funclet arguments, and global variables. This request is then sent to the target storage node. At the storage node, the Funclet is started as a Faasm module (namely, Faaslet). The Funclet’s state is first initialized with the information contained in the request, and then it begins execution. When the Funclet terminates execution, the final state containing the result of the Funclet execution is captured. A request that encloses this final state is created and sent back to the source compute node. At the compute node, the received information is unpacked into the Function state, and the Function continues its post-Funclet execution. The sequence of events that happen during Funclet offloading is presented in Figure 1b.

5.3 Snapshotting

To efficiently transfer the state of Function execution between the machines for offloading, we extended Faasm by adding a mechanism to capture incremental memory snapshots. These snapshots are stored as a set of changes to a known memory state.

As a Function reaches the point in its lifetime when it is about to make a call to an offloaded Funclet, it checkpoints its state before making the request. First, it searches for an existing stored snapshot corresponding to its state. If such a snapshot does not exist, then it generates a fresh snapshot of its current memory state and stores it. This snapshot is also sent to the storage node which is the offloading target. If a snapshot from a previous invocation exists, then the difference in memory state between the snapshot and the current state, known as a *Delta*, is computed. This Delta will be sent to the storage node. At the storage node, the Funclet can recover the state of the Function at the compute node using the stored snapshot from the first invocation and the Delta. After the Funclet finishes execution, it computes the new Delta for its updated memory state which contains the result of its computation. This Delta is sent back to the compute node, where it is used to recover the updated memory state with the result of the Funclet’s execution.

Since it is important to keep snapshots and Deltas small, we eliminate from them large regions in the Function address space that are filled with zero-bytes, fill large memory allocations with zero-bytes when freed on the storage node, and ensure that the raw file data read into memory at the storage node gets freed before the updated Delta is captured. The snapshots are stored in a distributed manner as Ceph objects and are managed using Faasm’s proto-faaslet management system, while the Deltas are created and destroyed every invocation. Since Deltas are smaller than full snapshots, the use of Deltas enables bandwidth and latency savings on subsequent Funclet invocations. For instance, on the *wordcount* benchmark, we saw that the snapshot size was 16MB while the Delta size was under 4.7KB.

6 Evaluation

Herein we evaluate our work with real-world benchmarks to measure and quantify the different behaviors exhibited by our system. We first evaluate it on a small *test cluster* to highlight and closely study the trends that surface with offloading, and then evaluate it on a *realistic data center cluster*. We introduce our experimental workloads first.

6.1 Experimental Workloads

We selected a variety of workloads with different behaviors to study the effects of offloading on program and cluster performance. All the workloads were implemented in C++. All the associated datasets were stored as Ceph objects with 3x replication. Workloads are described below.

6.1.1 Text-Processing Benchmarks. We use two simple text-processing workloads with datasets of varying sizes to quantify the different properties of the system under varying loads. These functionalities are generic and barebones enough to be used extensively in real-world scenarios. The output of the text-processing is typically much smaller than the data accessed, which makes such workloads ideal candidates for offloading.

Substring-Finder. is a benchmark that takes a string and the key (name) to a stored text file object as parameters. It reads the text file, finds all occurrences of the substring in the file, and returns the indices of all positions where they occur. Its functionality is similar to the *substr* function of the `string` library in C++ if run iteratively till the end of the file.

Wordcount. is a benchmark that takes as a parameter the key to a stored text file, reads through it, and returns a vector of all the unique words present in it, along with the number of occurrences of each word.

These two benchmarks are evaluated on three datasets of varying sizes. The datasets were created from the Project Gutenberg E-books corpus [15], which contains over 29 GB of text files. We created three datasets - (a) *large* with 100 files of size 128 MB each, (b) *medium* with 1000 files of size 4 MB each, and (c) *small* with 1000 files of size 128 KB each. The file sizes were controlled by concatenating files together (to lengthen) and truncation (to shorten).

6.1.2 Special-Purpose Application Benchmarks. These benchmarks are based on real-world applications which we felt were good candidates for highlighting the behavior of our offloading framework.

PCA-KMM. This benchmark is a machine learning workload that performs dimensionality reduction with PCA (Principal Component Analysis), followed by *k-means* clustering. Machine Learning applications often do feature extraction with dimensionality reduction to reduce the size and improve the quality of the dataset for training and inference. This benchmark reduces the number of variables in the dataset by orders of magnitude, which makes it an interesting candidate for offloading to storage. We run this benchmark on the MNIST Handwritten digits dataset [13] which has 10, 000 image files with an average file size of 7.5 MB.

Damon-Filter. is based on *damo* [31], the user-space tool for DAMON [29]. DAMON is a data access monitoring tool that runs in kernel space. It can monitor the in-memory data access patterns of any user process in a lightweight manner. DAMON is now a part of the Linux kernel (since v5.15) and is an ideal tool for Cloud Service Providers to run on their storage servers to monitor data access patterns. This enables them to optimize data placement strategies on tiered memory systems. *damo* is a user-space tool that generates readable reports and heatmaps from the raw data generated by DAMON. DAMON assigns a hotness score to each memory region based on their access frequencies. Our benchmark takes as parameters the key (name) to the stored raw DAMON data object, and an integer as a hotness threshold. It returns the total amount of data whose hotness score exceeds this threshold. We use raw DAMON data generated from monitoring the MASIM (Memory Access Simulator) [30] workload. This comprises 100 files of around 5 MB each. *This benchmark was tested only on the realistic data center cluster*, and not on the test cluster.

Thumbnail-Generator. reads and decodes a PNG image file, storing the raw pixel data into an array of 8-bit RGBA values. It then generates a shrunken version of the image, by taking averages of color values of neighboring pixels to finally output a PNG file containing the thumbnail. This benchmark runs on the Large Logo Dataset [32] whose size is 13 GB. *We expect this application to perform poorly* since the offloaded operation is decompression which inflates the amount of data sent over the network and also results in larger snapshots. This is an example of a workload that is better *NOT* to offload.

6.2 Evaluation on Test Cluster

We evaluate the above benchmarks on a small test cluster to perform a detailed analysis of our runtime's behaviors in a well-controlled environment. The purpose of this evaluation is to observe how the workloads from Section 6.1 behave under offloading. We track various metrics such as Function latency, throughput, the overheads of offloading and data transfer, and the sensitivity to the fraction (p) of offloaded Funclets.

6.2.1 Experimental Setup. Our experimental setup comprises of three servers: a compute node, a storage node, and a load generator. Their hardware configuration is given in Tables 1a, 1b, and 1c respectively. Our offloading framework is deployed on the storage and the compute node. All the servers communicate over a dedicated 10 Gbit/s network.

The load generator generates a configurable number of Function invocations per second on the compute node at a steady rate and reports the latency details of each request from initialization to response or timeout. Under very high loads the real rate of requests per second becomes lower than requested because the compute node can no longer handle new connections, so they time out.

We ran experiments for different values of the ratio of Funclets offloaded. We denote the ratio of funclets offloaded in an experiment as p. We also ran the experiments where the Functions execute only on the storage node, with the compute node not involved at all. These runs are marked as “AoS” (All on Storage) on the graphs – i.e.,

Table 1: Configuration of the test cluster.

(a) Compute node

Make	Supermicro X11DPI-N		
Model	Intel(R) Xeon(R) Gold 6230R		
# Procs	1	**Freq.**	2.1 GHz
# Cores	26	**# Thrds**	SMT off
Mem.	64 GiB	**L1$**	1.625 MiB
L2$	26 MiB	**L3$**	35.75 MiB
OS	Ubuntu 20.04, Linux 5.14.0		
Storage	480GB NVMe SSD		

(b) Storage node

Make	Dell PowerEdge R7425		
Model	AMD EPYC 7501		
# Procs	1	**Freq.**	2.1 GHz
# Cores	12	**# Thrds**	SMT off
Mem.	64 GiB	**L1$**	2 MiB
L2$	8 MiB	**L3$**	32 MiB
OS	Debian 11, Linux 5.14.0		
Storage	1TB NVMe SSD		

(c) Load generator

Make	Dell PowerEdge R440		
Model	Intel Xeon Silver 4110		
# Procs	1	**Freq.**	2.10 GHz
# Cores	8	**# Thrds**	16
Mem.	64 GB	**L1$**	64 kB
L2$	1 MB	**L3$**	11 MB
OS	Debian 11, Linux 5.10.0		
Storage	1 TB SATA HDD		

what have been proposed before in the literature. This is different from the $p = 1$ case, where for all Function invocations only the code parts marked as offloadable run on the storage node, the rest of each function runs on the compute node.

6.2.2 Results. We report the graphs to illustrate how the various workloads perform under offloading. Figures 3a to 6c show the results for Wordcount, Substring-Finder, Thumbnail-Generator, and PCA-KMM respectively.

In each of these figures, sub-figure *a* shows the relationship between the observed throughput and median latency, sub-figure *b* compares the median latency observed at low throughputs, and sub-figure *c* plots the maximum observed throughput for different ratios of offloaded Funclets (p) corresponding to the given benchmark. Here *observed throughput* refers to the rate at which requests are processed by the system.

Throughput vs Latency. In Figures 3a to 6a, we see how the median Function latency varies with the throughput (requests per second) achieved by the system. At low throughput, the median latency stays within a narrow range. This trend holds until throughput rises to a certain "knee" point, after which the median latency rises sharply for even a small increase in throughput. This happens because at low throughput, the inter-request-arrival-time, or the time between subsequent requests sent from the load generator, is greater than the Function latency. Hence there is no queueing, and the compute node begins to process a Function invocation as soon as it is received. But beyond the "knee" point, the inter-request-arrival-time becomes lower than the Function latency, therefore when a request is received on the compute node, it must first wait in a queue for the previous requests to finish. At high throughputs, queueing dominates the overall latency.

Modern datacenters typically deploy workloads with some Quality of Service (QoS) guarantees. The QoS guarantees typically involve an upper bound on median or tail request latency. Datacenter service providers try to maximize throughput without violating the established QoS guarantees. The QoS tail latency bounds are typically defined such that the latency-vs-throughput graph stays on the left side of the "knee" point, where queueing does not dominate Function latency. For the workloads being evaluated, the metric of interest is the maximum throughput achievable before the "knee" point. This is the maximum *sustainable* throughput for the workload on the system.

For the Thumbnail Generator (Figure 5a) workload, the maximum sustainable throughput is achieved when $p = 0$, or when there is no offloading. This is because the data decompression performed by the workload makes it a poor candidate for near-storage processing. For the Substring-Matching (Figure 4a) workload, the maximum sustainable throughput is achieved when the whole Function is deployed on the storage node, that is the *Always On Storage (AoS)* case, indicating that it is bound by the overheads of the offloading mechanism. In the Word Frequency (Figure 3a) and PCA-KMM (Figure 6a) workloads, the maximum sustainable throughput is observed at $p = 0.33$, indicating the benefits of dynamic offloading enabled by our work.

Table 2: Latency breakdown of a Function invocation

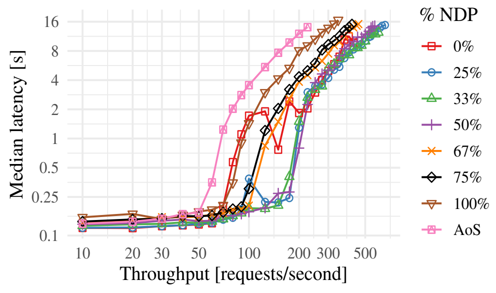
Task	Latency
Compute node startup time	550 μs
Preparing snapshot to send to storage node	**1.7 ms**
Network delays	1.3 ms
Storage node startup time from snapshot	**2.5 ms**
Reading file into WASM memory	600 μs
Executing WASM on the storage node	7 ms
Preparing snapshot to send to compute node	**1.2 ms**
Sending snapshot back to the compute node	4 ms
Restoring from the snapshot	**650 μs**
Executing WASM on the compute node	10 ms

Median Latency at Low Throughput. Figures 3b to 6b show the median latency observed at low throughput - when the overheads due to queueing are negligible. For the Thumbnail Generator (Figure 5b) the median latency increases with more offloading. For the Substring-Matching benchmark (Figure 4b), median latency decreases with more Funclets offloaded. For both, the *AoS* case has the lowest latency as it benefits from near-storage processing while avoiding the overheads of our offloading mechanism. This is not true for Word Frequency (Figure 3b) and PCA-KMM (Figure 6b). Their best latencies were not seen in the *AoS* case but for $p = 0$ and $p = 0.25$ respectively.

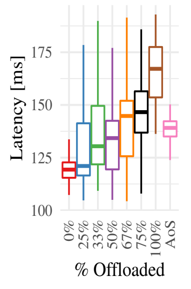
To investigate the additional latency in the system, we use precise performance monitoring tools to break down the latency cost into detailed components. We present as an example the breakdown for one invocation of the *wordcount* benchmark in Table 2.

In total, for a 30ms invocation, the overhead time of our migration mechanism (bold times) was 6ms, 20% of this very short-lived Function execution time.

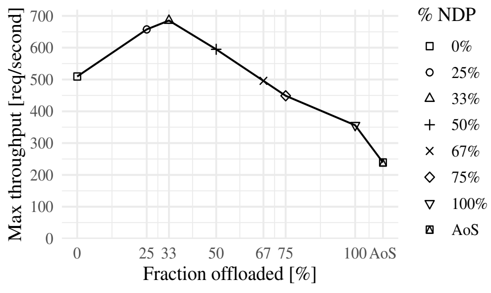
Maximum Achievable Throughput. In figures 3c to 6c we see the maximum throughput achieved on the system. For Substring-Matching (Figure 4c) and Thumbnail Generator (Figure 5c), this corresponds to the *AoS* and $p = 0$ cases respectively, while for Word Frequency (Figure 3c) and PCA-KMM (Figure 6c) this corresponds to the $p = 0.33$ case.



(a) Relationship between Throughput and Latency.

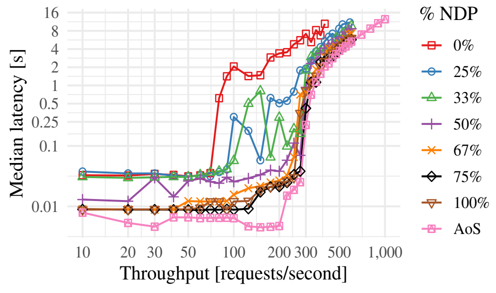


(b) Overhead of offloading.

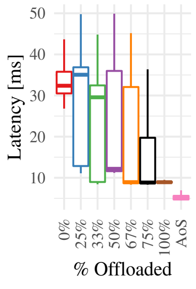


(c) Relationship between p and Max Throughput.

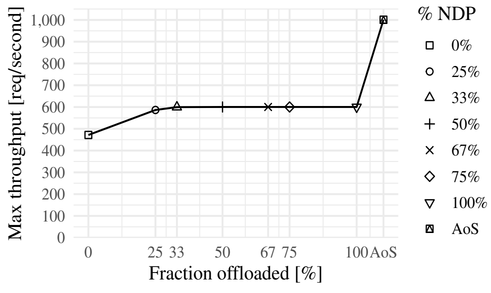
Figure 3: Results for the Word Frequency benchmark



(a) Relationship between Throughput and Latency.

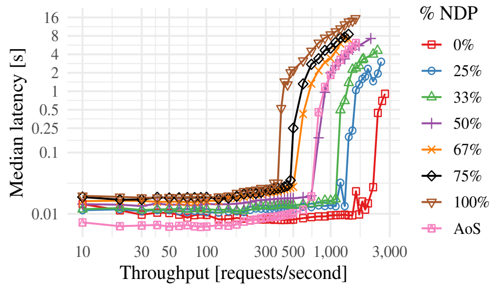


(b) Overhead of offloading.

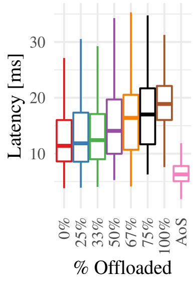


(c) Relationship between p and Max Throughput.

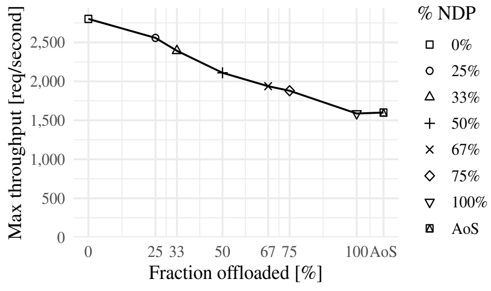
Figure 4: Results for the Substring Match Finder benchmark.



(a) Relationship between Throughput and Latency.

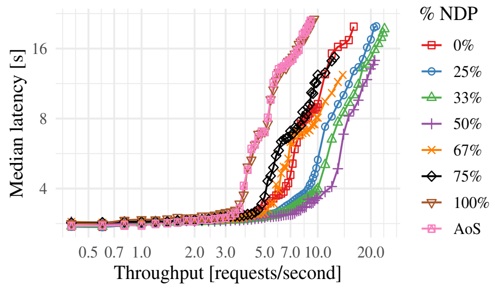


(b) Overhead of offloading.

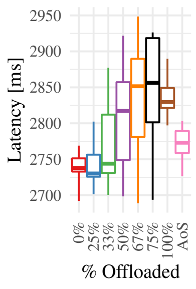


(c) Relationship between p and Max Throughput.

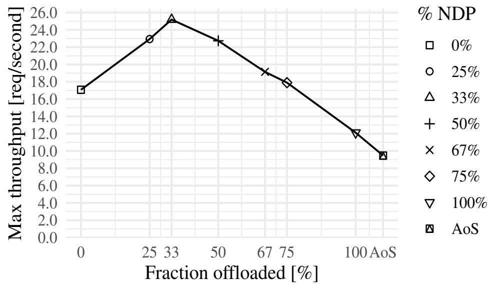
Figure 5: Results for Image Thumbnail Generator



(a) Relationship between Throughput and Latency.



(b) Overhead of offloading.



(c) Relationship between p and Max Throughput.

Figure 6: Results for PCA-KMM

Table 3: All servers in the realistic datacenter setup have this configuration.

Make	Dell R6525	**Model**	AMD EPYC 7543
# Cores	2x32	**# Freq.**	2.8GHz
Memory	256 GB ECC (16x16 GB 3200MHz DDR4)		
L1i$	512 KB	**L1d$**	512 KB
L2$	8 MB	**L3$**	128 MB
OS	Ubuntu 20.04, Linux 5.14.0		
Disk	1x480GB SATA SSD		
NIC	Dual-port Mellanox ConnectX-6 100 Gb NIC		

Table 4: Configuration of the realistic data center cluster.

Compute	64 cores online, SMT *on*	**# Nodes**	2
Storage	8 cores online, SMT *off*	**# Nodes**	5
Loadgen	64 cores online, SMT *on*	**# Nodes**	1

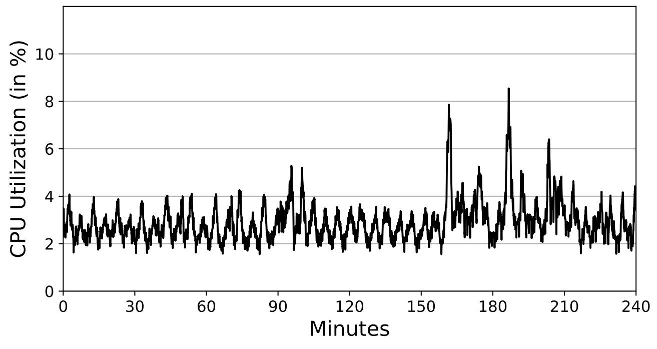
6.3 Evaluation on a Realistic Data Center Cluster

Now we evaluate our work on a setup that more closely resembles a realistic data center cluster. Here we study how well Funclets can utilize spare resources across a real cluster, and the impact of offloading on the Function latency.

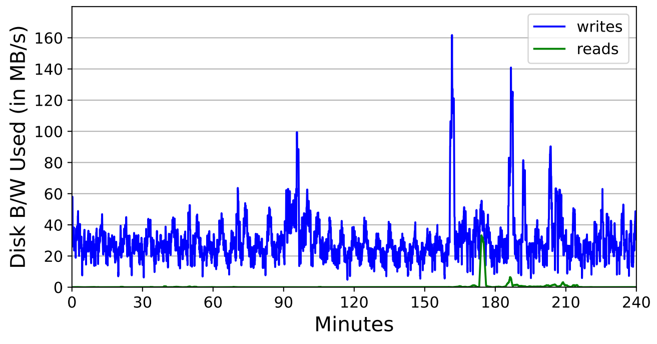
6.3.1 Experimental setup. Our experimental setup comprises 8 identical servers from the Clemson cluster of Cloudlab [14]. The configuration of each server is outlined in Table 4. We use 5 storage nodes, 2 compute nodes, and 1 load-generator for a realistic cloud cluster setup. We disable hyper-threading and keep just 8 cores online on storage nodes to align their computing power with that of the most low-end commercial storage servers [20, 21]. The nodes are connected to each other with a 100 Gbps NIC, but network speed tests using the `iperf` tool reported an average bandwidth of 33 Gbits/sec.

We expect the data stored on storage servers in data centers to serve requests from external clients as well, and not just the ones generated by our framework. We use block-level I/O traces of I/O requests collected Alibaba Cloud [25] to account for the additional load incurred due to these external requests. To account for the additional load incurred due to these external requests, we create 200 GB of synthetic Ceph objects to simulate stored data on each of the five storage nodes, each replicated 5 times. We use block-level I/O traces of I/O requests collected Alibaba Cloud [25] to simulate external accesses to this data. The accesses in the trace are mapped to the Ceph objects in such a way that all the storage servers experience a uniform load. This allows us to perform our experiments with offloading in the backdrop of realistic data-center resource usage.

We show CPU (Figure 7a), and disk usage (Figure 7b) on storage nodes running these traces over a period of four hours. The memory usage hovered around 60 GB. Note that even with just 8 CPUs, the CPU utilization rarely exceeds 8%. This is in line with findings from prior studies [40] and provides scope for our work which aims to utilize idle CPU cycles on storage servers by moving some



(a) CPU Utilization



(b) Disk Bandwidth

Figure 7: Average Resource Utilization on storage nodes with just the Alibaba Cloud block traces running.

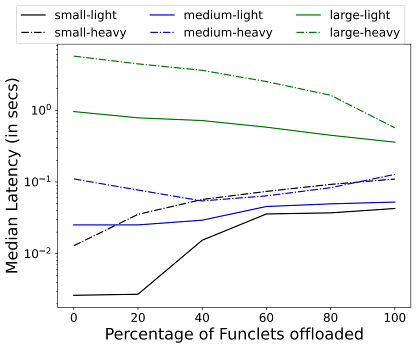
of the computing tasks from the compute nodes to the storage nodes. We do not account for network bandwidth because typically data center servers have separate NICs to handle connections from inside and outside the cluster. All requests from within the cluster are generated by our offloading framework, while the block traces simulate requests coming from outside the cluster.

All experiments with our offloading framework were conducted with these traces running in the background. Timeout was set at 30 seconds for each request. We monitor the time taken by each request to complete, as well as CPU usage, memory usage, network bandwidth usage, and disk bandwidth usage on all nodes. The monitoring software is lightweight and does not contribute much to the observed values of these metrics.

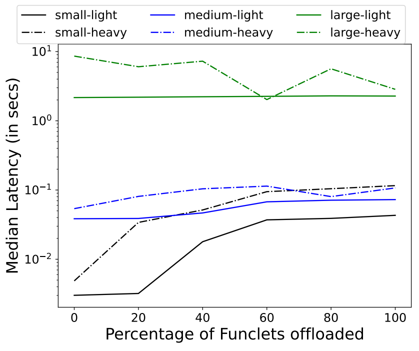
6.3.2 Results. Here we present the results related to Function latency and cross-cluster resource utilization from our experiments.

Latency. For each benchmark, we generate requests from the load generator node to the compute nodes. If a request is marked as offloadable, then its Funclets will be offloaded to the relevant storage node for near-storage execution. We run batches of 1000 requests with different values for the percentage of offloadable requests - 0% (no offloading), 20%, 40%, 60%, 80%, and 100% (all Funclets offloaded). We run each batch once with *light* load at 10 parallel requests and once with *heavy* load at 50 parallel requests.

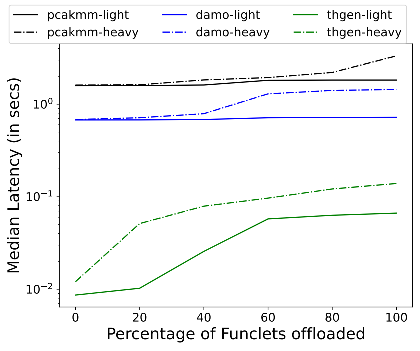
The median latencies are shown in Figure 8. In the results for Substring-Finder, we see that offloading is not preferable for the



(a) Substring-Finder

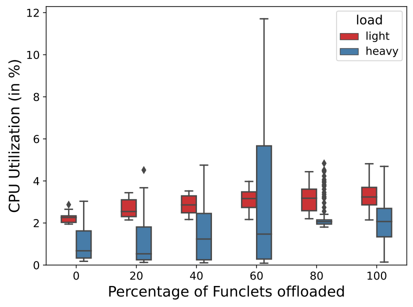


(b) Wordcount

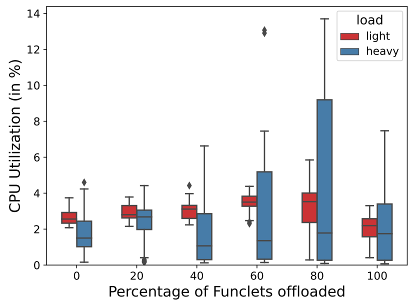


(c) Application Benchmarks

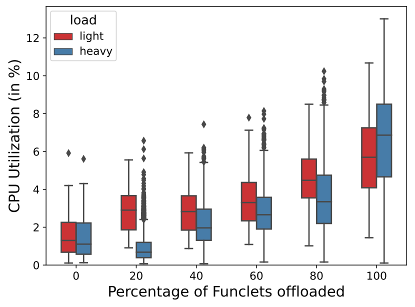
Figure 8: Median Latencies for different offloading ratios.



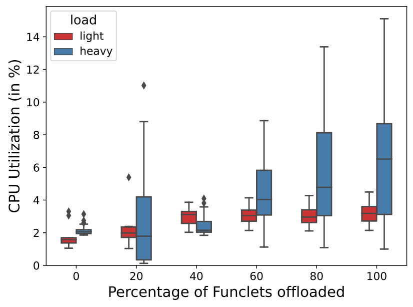
(a) Substring-Finder (Small)



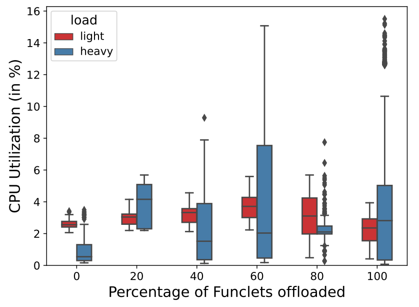
(b) Substring-Finder (Medium)



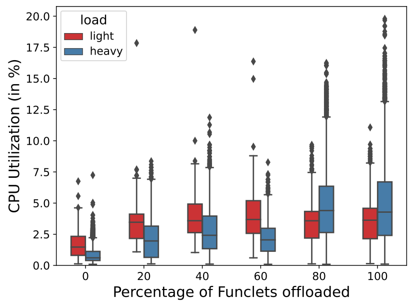
(c) Substring-Finder (Large)



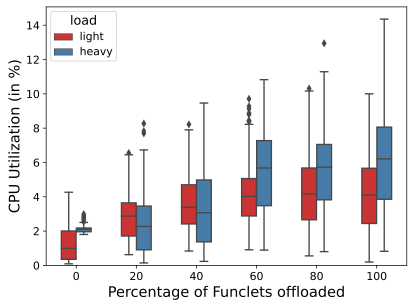
(d) Wordcount (Small)



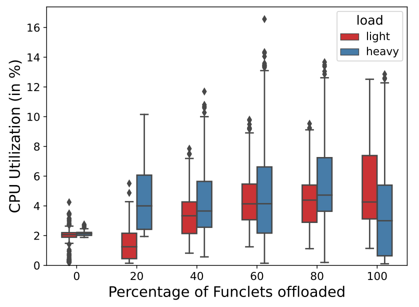
(e) Wordcount (Medium)



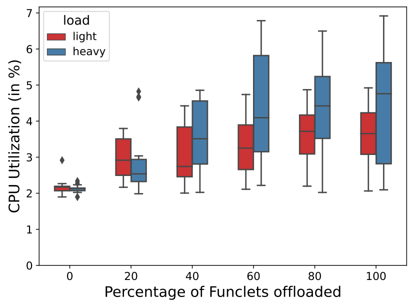
(f) Wordcount (Large)



(g) PCAKMM



(h) Damo-Filter



(i) Thumbnail-Generator

Figure 9: CPU Utilization of different benchmarks on the storage nodes.

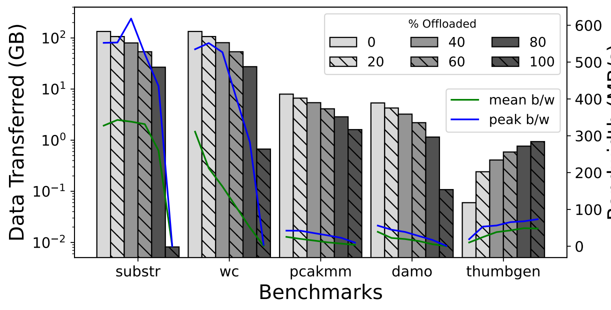


Figure 10: Bandwidth Utilization on the Compute Nodes.

small dataset as the median latency gets progressively worse with an increase in the ratio of offloaded requests. In contrast, offloading is preferable for the *large* dataset because the median latency decreases as the ratio of offloaded requests increases. For the *heavy* load configuration, we find that offloading all Funclets provides a 10x improvement in median latency while for the *light* load configuration, the improvement is 2.67x. With the *medium* dataset, the median latency is the lowest for an offload ratio of 20% when the load is *light*, while it is lowest for an offload ratio of 40% when the load is *heavy*. Similarly, for the Wordcount benchmark, with the *large* dataset on *heavy* load, the median latency is minimum for an offloading ratio of 60%. In all other cases, offloading Funclets to the storage nodes is not beneficial. The existence of data points where the lowest median latency is observed when the offloading ratio is neither 0% (no offloading or conventional execution) nor 100% (full offloading or "always-on-storage" execution) underlines the need for a flexible offloading framework that allows the offloading ratio to be tuned, which is what our runtime fulfills.

Two trends are to be highlighted. First, offloading becomes preferable, i.e., results in lower median latency, as the size of the accessed data increases. Second, offloading becomes preferable when the number of parallel requests increases. As the size of the accessed data increases, more bandwidth is needed to serve the non-offloadable requests. When bandwidth is the bottleneck, offloading becomes the preferred option.

Resource Utilization. Figure 9 shows the CPU utilization observed on the storage nodes for all the workloads. Even with offloading, the CPU utilization was never seen exceeding 20%. We also show the bandwidth usage for the benchmarks in Figure 10. For brevity, we show only the results of the *heavy* load configuration for all benchmarks, and for the text processing benchmarks, we show only the results for the *large* dataset because bandwidth becomes more significant as the data transactions increase in size. We see that there is a drastic drop in bandwidth usage as we offload more Funclets. The only exception to this trend is *Thumbnail-Generator* because its Funclet performs decompression on the storage nodes and ends up transferring more data over the network when offloaded, compared to conventional execution.

6.4 Discussion

We have looked at the behavior of different Functions when parts have been offloaded to the storage node for various fractions of requests handled by the cluster. Despite some Functions being more amenable than others to having parts offloaded to the storage, *the key takeaway* from the experiments on both the test cluster and the realistic datacenter, is that runtime control of the fraction of Functions p offloaded is crucial for good performance. Demarcating Funclets is easy in our system, and by carefully controlling the fraction of invocations with Funclet offloading enabled, datacenter service providers can achieve equal or better performance and better cross-cluster resource utilization on most Functions, compared to the conventional FaaS paradigm (where data is transferred to the compute node), or to approaches which deploy entire Functions on the storage servers. Our design achieves these benefits without violating the principle of compute-storage disaggregation, thus preserving the elastic scaling promised by the FaaS paradigm.

7 Related Work

Faasm [34] exploits shared memory when Functions can be scheduled on a single node, to efficiently share state over the network with its two-tier memory architecture. Our work extends Faasm by allowing on-the-fly code migration.

Shredder [41] enables storage services to run full Functions, however it focuses on in-memory stores, and code must explicitly send marshaled data along with the request to share state. LambdaObjects [26] integrates stored data with the methods that operate on them. Data and the associated methods are encapsulated into objects, which are deployed as a whole on the storage server. The Zion [33] framework brings compute closer to storage by building a separate compute layer before the storage layer, for intercepting and modifying requests in flight. Unlike our work, these approaches re-aggregate compute and storage, eschewing all the scaling benefits promised by FaaS.

ASFP [9] allows user code to run on storage nodes in an in-memory KV store. Compared to our approach, ASFP is more focused on coalescing many accesses to small objects, and is less flexible in the functions having to be specifically split into chunks of computation registered as extensions in the KV store.

S3 Select [4] and S3 Object Lambda [5] are existing FaaS storage solutions that try to address some of the issues we also address. However, *S3 Select* enables SQL-style statements to be computed in storage before sending the results, but for JSON or CSV formats only. In contrast, we support arbitrary code and data formats. At the same time, *Object Lambda* allows the user to create virtual objects with efficient implementations of this potentially running user-provided code on the storage nodes. However, we provide partial offloading of existing functions, requiring fewer changes to existing application code.

Adams, *et al.* [3] address the problem of sharding data across many storage nodes in a way that is efficient for near-data processing. Their research focuses on the low-level details about data layout across nodes, while we focus on how to make processing in storage easily accessible to FaaS applications.

Kayak [39] is a dynamic scheduler for choosing to run operations in or outside of an in-memory key-value store. In contrast, our work looks at persistent storage in storage nodes, which involves different access latency and throughput characteristics, and in our configuration significantly lower CPU power available for computation at the storage nodes creates an asymmetric scenario unlike the symmetric one presented in the Kayak paper.

8 Conclusion

To help with the always-increasing amount of data being processed in the Cloud, we proposed an extension to the traditional FaaS execution model that allows parts of FaaS Functions to be selectively offloaded onto Cloud storage node(s) on-the-fly, achieving better overall resource utilization and Function performance. We prototyped our extension on a state-of-the-art WASM-based FaaS runtime. We study the behaviour of different workloads on various degrees of offloading and evaluate our extension with these workloads on a realistic datacenter setup. We showed that our work, if combined with precise monitoring and configuration, can improve Function latency, throughput, and cross-cluster resource utilization in the datacenter. Similarly, many workloads which are considered unsuitable for FaaS now due to their storage requirements can be suitable candidates for FaaS on a datacenter that adopts our approach.

References

[1] 2024. *ARM Processor - AWS Graviton Processor - AWS.*

[2] 2024. *ORACLE Ampere A1 Compute.*

[3] Ian F. Adams, Neha Agrawal, and Michael P. Mesnier. 2021. Enabling near-data processing in distributed object storage systems. In *HotStorage '21*, Philip Shilane and Youjip Won (Eds.). ACM / USENIX Association, 28–34. https://doi.org/10.1145/3465332.3470881

[4] Amazon Web Services. 2021. Filtering and retrieving data using Amazon S3 Select. https://docs.aws.amazon.com/AmazonS3/latest/userguide/selecting-content-from-objects.html.

[5] Amazon Web Services. 2021. Introducing Amazon S3 Object Lambda. https://aws.amazon.com/blogs/aws/introducing-amazon-s3-object-lambda-use-your-code-to-process-data-as-it-is-being-retrieved-from-s3/.

[6] ARM. 2020. Computational Storage. https://www.arm.com/solutions/storage/computational-storage.

[7] asmjs 2014. asm.js Specification. http://asmjs.org/spec/latest/.

[8] Antonio Barbalace, Robert Lyerly, Christopher Jelesnianski, Anthony Carno, Ho-Ren Chuang, Vincent Legout, and Binoy Ravindran. 2017. Breaking the Boundaries in Heterogeneous-ISA Datacenters. In *Twenty-Second ASPLOS Proceedings*, Yunji Chen, Olivier Temam, and John Carter (Eds.). ACM, 645–659. https://doi.org/10.1145/3037697.3037738

[9] Ankit Bhardwaj, Chinmay Kulkarni, and Ryan Stutsman. 2020. Adaptive Placement for In-memory Storage Functions. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, Ada Gavrilovska and Erez Zadok (Eds.). USENIX Association, 127–141. https://www.usenix.org/conference/atc20/presentation/bhardwaj

[10] Sharath K. Bhat, Ajithchandra Saya, Hemedra K. Rawat, Antonio Barbalace, and Binoy Ravindran. 2016. Harnessing Energy Efficiency of Heterogeneous-ISA Platforms. *SIGOPS Oper. Syst. Rev.* 49, 2 (Jan. 2016), 65–69. https://doi.org/10.1145/2883591.2883605

[11] Dhruba Borthakur. 2013. Petabyte scale databases and storage systems at Facebook. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 1267–1268. https://doi.org/10.1145/2463676.2463713

[12] Yang Chen, Feng Zhang, Yinhao Hong, Yunpeng Chai, Wei Lu, Hong Chen, Xiaoyong Du, Peipei Wang, Le Mi, Jintao Li, Xilin Tang, Yanliang Zhou, Wei Zhou, Peng Zhang, Fengyi Chen, Pengfei Li, and Yu Li. 2022. Taming the Big Data Monster: Managing Petabytes of Data with Multi-Model Databases. In *2022 IEEE 34th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. 283–292. https://doi.org/10.1109/SBAC-PAD55451.2022.00039

[13] Li Deng. 2012. The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web]. *IEEE Signal Processing Magazine* 29, 6 (2012), 141–142. https://doi.org/10.1109/MSP.2012.2211477

[14] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 1–14. https://www.flux.utah.edu/paper/duplyakin-atc19

[15] Martin Gerlach and Francesc Font-Clos. 2018. A standardized Project Gutenberg corpus for statistical analysis of natural language and quantitative linguistics. arXiv:1812.08092 [cs.CL]

[16] WebAssembly Community Group. 2019. WebAssembly Core Specification. https://webassembly.github.io/spec/core/_download/WebAssembly.pdf.

[17] Felipe Gutierrez. 2021. *Cloud and Big Data.* Apress, Berkeley, CA, 3–8. https://doi.org/10.1007/978-1-4842-1239-4_1

[18] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the Web up to Speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 185–200. https://doi.org/10.1145/3062341.3062363

[19] Staysail Systems Inc. 2021. *NNG: Lightweight Messaging Library.* https://github.com/nanomsg/nng

[20] Intel. 2021. *Intel Xeon Gold 6434H Processor.* https://ark.intel.com/content/www/us/en/ark/products/232387/intel-xeon-gold-6434h-processor-22-5m-cache-3-70-ghz.html

[21] Intel. 2021. *Intel® Xeon® Gold 5415+ Processor.* https://ark.intel.com/content/www/us/en/ark/products/232373/intel-xeon-gold-5415-processor-22-5m-cache-2-90-ghz.html

[22] David Katz, Antonio Barbalace, Saif Ansary, Akshay Ravichandran, and Binoy Ravindran. 2015. Thread Migration in a Replicated-Kernel OS. In *2015 IEEE 35th International Conference on Distributed Computing Systems*. 278–287. https://doi.org/10.1109/ICDCS.2015.36

[23] Ana Klimovic, Yawen Wang, Christos Kozyrakis, Patrick Stuedi, Jonas Pfefferle, and Animesh Trivedi. 2018. Understanding Ephemeral Storage for Serverless Analytics. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 789–794. https://www.usenix.org/conference/atc18/presentation/klimovic-serverless

[24] H. Li, M. Hao, S. Novakovic, V. Gogte, S. Govindan, D. Ports, I. Zhang, R. Bianchini, H. S. Gunawi, and A. Badam. 2020. LeapIO: Efficient and Portable Virtual NVMe Storage on ARM SoCs. In *ASPLOS Proceedings*.

[25] Jinhong Li, Qiuping Wang, Patrick P. C. Lee, and Chao Shi. 2020. An In-Depth Analysis of Cloud Block Storage Workloads in Large-Scale Production. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*. 37–47. https://doi.org/10.1109/IISWC50251.2020.00013

[26] Kai Mast, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2022. LambdaObjects: Re-aggregating storage and execution for cloud computing. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems* (Virtual Event) *(HotStorage '22)*. Association for Computing Machinery, New York, NY, USA, 15–22. https://doi.org/10.1145/3538643.3539751

[27] nacl 2020. Native Client developer documentation. https://developer.chrome.com/docs/native-client/.

[28] NGD 2020. NGD Systems. https://www.ngdsystems.com/.

[29] SeongJae Park, Yunjae Lee, and Heon Y. Yeom. 2019. Profiling Dynamic Data Access Patterns with Controlled Overhead and Quality. In *Proceedings of the 20th International Middleware Conference Industrial Track* (Davis, CA, USA) *(Middleware '19)*. Association for Computing Machinery, New York, NY, USA, 1–7. https://doi.org/10.1145/3366626.3368125

[30] Song Jae Park. 2021. *MASIM: Memory Access Simulator.* https://github.com/sjp38/masim

[31] Song Jae Park. 2023. *DAMO: Data Access Monitoring Operator.* https://github.com/awslabs/damo

[32] Alexander Sage, Eirikur Agustsson, Radu Timofte, and Luc Van Gool. 2017. LLD - Large Logo Dataset - version 0.1. https://data.vision.ee.ethz.ch/cvl/lld.

[33] Josep Sampé, Marc Sánchez Artigas, Pedro García López, and Gerard París. 2017. Data-driven serverless functions for object storage. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference, Las Vegas, NV, USA, December 11 - 15, 2017*, K. R. Jayaram, Anshul Gandhi, Bettina Kemme, and Peter R. Pietzuch (Eds.). ACM, 121–133. https://doi.org/10.1145/3135974.3135980

[34] Simon Shillaker and Peter R. Pietzuch. 2020. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, Ada Gavrilovska and Erez Zadok (Eds.). USENIX Association, 419–433.

[35] Kenton Varda. 2018. WebAssembly on Cloudflare workers. https://blog.cloudflare.com/webassembly-on-cloudflare-workers/.

[36] WAVM. 2021. WAVM is a WebAssembly virtual machine. https://wavm.github.io.

[37] Sage A Weil. 2007. *Ceph: reliable, scalable, and high-performance distributed storage.* Ph. D. Dissertation. University of California, Santa Cruz.

[38] Sage A. Weil, Andrew W. Leung, Scott A. Brandt, and Carlos Maltzahn. 2007. RADOS: a scalable, reliable storage service for petabyte-scale storage clusters. In *Proceedings of the 2nd International Workshop on Petascale Data Storage: Held in Conjunction with Supercomputing '07* (Reno, Nevada) *(PDSW '07)*. Association for Computing Machinery, New York, NY, USA, 35–44. https://doi.org/10.1145/1374596.1374606

[39] Jie You, Jingfeng Wu, Xin Jin, and Mosharaf Chowdhury. 2021. Ship Compute or Ship Data? Why Not Both?. In *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021*, James Mickens and Renata Teixeira (Eds.). USENIX Association, 633–651.
[40] Qi Zhang, Joseph Hellerstein, and Raouf Boutaba. 2011. Characterizing Task Usage Shapes in Google Compute Clusters. In *Proceedings of the 5th International Workshop on Large Scale Distributed Systems and Middleware.*
[41] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. 2019. Narrowing the Gap Between Serverless and its State with Storage Functions. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019*. ACM, 1–12. https://doi.org/10.1145/3357223.3362723

Received 31 May 2024; revised 27 September 2024; accepted 25 October 2024