Pocket: Ephemeral Storage for Serverless Analytics

Paper #77

Abstract

Serverless computing is becoming increasingly popular, enabling users to quickly launch thousands of short-lived tasks in the cloud with high elasticity and fine-grain billing. While these properties make serverless computing appealing for interactive data analytics, a key challenge is managing intermediate data shared between tasks. Since communicating directly between short-lived serverless tasks is difficult, the natural approach is to store such ephemeral data in a common remote data store. However, existing storage systems are not designed to meet the demands of serverless applications in terms of elasticity, performance and cost. We present *Pocket*, a distributed data store that elastically scales to automatically provide applications with desired performance at low cost. Pocket dynamically rightsizes storage cluster resource allocations across multiple dimensions (storage capacity, network bandwidth and CPU cores) as application load varies. We show that Pocket cost-effectively rightsizes the type and number of resources such that applications are not bottlenecked on I/O.

1 Introduction

Serverless computing is becoming an increasingly popular cloud service due to its high elasticity and fine-grain billing. Serverless platforms like AWS Lambda, Google Cloud Functions, and Azure Functions enable users to quickly launch thousands of light-weight tasks (as opposed to entire virtual machines), while automatically scaling compute and memory according to application demands and charging users only for the resources their tasks consume, at millisecond granularity [11, 20, 33]. While serverless platforms were originally developed for web microservices, their elasticity and billing advantages make them appealing for data intensive applications such as *interactive analytics*. Several recent frameworks launch massive numbers of fine-grain tasks on serverless platforms to exploit all available parallelism in an analytics workload and achieve near real-time performance [18, 26, 16]. In contrast to traditional serverless applications that consist of a single function executed as new data arrives, analytics workloads typically consist of multiple stages and require sharing of state and data across stages of tasks (e.g., data shuffling).

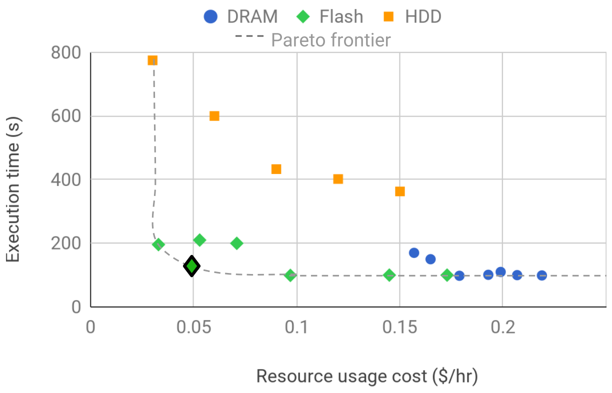

Figure 1: Performance-cost trade-off for a serverless video analytics job using different cluster resources.

In most cluster computing frameworks (e.g, Hadoop, Spark), data sharing is implemented with tasks buffering intermediate data in local storage and directly exchanging this data between each other over the network [43, 45]. However, serverless applications have no control over task scheduling or placement, making direct communication among tasks difficult. Moreover, serverless platforms achieve high elasticity by requiring tasks to be stateless, thereby preventing tasks from sharing data via the local file system. As a result of these limitations, the natural approach for data sharing in serverless applications is to use a remote storage service. For instance, early frameworks for serverless analytics either use object stores (e.g., S3 [10]), databases (e.g., CouchDB [1]) or distributed caches (e.g., Redis [29]).

Unfortunately, existing storage services are not a good fit for sharing short-lived intermediate data in serverless applications. We refer to the intermediate data as *ephemeral data* to distinguish it from input and output data which requires long-term storage. File systems, object stores and NoSQL databases focus on providing this durable, long-term, and highly-available storage at the expense of performance and cost. Distributed key-value stores offer good performance, but often impose limitations on the dataset size that can be cached, and burden users with managing the storage system in terms of scale and configuration. The availability of different storage technologies (e.g., DRAM, Flash, disk) further increases the complexity and makes finding the best configuration in terms of performance and cost difficult [28]. As an example, Figure 1 plots the performance-cost trade-off for a serverless video analytics application storing ephemeral data in a distributed data store with a variety of allocations in terms of storage technology, number of nodes, and available bandwidth resources. Each allocation results in different cost and performance. Finding Pareto efficient storage allocations for a single job is non-trivial and gets more complicated when running multiple jobs.

In this paper, we present Pocket, a distributed data store designed explicitly to support efficient data sharing in serverless analytics. Pocket offers high performance in terms of throughput and latency for arbitrary size data sets, combined with automatic fine-grain scaling, self management and intelligent data placement across different storage tiers such as DRAM, and Flash. The unique properties of Pocket are the result of a strict separation of responsibilities across three planes: a control plane which determines the right storage policies for jobs, a metadata plane which implements the distributed data placement decisions, and a "dumb" (i.e., metadata-oblivious) data plane responsible for storing the actual data. Pocket scales all three planes independently in a fine-grain manner based on the current load. For each analytics job, Pocket uses a set of heuristics to find a Pareto-optimal allocation in terms of storage technology and the number of storage servers used. We evaluate Pocket on AWS Lambda using three serverless workloads: video analytics, map-reduce sort and distributed source code compilation. Our results show that Pocket is capable of rightsizing the type and number of resources in a way that applications are provided with sufficient performance while minimizing cost. In summary, our contributions are as follows:

- We identify the key characteristics of ephemeral data in serverless analytics and synthesize requirements for storage platforms used for sharing such data among serverless tasks.
- Based on these findings, we introduce Pocket, which we believe is the first storage platform targeting the important use case of efficient data sharing in serverless analytics workloads.
- We evaluate Pocket on AWS Lambda for serverless analytics workloads, demonstrating its effectiveness in terms of performance and resource usage.

2 Storage for Serverless Analytics

Early work in serverless analytics has identified the challenge of storing and exchanging data between hundreds to thousands of fine-grain, short-lived tasks [26, 18]. We present a more thorough analysis of the performance demands of serverless analytics applications. Based on our findings, we synthesize essential properties for an ephemeral data storage solution for serverless analytics and discuss why current systems are unable to meet these demands. We focus on ephemeral data as the original input and final output data of analytics jobs typically has long-term availability and durability requirements that are well served by the variety of file systems, object stores, and databases available in the cloud.

2.1 Ephemeral Storage Requirements

High performance for a wide I/O range: Analytics applications on serverless vary considerably in the way they store, distribute, and process data. This diversity is reflected in the granularity of ephemeral data that is generated during a job. For example, in Figure 2 we show the size distribution for a 100 GB MapReduce sort with Pywren, a distributed lambda compilation of the `cmake` program, and video analytics using the Thousand Island (THIS) video scanner [26]. The x-axis shows ephemeral I/O size, and the y-axis shows the CDF. The key observation from the figure is that ephemeral data access granularity varies greatly in size, from hundreds of bytes to hundreds of megabytes. We observe a straight line for sorting as its ephemeral data size is equal to the partition size. However, with a different dataset size and/or number of workers, the location of the line changes. In addition to support for a wide range of I/O sizes, applications also expect high throughput and low latency to avoid I/O bottlenecks. For instance, we find that sorting 100 GB with 500 lambdas requires up to 7.5 GB/s ephemeral storage throughput. *Thus, an ephemeral data store must deliver high bandwidth, low latency, and high IOPS for the entire I/O range.*

Automatic and fine-grain scaling: One of the key promises of serverless computing is its agility to dynamically meet application demands. Serverless frameworks can launch thousands of short-lived tasks instantaneously. Consequently, an ephemeral data store for serverless applications can observe a storm of I/O requests within a fraction of a second. Once the load dissipates, the storage, just like the compute, resources should be scaled down. Scaling up or down to meet such performance demands requires a storage solution capable of growing and shrinking in multiple resource dimensions (e.g., adding/removing storage capacity and bandwidth, network bandwidth, CPU cores) with fine-grained elasticity and and time granularity (on the order of seconds). In addition, the storage service should scale cluster resources automatically and charge users only for the resource consumed by their application. Navigating the cloud storage landscape is challenging for users due to the vast number of storage cluster configuration options available, each providing different performance-

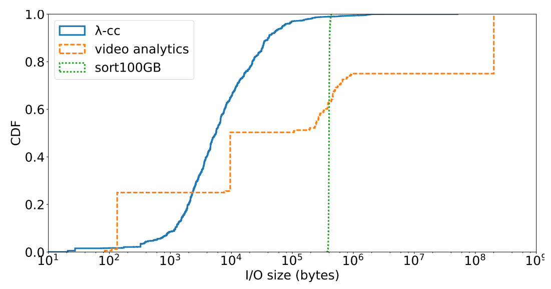

Figure 2: I/Os range from 100s of bytes to 100s of MBs.

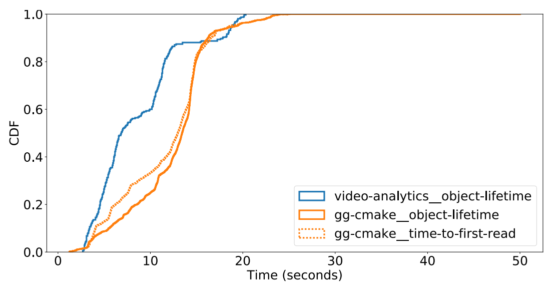

Figure 3: Object lifetime CDF.

cost trade-offs [28]. *Hence, an ephemeral data store must be able to automatically manage storage resources to satisfy application's I/O requirements while providing a pay-what-you-use cost model that matches the convenient abstraction serverless computing already provides for compute and memory resources.*

Cost-efficiency: Multiple storage media (HDD, Flash, DRAM) in the cloud allow for different performance-cost trade-offs, as shown in Figure 1. Ideally, the ephemeral data store provides applications with the required I/O performance while using the lowest cost resource configuration that satisfies these performance requirements. This corresponds to the Pareto optimal resource allocation point labeled in Figure 1; adding resources beyond this point only increases cost without improving execution time (storage is no longer the application performance bottleneck) while using any lower-cost resource allocation results in sub-optimal execution time. Although there exist systems that move data between multiple storage media based on various heuristics, these systems target long-term data as opposed to short-lived ephemeral data with lifetime as shown in Figure 3 [44, 14, 21, 7]. *A storage solution should be cost effective by provisioning just enough resources so that applications are not bottlenecked on I/O.*

Fault-(in)tolerance: Typically a data store must deal with failures while keeping the service up and running. Hence it is common for storage systems to use techniques such as replication and erasure codes. In a typical deployment, the cost of data (defined as a function of the likelihood of a failure and the total cost of resources that are required to recompute/generate the data) outweighs the overheads of fault-tolerance mechanisms. However, as shown in Figure 3, ephemeral data has a short lifetime of 10-100s of seconds, and, unlike input and output data, ephemeral data is only valuable during the execution of a workload. Furthermore, fault tolerance is typically baked into compute frameworks [45], where storing the data and computing it become interchangeable [22]. *Hence, in this work, we argue that an ephemeral storage solution does not have to provide high fault-tolerance as expected of traditional storage systems.*

2.2 Existing Systems

Existing storage systems do not satisfy the combination of requirements outlined in § 2.1. We describe different categories of systems and why they fall short for elastic ephemeral storage. Table 1 summarizes the characteristics of some example systems.

The first category of existing systems most commonly used today by serverless applications is fully-managed cloud services, such as Amazon S3 and DynamoDB, which extend the 'serverless' abstraction to storage, charging users only for the capacity and bandwidth they use [10, 17]. While such services automatically scale resources based on usage, they are optimized for high durability hence their agility is limited and they do not meet the performance requirements of serverless analytics applications. For example, DynamoDB only supports up to 400KB sized objects and S3 has high latency overhead (e.g., a 1 KB read takes ∼12 ms) and insufficient throughput for highly parallel applications (e.g., sorting 100 GB with 500 or more workers results in request rate limit errors when S3 is used for intermediate data).

In-memory key-value stores, such as Redis and Memcached, are another category of existing storage systems [29, 6]. These systems offer low latency and high throughput but at the higher cost of DRAM. They also require users to manage their own storage instances and manually scale resources. Although Amazon offers managed Redis and Memcached instances through its ElastiCache service, the service does not automate storage management as desired by serverless applications. Users must still select instance types with the appropriate memory, compute and network resources to match their application requirements. In addition, changing instance types or adding/removing nodes can require tearing down and restarting clusters, with nodes taking minutes to start up while the service is billed at hourly usage granularity [9].

Another category of systems use Flash storage to decrease cost while still offering good performance. For example, Aerospike is a popular Flash-based NoSQL database [39]. Alluxio/Tachyon [30] is another system that is designed to enable fast and *fault-tolerant* data

	Elastic Scaling	Latency	Throughput	Max object size	Cost
S3	Auto, coarse-grain	High	Medium	5 TB	$
DynamoDB	Auto, fine-grain, pay per hour	Medium	Low	400 KB	$$
Elasticache Redis	Manual	Low	High	512 MB	$$$
Aerospike	Manual	Low	High	1 MB	$$
Apache Crail	Manual	Low	High	any size	$$
Desired for λs	Auto, fine-grain, pay per second	Low	High	$\sim$1 GB	$

Table 1: Comparison of existing storage systems and desired properties for ephemeral storage in serverless analytics.

sharing between multiple jobs. Apache Crail is a distributed storage system that uses multiple media tiers to balance performance and cost. Unfortunately, users must manually scale storage cluster resources to adapt to elastic I/O requirements. Navigating the performance-cost trade-off space illustrated in Figure 1 and finding Parteo optimal deployments is challenging.

3 Pocket Design

Pocket is an elastic distributed storage service for ephemeral data that automatically and dynamically right-sizes storage cluster resource allocations to provide high I/O performance while minimizing cost. Pocket addresses the requirements outlined in §2.1 with following key design principles:

1. **Separation of responsibilities:** Pocket divides responsibilities across three different planes: a control plane, a metadata plane, and data plane. The control plane is responsible for generating policies based on job requirements. The metadata plane tracks the data stored in the data plane. All three planes consist of multiple servers and can be scaled independently in response to variations in the load.

2. **Sub-second response time:** All operations in Pocket are kept deliberately simple, targeting sub-second latencies. Pocket's storage servers are optimized for fast I/O and are only responsible for storing data (not metadata) making them simple to scale up or down. The controller scales resources at the granularity of seconds and balances load by carefully steering data for incoming jobs.

3.1 System Architecture

Figure 4 shows Pocket's system architecture. The system consists of three components: i) a (logically) centralized controller, ii) a set of metadata servers, and iii) a number of data plane storage servers. The controller (described in detail in §4) is responsible of allocating storage resources to analytics jobs and scaling the Pocket system up an down as the number and requirements of jobs varies. The controller further creates the data placement policies that decide which nodes and which storage technologies should be used to store the data of a job. The controller computes these policies based on heuristics and application characteristics that are adapted continously with repeated runs of the same application. The metadata servers are responsible of enfording the data placement policies generated by the controller. More precisely, metadata servers map client data objects to arrays of blocks spread across storage nodes and tiers in the cluster. Clients access data blocks on metadata-oblivious, performance-optimized storage nodes equipped with different storage media technologies (DRAM, Flash and/or HDD). The oblivious nature of storage nodes allows Pocket to quickly start up and tear down nodes for elastic scaling.

3.2 Application Interface

Table 2 describes the main functions that applications use to interface with Pocket. Pocket exposes an enhanced object store API with additional functions that are tailored to the ephemeral storage use-case. Similar to other object stores, Pocket provides a one-level hierarchical namespace of directories with objects stored inside directories – with just one extra aspect, that is, directories are further logically grouped on per job ID basis. Appplications can create, delete and enumerate directories, as well as create and delete objets and put and get data to/from objects. Serverless applications, upon startup, are further required to register with the Pocket controller using the *register_job* API call. Upon registration, the controller generates a unique job ID and a top-level directory by the same ID in its hierarchical namespace for all the objects generated by this job. The controller returns to the client the job ID and the IP address of the metadata server selected for this job. Note that the registration API allows the client to pass to the controller optional hints about the job's characteristics and requirements. We discuss in the next section how this hint is used for right-sizing the cluster. While the job is active, it keeps a connection alive with the controller. In the end, the job deregisters after finishing, which enables the controller to cleanly reclaim the storage resources by deleting the directory associated with the job ID.

3.3 Reading and writing

For all I/O operations, serverless clients first contact a metadata server (assigned to a job during the registration process) and provide the job's ID obtained from job registration. To write an object, a client issues an RPC to the metadata server, requesting capacity allocation for the object. The metadata server replies with a dataplane server IP/port address and a block number. The client then proceeds to connect to the dataplane server, and issues an RPC with the data to be written and block number. An object can be written using a streaming API, where when the capacity of a single block is exhausted, a new capacity allocation request is issued to the metadata server by the client. Hence, using this architecture, Pocket's objects can be of arbitrary size and span multiple data servers. For reading, a client performs a lookup for an object using its path name, and the metadata server replies with the location of the dataplane server that store the block. Internally operations that span multiple blocks, the client overlaps metadata RPCs for the next block while fetching data for the current block, hence, avoiding stalls. All lambdas in a job access the same metadata server. However, we do not expect this to limit single job performance since, as we will show in §6, a metadata server can handle 175K metadata operations per second. Clients cache metadata in case they need to read the same object multiple times.

3.4 Data Lifetime Management

Pocket targets ephemeral (i.e., short-lived) data for serverless analytics. By default, as discussed in §2.1, all ephemeral data has a defined lifetime, which is bounded by the runtime of a job. As soon as a job finishes (notified explicitly by deregistration from the controller), all data associated with the job is deleted, and the storage capacity is reclaimed. Pocket efficiently deletes all of a job's data with a single operation by deleting the top-level job ID directory. There are two exceptions to this default setup. First, if data is to be shared between multiple jobs, for example, the output of a streaming job becomes the input for a machine learning job. In this case, the deletion of data must be delayed. Second, most ephemeral data is only valuable during a particular stage, not the whole job. For example, neighbour's rank data during the pagerank calculation is only valid for a particular round of rank calculation. Hence, it can be deleted as soon as it is consumed without waiting for the job to deregister. For both these scenarios, Pocket's API allows I/O operations to specify data lifetime hints in order to prompt timely garbage collection and decrease capacity usage. A client can set the PERSIST data flag when writing an object if the object needs to be read by a future job; Pocket writes this data to a separate top-level directory which is still associated with the job but not deleted when the job is deregistered. Furthermore, since intermediate data is commonly written and read only once (e.g., a mapper writes an object destined for a single reducer), Pocket also allows clients to specify that data should be deleted immediately after it is read by setting the DELETE flag. The DELETE read hint is also used to delete data that was persisted beyond a job's lifetime.

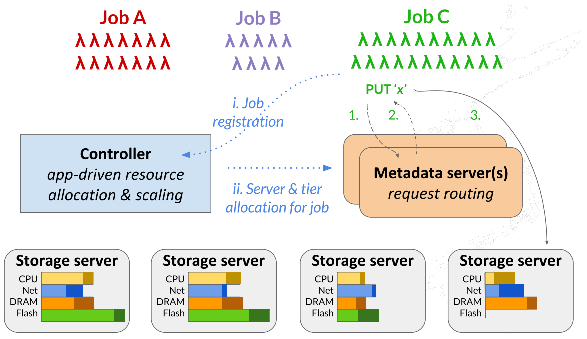

Figure 4: Pocket system architecture and the steps to register job C and issue a PUT from a lambda. The colored bars on storage servers show used and allocated resources for all jobs in the cluster.

3.5 Life of a Pocket application

To show how these components come together we walk through lifetime of a serverless analytics application that uses Pocket. At first, the application uses Pocket's client API to register with the Pocket controller, optionally providing hints about its characteristics (step i in Figure 4). The controller uses heuristics that take into account application hints to determine the storage capacity allocated for the job's ephemeral data, the storage tier used (DRAM, Flash, disk), and the number of storage servers to be used (step ii). The latter two decisions determine the storage throughput available for ephemeral data I/O. The controller assigns the job to a metadata server which is updated with the list of storage servers and tier used so that it can properly steer I/O requests. At this point,

Client API Function	Description
register_job(jobname, hints=None)	register job with controller and provide optional hints, get job ID and metadata server address
deregister_job(jobid)	notify controller job has finished, delete job's non-PERSIST data
connect(metadata_server_address)	open connection to metadata server
close()	close connection to metadata server
create_dir(jobid, dirname)	create a directory
delete_dir(jobid, dirname)	delete a directory
enumerate(jobid, dirname)	enumerate objects in a directory
put(jobid, src_filename, obj_name, PERSIST=false)	write data, set PERSIST flag if want data to remain after job finishes
get(jobid, dst_filename, obj_name, DELETE=false)	read data, set DELETE true if data is only read once
delete(jobid, obj_name)	delete data

Table 2: Main functions exposed by Pocket's client API.

the job can launch lambdas that access Pocket to store data that they want to share. Lambda clients write/read data in Pocket by first contacting their assigned metadata server to find out which data plane servers they should read/write data objects from (steps 1 and 2). Data objects are chopped and distributed over multiple data plane servers in the granularity of blocks. Clients can send/receive data to/from multiple storage servers in parallel. They can also specify hints about the lifetime of the data read/written to optimize garbage collection performed by metadata servers. When the last lambda in a job finishes, the job deregisters the job to free up Pocket resources. As jobs execute, the controller monitors resource utilization in storage and metadata servers (the horizontal bars on storage servers in Figure 4) and adds or removes servers as needed to minimize cost while providing high application performance

4 Rightsizing Resource Allocations

Pocket's control plane elastically and automatically rightsizes the storage cluster across multiple resource dimensions, navigating the performance-cost trade-off space (e.g., Figure 1) for each job. Pocket prioritizes providing applications with the I/O performance they require with the secondary goal of minimizing resource allocation cost. When a job registers, Pocket's controller conservatively estimates its latency, throughput and capacity requirements based on optional user-provided hints. §4.1 describes how Pocket uses hints to estimate resource requirements and the controller's online bin-packing algorithm for deciding a job's data placement across nodes and storage media tiers. In addition to cost-effectively rightsizing resource allocations for each individual job, Pocket monitors overall resource utilization in the cluster and decides when to add and remove storage and metadata nodes based on load. §4.2 describes Pocket's mechanism for adding and removing nodes along with its data steering policy to balance load.

4.1 Rightsizing I/O per Application

Pocket's controller allocates the resources a job needs with the primary goal of ensuring that job execution time is not bottlenecked on I/O and the secondary goal of minimizing cost. Pocket decides resource allocation along three main dimensions: throughput allocation, capacity allocation and the choice of storage tier. The throughput allocation is based on the bandwidth the application needs to not be bottlenecked on I/O, the capacity allocation is based on the amount of ephemeral data generated by the job, and the choice of storage tier depends on the application's sensitivity to latency. Thus, when a job registers with the controller, the first step is to estimate its throughput, capacity and latency requirements based on application-driven heuristics. The determining the *resource allocation* required for the job, the second step is to translate these requirements into a *resource assignment* across nodes. Pocket generates a *weightmask* for each job to define the list of storage servers (identified by IP address and port number) across which the job's data should be placed, along with a weight from 0 to 1 for each server in the weightmask. While the controller generates the weightmask, this data placement policy is enforced by the metadata server which routes client write requests to storage nodes, spreading data across the storage servers in the job's weightmask using weighted random selection. We now describe the resource allocation and assignment steps in more detail.

Determining job I/O requirements: Pocket uses heuristics that adapt to optional hints provided by the user to estimate the throughput, capacity, and type of

Hint	Impact on throughput	Impact on capacity	Impact on storage media
No hint (default policy)	Use default cluster (e.g., 50 `i3.2xlarge` nodes give 50 $\times$ 8 Gb/s)	Use default cluster (e.g., 50 `i3.2xlarge` nodes give 50 $\times$ (60 + 1900) GB)	Fill DRAM across nodes first, then Flash
Latency sensitivity	-	-	Use Flash if not latency sensitive, use DRAM first if latency sensitive
Maximum number of concurrent lambdas = N	Provision per-λ network limit $\times N$ (e.g., 0.6 $\times N$ Gb/s)	Provision capacity based on per-λ throughput limit (e.g., $\frac{0.6N}{8\text{ Gb/s}} \times$ (60 + 1900) GB)	-
Ephemeral capacity C	-	Provision capacity C	-
Peak bandwidth B	Provision throughput B	-	-

Table 3: The impact that hints provided about the application have on Pocket's resource allocation decisions for throughput, capacity and the choice of storage media (with specific examples in parentheses for our AWS deployment with `i3.2xlarge` instances, each with 8 cores, 60 GB DRAM, 1.9 TB Flash and $\sim$8 Gb/s network bandwidth).

storage media to allocate for a job. Table 3 lists the hints that Pockets supports and the impact of each hint on the throughput, capacity and the choice of storage media allocation for a job. We also provide an example for our deployment of Pocket on AWS. Given no hints about the application, Pocket uses a conservative baseline which over-provisions resources to achieve good performance at high cost. In our AWS deployment, we use a default cluster size of 50 `i3.2xlarge` nodes which supports up to $\sim$50 GB/s aggregate throughput. Pocket conservatively assumes a job is latency sensitive and thus places the job's data on DRAM before spilling to Flash. If a user hints that a job is not sensitive to latency, Pocket places the job's data on Flash to save cost. Pocket can also be configured to spill to HDD although on AWS, we find NVMe Flash has higher performance-cost efficiency than HDD. If a user provides the maximum number of concurrent lambda tasks, N, Pocket allocates throughput corresponding to N times the peak throughput per lambda (e.g., $\sim$600 Mb/s per lambda on AWS) for the job. N can be either be limited by the job's inherent parallelism or by the user's invocation limit imposed by serverless cloud provider (e.g., 1000 default on AWS). Unless the capacity requirement is provided by the user, Pocket allocates capacity proportional to the number of nodes allocated for throughput (e.g., close to 2 TB per node for `i3.2xlarge` instances). Users can also directly provide throughput and capacity requirements which override Pocket's estimates based on the number of lambdas. The latency sensitivity hint is complementary to throughput and capacity hints.

Assigning resources: After determining the throughput, capacity and choice of storage media to allocate for a job, Pocket translates this resource allocation into a resource assignment on specific storage servers in the cluster, represented as a weightmask. The assignment is guided by the available resources in the cluster and whether the job is capacity or throughput bound. A job is throughput bound if its throughput to capacity requirement ratio is greater than or equal to the ratio of throughput and capacity on cluster nodes. For example, a 100GB sort requires up to 7.5GB/s, making the job throughput-limited on the AWS instances we use, hence the number of nodes assigned to this job is based on its throughput requirements. Pocket uses an online bin-packing algorithm which first tries to fit a job on active storage servers and only launches new servers when the job's requirements can not be satisfied by sharing resources with other jobs. If the controller determines the job requires more resources, the controller launches the necessary storage nodes while the application waits for its job registration command to return. As we will show in §6.2, starting up nodes takes only a few seconds to a couple of minutes, depending on whether a new VM or an additional container is launched.

4.2 Rightsizing Cluster Size

While Section 4.1 described how Pocket rightsizes resources for a single application, we now discuss how Pocket dynamically scales cluster resources to accommodate elastic application load with multiple jobs. The Pocket cluster consists of a few long-running servers used to host the controller, the minimum number of metadata servers (one) and the minimum number of storage servers (in our evaluation, we use a minimum of two). Beyond these persistent resources, Pocket adds and removes additional resources on demand based on load. We first describe the mechanism to add/remove resources and then discuss the policy Pocket uses to balance cluster load by carefully steering requests across servers.

Mechanisms: The controller monitors overall cluster resource utilization by processing heartbeats from storage and metadata servers containing their CPU, network, DRAM capacity and/or Flash capacity usage. Heartbeats are also exchanged between metadata and storage servers to inform the metadata servers about the current state of a storage server and also to facilitate with storage server removal. Nodes send statistics every second (the interval is configurable).

When launching a new storage server, the controller provides the IP addresses of all metadata servers that the storage server must establish connections with to join the cluster. A new storage server sends a request to each metadata server to register its available capacity and periodically sends heartbeats with updated information about available capacity. To remove a storage server, the controller first blacklists the storage server by removing the server from the list of eligible servers for weightmask assignment of incoming jobs. The controller informs metadata servers to blacklist the storage server. When all capacity on the blacklisted server eventually becomes available (since ephemeral data has a short, bounded lifetime and will be promptly garbage collected), the storage server terminates and gives up its resources. In addition to adding/removing storage servers, the controller can also instruct (via heartbeat messages) current storage servers to use additional CPU cores, if the controller observers that CPU utilization is becoming a bottleneck and additional cores are available on the node. This allows Pocket to scale resources in a fine-grain manner.

Cluster sizing policy: Pocket elastically scales the cluster with the goal of maintaining overall resource utilization for each resource type (DRAM, Flash, CPU and network bandwidth utilization) within a desired range. The target utilization range can be configured separately for each resource type and is managed separately for metadata and storage servers. We use a lower utilization target of 60% and a upper utilization target of 80% for all resource dimensions, for both the metadata and storage nodes. The range is empirically tuned and depends on the time it takes to add/remove nodes. Pocket's controller scales down the cluster by removing a storage server if overall CPU, network bandwidth *and* capacity utilization is below the lower limit of the target range. Pocket chooses to remove a DRAM vs. Flash storage server by removing the server belonging to the tier with lower capacity utilization. Pocket adds a storage server if overall CPU, network bandwidth *or* capacity utilization is above the upper limit of the target range. While Pocket scales throughput and capacity by launching/removing nodes, to respond to CPU load spikes or lulls, Pocket first tries to scale up/down the number of CPU cores that metadata and/or storage servers use before adding/removing nodes. The clusters always keeps the minimum resources needed to run the controller and a single metadata server active. Pocket also requires a few (we use a minimum of two) long-running storage servers which are used to place data data that users request to persist (via the PERSIST hint described in §3.2) for longer durations such as to share data between jobs.

Balancing load with data steering: To balance load when dynamically scaling the Pocket cluster, the controller leverages the short-lived nature of ephemeral data and serverless jobs. Since ephemeral data objects only live for tens to hundreds of seconds (as observed in Figure 3), it is not efficient to migrate data between nodes to balance load when nodes are added or removed. Instead, Pocket relies on steering the data for new, incoming jobs across storage servers as they join the cluster by including these servers in the weightmask. The controller assigns a more weight to under-utilized storage servers when generating job weightmasks in order to balance load. The controller uses a similar approach, but at a coarser granularity, to balance load across metadata servers. The controller assigns jobs to particular metadata servers (since all of a job's data is contained in a single top-level directory, unless the data needs to be persisted beyond the job's lifetime). Thus by steering incoming job registration requests to metadata servers, the controller dynamically spreads load across metadata servers. Serverless jobs have bounded lifetime, for example 5 minutes on AWS Lambda.

5 Implementation

Controller: The Pocket controller is implemented in python and leverages the Kubernetes container orchestration system to launch and tear down metadata servers and storage servers running in separate Docker containers [5]. In addition to launching containers, the Pocket controller can spin up virtual machines which run the containers. As explained in §4.2, Pocket rightsizes cluster resources to maintain a target utilization range. We empirically tune the target utilization range based on anticipated startup time.

Metadata management: We implement Pocket's metadata and storage server architecture on top of the Apache Crail distributed storage system [2, 40]. Crail is an I/O architecture designed for high performance storage of arbitrarily sized data with low durability requirements. Its modular architecture supports pluggable storage tiers and RPC libraries. While Crail is originally designed for RDMA networks, we implement a TCP-based RPC library for Pocket since RDMA is not readily available in public clouds. We also implement multiple storage tiers which leverage different storage media to balance cost and performance trade-offs. Like Crail, Pocket's metadata servers are implemented in Java. We run each metadata and storage server inside a separate Docker container launched by the controller.

Storage tiers: We implement three different storage tiers for Pocket. The first is a DRAM tier implemented in Java, using NIO APIs to efficiently serve requests from clients over TCP connections. The second tier uses NVMe-Flash storage devices. We implement NVMe storage servers for Pocket on top of ReFlex, a system that allows clients (i.e., lambdas in our use-case) to access Flash over commodity networks with high performance [27]. ReFlex is implemented in C and leverages Intel's DPDK [24] and SPDK [25] libraries to directly access network and NVMe device queues from userspace. ReFlex uses a polling-based execution model to efficiently process network storage requests and the system uses a quality of service (QoS) aware scheduler to manage read/write interference on Flash and provide predictable performance to clients. The third tier we implement is a generic block storage device tier that allows Pocket to use any block storage device (e.g., hard drive disk or SATA/SAS SSD) via a Linux kernel driver. Similar to ReFlex, this tier is implemented in C and uses DPDK for efficient networking. This tier uses the Linux libaio library for asynchronous I/O request processing. Since we find that NVMe Flash is significantly more performance-cost efficient than disk (using Amazon's EC2 pricing model), our evaluation focuses on the DRAM and Flash tiers.

Client library: The application interface that Pocket exposes is shown in Table 2. Since the serverless applications we use are written in Python, we implement this API as a Python client library for Pocket. To optimize performance, we implement the core of the library in C++ and use `Boost` to wrap the code into a Python library. The library internally manages TCP connections with metadata and storage servers.

6 Evaluation

We evaluate the latency and throughput of Pocket's metadata and data planes as well as the control plane's ability to elastically and cost-effectively rightsize resources.

6.1 Methodology

We deploy pocket on Amazon Web Service (AWS) cloud infrastructure. We use EC2 instances as nodes for the Pocket storage servers, metadata servers and controller. DRAM servers run on `r4.2xlarge` instances, NVMe Flash servers run on `i3.2xlarge` instances, HDD servers run on `h1.2xlarge` instances while metadata servers and the controller run on `m5.xlarge` instances. We run Pocket storage servers and metadata servers as containers inside these VMs and leverage Kubernetes v1.9 to orchestrate containers at the controller. We use AWS Lambda as our serverless computing framework to run applications. We enable lambdas to access Pocket EC2 nodes by deploy them in the same virtual private cloud (VPC). We configure lambdas with the maximum supported memory (3 GB) and Amazon allocates compute resources for lambdas proportional to the amount of memory requested [12]. We compare Pocket's performance and cost-efficiency to ElastiCache Redis (cluster-mode enabled) and Amazon S3 [9, 29, 10].

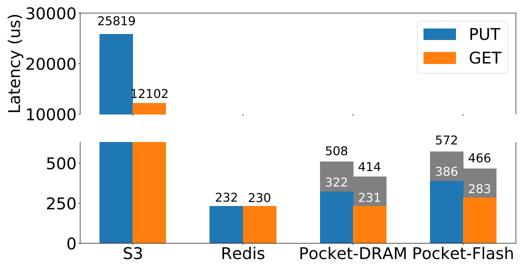

Figure 5: Unloaded latency for 1KB access from lambda

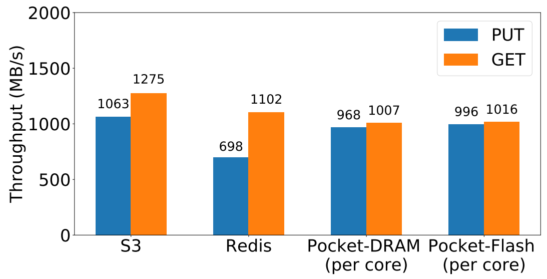

Figure 6: 1 MB I/O throughput measured from lambdas

We consider three different serverless applications:

Video analytics: We use the Thousand Island Scanner (THIS) for distributed video processing [35]. The first stage of lambdas read compressed video frame batches, decode the frames, and write decoded data to ephemeral storage. Each lambda then launches a second stage lambda which reads decoded frames from ephemeral storage, computes a MXNET deep learning classification algorithm and output the classification result. We run THIS with a 25 minute video consisting of 40K 1080p frames. We set the batch size such that the first stage consists of 160 lambdas and the second, more compute-bound stage, has 305 lambdas.

MapReduce Sort: We implement a MapReduce style sort application on AWS Lambda, similar to PyWren [26]. Lambda map workers fetch input files from long-term storage (S3, in our case) and write intermediate files to ephemeral storage. Reduce lambdas merge and sort intermediate data and upload output files to long-term storage. We run a 100GB sort, which generates 100GB of ephemeral data.

Distributed software compilation: We upload source code to ephemeral storage, then use a framework (which we refer to as lambda-cc) to infer the software build dependency tree and invoke lambdas to compile the code with high parallelism. Each lambda fetches its dependencies from ephemeral storage, computes (i.e., compiles, archives or links), and writes its output to ephemeral storage, including the final executable for the user to download. We present results for compiling the `cmake` project source code. This build job has a maximum parallelism of 650 tasks and generates a total of 850 MB ephemeral data.

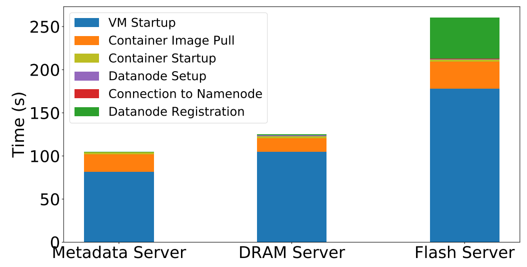

Figure 7: Node startup time breakdown

6.2 Microbenchmarks

We measure the latency of Pocket storage and metadata operations from lambda clients, the throughput achieved per core on Pocket storage and metadata servers, and the time it takes to start and stop these servers.

Storage request latency: Figure 5 compares the latency of S3, Redis, and Pocket measured from a lambda client. Access time to Redis and Pocket is below 600 μs, which is over 40× faster than S3. Pocket has higher latency than Redis, mainly due to the metadata RPC latency (shown in gray). While Redis cluster clients simply hash keys Redis nodes, Pocket clients must contact a metadata server to get storage location information. While the extra round trip increases request latency, it allows Pocket to carefully manage data placement for performance-cost optimization and to dynamically scale cluster nodes in response to load.

Storage request throughput: We compare the throughput of S3, Redis (`cache.r4.2xlarge`) and Pocket by launching 100 concurrent lambdas clients. As shown in Figure 6, both Pocket DRAM and Flash tiers achieve ∼1 GB/s. We also measure IOPS for 1KB requests for the three storage systems (not shown in the figure). Using 200 lambdas, we find Redis supports 178K GET IOPS, while Pocket-DRAM and Pocket-Flash reach 134K GET IOPS and 78K GET IOPS, respectively, with a single core. While S3 achieves comparable throughput to Redis and Pocket for 1MB requests (Figure 6), the storage service is incapable of handling high request rates. S3 incurs a `SlowDown` error that requires clients to back off. This makes S3 unusable for throughput-intensive analytics application (e.g., for the MapReduce sort application).

Metadata throughput: We measure the number of metadata operations issued from lambdas that a metadata server can handle per second. A single core metadata server on the `m5.xlarge` instance supports up to 90K operations per second and up to 175K operations per second with four cores. Thus, although Pocket limits each job to access a single metadata server, we do not expect this to become a bottleneck as a single job is unlikely to issue over 175K metadata operations per second.

Adding/removing servers: Since we deploy Pocket in containers on AWS EC2 nodes, the time to start a Pocket server can be broken down as the time to launch a VM, the time to pull the container image, and the time to launch the container. A Pocker storage servers must also register its storage capacity with metadata servers to join the cluster. Figure 7 shows the time breakdowns. The VM launch time varies across instance EC2 instance types. The container image for both Pocket metadata server and DRAM server have a compressed size of 249MB while the Pocket-Flash container image is 540MB compressed due to dependencies for DPDK and SPDK to run ReFlex. The image pull time depends on the VM's network bandwidth. The VM launch time and container image pull time only need to be done once when the VM is first started. Once the VM is 'warm' (the image is available locally), starting and stopping containers takes only a few seconds. The time to terminate a VM is tens of seconds. The Flash server has a significantly longer registration time because it registers a larger capacity with the metadata servers.

6.3 Serverless Analytics with Pocket

We now evaluate Pocket with the three different serverless applications described in §6.1.

Rightsizing with application hints: Figure 8 shows how Pocket leverages user hints to make cost-effective resource allocations for three different jobs, assuming each hint is provided in addition to the previous ones. With no knowledge of application requirements, Pocket defaults to a policy that spreads data for a job across a default cluster of 50 nodes, filling DRAM first then Flash. Since the jobs consist of 160 to 650 lambdas, with knowledge of the number of lambdas, Pocket allocates lower aggregate throughput than with the default policy, hence requiring less nodes and lower cost. Since the applications are not latency sensitive (MapReduce sort and the first stage of video analytics are throughput-intensive while lambda-cc and the second stage of video analytics are compute-limited), Pocket uses Flash resources as opposed to DRAM to save cost. The green bar shows the al-

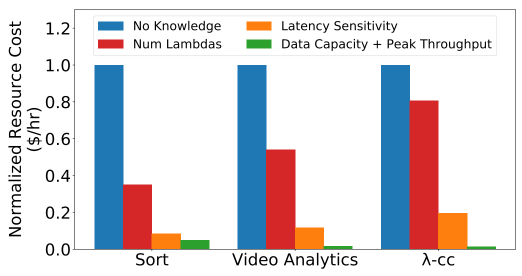

Figure 8: Pocket cost-effectively allocates resources based on user hints.

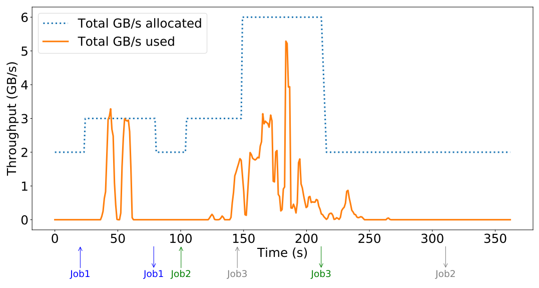

Figure 10: Pocket's controller dynamically scales cluster resources to meet I/O requirements as jobs come and go.

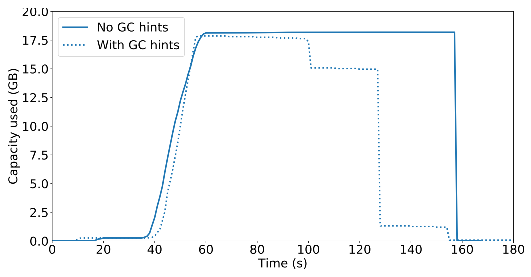

Figure 9: Video analytics with delete-after-read garbage collection hints reduces storage capacity usage.

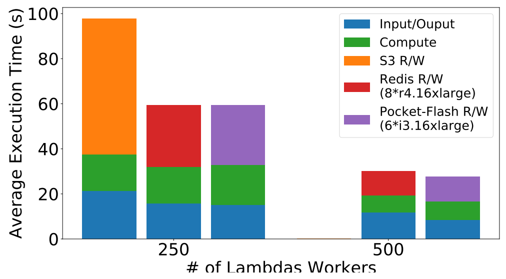

Figure 11: Execution Time Breakdown of 100GB Sort

location when the application provides explicit hints for its capacity and peak throughput requirements (this can be obtained by users from a profiling run). Reducing the cost of resource allocations with hints in Figure 8 does not affect execution time.

Reclaiming capacity using hints: Figure 9 shows the capacity savings achieved when users specify delete-after-read hints using Pocket's client API, enabling Pocket to promptly garbage collect data. As an example, we run the video analytics application, in which all ephemeral data is written and read only once, with and without setting the delete-after-read GC hint for GET operations. When no GC hint is provided, Pocket deletes the job's data when the job deregisters, whereas with hints, Pocket can readily reclaim capacity and offer it to other jobs in the cluster.

Rightsizing cluster size: Elastic and automatic resource scaling is a key property of Pocket. Figure 10 shows how Pocket scales cluster resources as multiple jobs register and deregister with the controller. Job registration and deregistration times are indicated by upwards and downwards arrows, respectively. In this experiment, we assume the user explicitly provides capacity and throughput hints of the application's requirements. The first job is a 10GB sort application requesting 3 GB/s, the second job is a video analytics application requesting 2.5 GB/s and the third job is a different invocation of a 10 GB sort also requesting 3 GB/s. Each storage server provides 1 GB/s. We set the minimum number of storage servers in the cluster. In total, we provision seven VMs for this experiment and ensure that storage server containers are locally available, such that when the controller launches new storage servers, only container startup and capacity registration time is included. As shown in the figure, Pocket quickly and effectively scales resources to meet application demands. The latency spike that surpasses the allocated throughput is due to the AWS instance providing higher than expected network bandwidth for a short period of time.

Comparison with S3 and Redis: Finally, we compare end-to-end application performance for a 100GB MapReduce sort application with S3, Redis and Pocket. S3 doesn't support sufficient request rates for the 500 lambda scenario. Pocket provides similar throughput to Redis but at significantly lower cost by using Flash instead of DRAM.

7 Discussion

Security: Pocket uses access control to secure applications in a multi-application environment. To prevent malicious users from accessing other tenants' data, Pocket metadata servers issue single-use certificates to clients which are verified at storage servers. I/O requests that are not accompanied with a valid certificate are denied. Clients communicate with metadata servers over SSL to protect against 'man in the middle' attacks. Users should set cloud network security rules to prevent malicious users from snooping TCP traffic on connections between lambda clients and Pocket storage servers. Alternatively, users can encrypt their data.

Resource harvesting: An additional, desirable way to reduce costs is to harvest idle datacenter resources. Datacenter providers commonly over-provision resources including CPU cores, DRAM and storage to accommodate unexpected load spikes, resulting in vast capacity underutilization [15, 42]. While these resources are only temporarily idle, harvesting them opportunistically is viable for ephemeral data storage due to low data durability requirements. Leveraging these 'free' resources for Pocket can dramatically reduce total cost of ownership.

Limitations: Pocket currently uses conservative heuristics to determine application requirements and relies on user hints to cost-effectively rightsize resource allocations for an application. The system does not currently autonomously learn properties of applications. Since users may repeatedly run jobs [32]as many data analytics and modern machine learning jobs are recurring, Pocket's controller could store state about previous job runs and use this information combined with machine learning techniques to rightsize resource allocation for future runs. We plan to explore this in future work.

8 Related Work

Section 2 provided an overview of related storage systems. Here we discuss work related to resource scaling and data placement across multiple storage media.

Elastic resource scaling: Various reactive (e.g., AutoScale [19]) and predictive solutions [37, 34] have been proposed [36] to automatically scale resources up and down based on demand. Pocket uses a reactive approach to add and remove nodes in the cluster for incoming jobs based on resource utilization monitoring. Similar to Kubernetes Horizontal Pod autoscaling [4], Pocket collects multidimensional metrics, some of them are provided and interpreted with the help of the application. Ameoba scales resources with possibility of a safe interruption of over-provisioned jobs by sizing tasks dynamically depending on global scarcity or abundance of resources [13]. Google has a VM rightsizing service for compute resources [3], however, it takes a single machine view. Systems like CloudScale [38], Elastisizer [23], Ernest [41], and CherryPick [8] take an application-centric view to right-size a workload. However, their analysis is based on traditional compute resources (i.e., VMs) as opposed to fine-grained abstractions like serverless tasks. The difference lies in the granularity of storage scaling requirements. However, the proposed approaches for buildling performance and cost models (e.g., Bayesian Optimization in CherryPick) are useful for Pocket as well.

Automating and optimizing data placement: Lim et al. propose autoscaling for storage cluster while consideration rebalancing, actuator lag, and interference [31]. In contrast, Pocket does not rebalance data when adding or removing node as the cost of necessary mechanisms out-weight the value of ephemeral data. Similar to the Pocket controller, Mirador is a dynamic storage placement service that *constantly* optimizes data placement for performance, efficiency, and safety [44]. However, Mirador focuses on long-running jobs (minutes to hours), not short-term ephemeral storage (in seconds) that Pocket targets. Tuba makes a case for configurable geo-replicated storage system [14]. Similar to Tuba, the Pocket controller receives performance and cost constraint from applications and optimizes for them. However, unlike Tuba, Pocket clients (i.e., lambda nodes) do not periodically communicate with the controller to update. Such a design is not feasible for ephemeral data with a life-time of few seconds/minutes. Extent-based Dynamic Tiering (EDT) [21] system leverages multi-tier storage media to provide a most cost effect storage solution for a given workload using access pattern simulations and monitoring. In contrast, the workload pattern of ephemeral data is simple and predictable (e.g., write-once-read-once). Hence, in Pocket multi-tier storage (DRAM and flash) is integrated in a single-level without any movement between data from DRAM to flash or vice-versa. Janus [7] provides flash provisioning recommendations for multiple workloads based upon their workload traces. Pocket can use the similar techniques to initially provision flash storage for workloads.

9 Conclusion

General-purpose analytics on a serverless infrastructure is an exciting field of research which presents some unique problems and opportunities. In this paper, we have analyzed challenges associated specifically with efficient data sharing, and presented Pocket, an ephemeral data store for serverless analytics. In a similar spirit to serverless computing, Pocket aims to provide a highly elastic, cost-effective, and fine-grained storage solution for analytics workloads. Pocket achieves these goals using a strict separation of responsibilities regarding control (when), metadata (how), and data (what) management. To the best of our knowledge, Pocket is the first system designed specifically for ephemeral data sharing in serverless analytics workloads. Our evaluation on AWS demonstrates that Pocket offers high performance data access for arbitrary size data sets, combined with automatic fine-grain scaling, self management and cost effective data placement across multiple storage tiers.

References

[1] Apache CouchDB. http://couchdb.apache.org, 2018.

[2] Apache crail (incubating). http://crail.incubator.apache.org/ , 2018.

[3] Applying Sizing Recommendations for VM Instances. https://cloud.google.com/compute/docs/instances/apply-sizing-recommendations-for-instances, 2018.

[4] Horizontal Pod Autoscaler. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/, 2018.

[5] Kubernetes: Production-Grade Container Orchestration. https://kubernetes.io/, 2018.

[6] Memcached – a distributed memory object caching system. https://memcached.org/ , 2018.

[7] Christoph Albrecht, Arif Merchant, Murray Stokely, Muhammad Waliji, Francois Labelle, Nathan Coehlo, Xudong Shi, and Eric Schrock. Janus: Optimal flash provisioning for cloud storage workloads. In *Proc. of the USENIX Annual Technical Conference*, ATC'13, pages 91–102, 2013.

[8] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 469–482, Boston, MA, 2017. USENIX Association.

[9] Amazon. Amazon ElastiCache. https://aws.amazon.com/elasticache/ , 2018.

[10] Amazon. Amazon simple storage service. https://aws.amazon.com/s3 , 2018.

[11] Amazon. AWS lambda. https://aws.amazon.com/lambda , 2018.

[12] Amazon. AWS lambda limits. https://docs.aws.amazon.com/lambda/latest/dg/limits.html , 2018.

[13] Ganesh Ananthanarayanan, Christopher Douglas, Raghu Ramakrishnan, Sriram Rao, and Ion Stoica. True elasticity in multi-tenant data-intensive compute clusters. In *Proc. of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 24:1–24:7, 2012.

[14] Masoud Saeida Ardekani and Douglas B. Terry. A self-configurable geo-replicated cloud storage system. In *Proc. of the 11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'14, pages 367–381, 2014.

[15] Luiz Andr Barroso, Jimmy Clidaras, and Urs Hlzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition.* 2013.

[16] Databricks. Databricks serverless: Next generation resource management for Apache Spark. https://databricks.com/blog/2017/06/07/databricks-serverless-next-generation-resource-management-for-apache-spark.html , 2017.

[17] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, 2007.

[18] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *Proc. of the 14th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'17, pages 363–376, 2017.

[19] Anshul Gandhi, Mor Harchol-Balter, Ram Raghunathan, and Michael A. Kozuch. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *ACM Trans. Comput. Syst.*, 30(4):14:1–14:26, November 2012.

[20] Google. Cloud functions. https://cloud.google.com/functions , 2018.

[21] Jorge Guerra, Himabindu Pucha, Joseph Glider, Wendy Belluomini, and Raju Rangaswami. Cost effective storage using extent based dynamic tiering. In *Proc. of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, pages 20–20, 2011.

[22] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, Yuan Yu, and Li Zhuang. Nectar: Automatic management of data and computation in datacenters. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 75–88, Berkeley, CA, USA, 2010. USENIX Association.

[23] Herodotos Herodotou, Fei Dong, and Shivnath Babu. No one (cluster) size fits all: Automatic cluster sizing for data-intensive analytics. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 18:1–18:14, New York, NY, USA, 2011. ACM.

[24] Intel Corp. Dataplane Performance Development Kit. https://dpdk.org, 2018.

[25] Intel Corp. Storage Performance Development Kit. https://01.org/spdk, 2018.

[26] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*, SOCC'17, pages 445–451, 2017.

[27] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Reflex: Remote flash == local flash. In *Proc. of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 345–359, 2017.

[28] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Learning heterogeneous cloud storage configuration for data analytics. In *The Inaugural Conference on Systems and Machine Learning*, SysML'18. http://www.sysml.cc/doc/82.pdf , 2018.

[29] Redis Labs. Redis. https://redis.io , 2018.

[30] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proc. of the ACM Symposium on Cloud Computing*, SOCC '14, pages 6:1–6:15, 2014.

[31] Harold C. Lim, Shivnath Babu, and Jeffrey S. Chase. Automated control for elastic storage. In *Proceedings of the 7th International Conference on Autonomic Computing*, ICAC '10, pages 1–10, New York, NY, USA, 2010. ACM.

[32] Omid Mashayekhi, Hang Qu, Chinmayee Shah, and Philip Levis. Execution templates: Caching control plane decisions for strong scaling of data analytics. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, pages 513–526, Berkeley, CA, USA, 2017. USENIX Association.

[33] Microsoft. Azure functions. `https://azure.microsoft.com/en-us/services/functions` , 2018.

[34] Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah, and John Wilkes. AGILE: Elastic distributed resource scaling for infrastructure-as-a-service. In *Proc. of the 10th International Conference on Autonomic Computing*, ICAC'13, pages 69–82, 2013.

[35] David Durst Qian Li, James Hong. Thousand island scanner (THIS): Scaling video analysis on AWS lambda. `https://github.com/qianl15/this` , 2018.

[36] Chenhao Qu, Rodrigo N. Calheiros, and Rajkumar Buyya. Auto-scaling web applications in clouds: A taxonomy and survey. *CoRR*, abs/1609.09224, 2016.

[37] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *Proc. of the 2011 IEEE 4th International Conference on Cloud Computing*, CLOUD '11, pages 500–507, 2011.

[38] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. Cloudscale: Elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 5:1–5:14, New York, NY, USA, 2011. ACM.

[39] V. Srinivasan, Brian Bulkowski, Wei-Ling Chu, Sunil Sayyaparaju, Andrew Gooding, Rajkumar Iyer, Ashish Shinde, and Thomas Lopatic. Aerospike: Architecture of a real-time operational DBMS. *Proc. VLDB Endow.*, 9(13):1389–1400, September 2016.

[40] Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Radu Stoica, Bernard Metzler, Nikolas Ioannou, and Ioannis Koltsidas. Crail: A high-performance i/o architecture for distributed data processing. *IEEE Data Engineering Bulletin*, 40(1):38–49, 2017.

[41] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 363–378, Santa Clara, CA, 2016.

[42] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proc. of the European Conference on Computer Systems*, EuroSys'15, Bordeaux, France, 2015.

[43] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2012.

[44] Jake Wires and Andrew Warfield. Mirador: An active control plane for datacenter storage. In *Proc. of the 15th USENIX Conference on File and Storage Technologies*, FAST'17, pages 213–228, 2017.

[45] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, 2010.