λ-KV: Distributed Key-Value store with co-located Serverless compute

Rajath Shashidhara
The University of Texas at Austin

Abstract

By embracing new paradigms such as Serverless computing, cloud resources and applications are more fluid and elastic than ever before. With minimum-to-none infrastructure management, applications can rapidly scale up or down as per demand. While cloud providers benefit from this model as fine-grained resource management and multi-tenancy drives up utilization, applications incur significant overheads as all state is forced to remote storage. We present the design and implementation of λ-*KV*, a serverless execution framework that aggregates compute and storage resources into a single entity that exploits *data locality* to significantly improve efficiency. We argue that by reducing the gap between computation and storage, we can achieve high efficiency without sacrificing resource utilization. Our evaluation shows that λ-KV improves the overall execution time of serverless compilation workloads by a factor of 1.6x and 5x reduction in time spent in data movement against conventional disaggregated deployments.

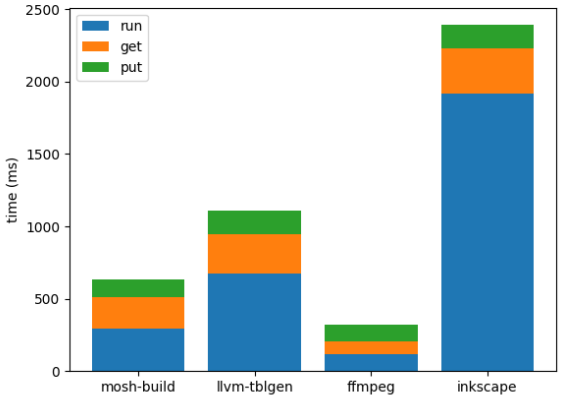

Figure 1: Average Execution profile of compiling 4 popular open-source applications - mosh, llvm, ffmpeg, inkscape using **gg** on AWS Lambda with AWS S3 as the remote storage backend. Fetching and Storing state from S3 comprises $>$ 50% of total execution time for mosh and ffmpeg, $>$ 40% for llvm and $\approx$ 20% for Inkscape.

1 Introduction

Function-as-a-Service (FaaS) platforms have garnered much attention from both academia and industry recently. It is an attractive programming model for the developers as they simply write applications as short-lived stateless functions in standard languages, deploy their code to the cloud without worrying about infrastructure provisioning and management and pay only for the compute resources used when their code is invoked. The platform transparently auto-scales resources in response to workload shifts. Furthermore, stateless code is generally easy to write and debug, can be trivially and dynamically replicated and restarted for purposes of scaling and fault tolerance. Cloud providers benefit as serverless computing drives up utilization due to fine-grained resource management and high-density multi-tenancy.

While the programming model provides nice benefits in terms of resource provisioning and fault-tolerance, it requires all invocations to be isolated and stateless. This limitation forces lambdas to access all data and state from remote storage by moving data over the network repeatedly. Furthermore, while Lambda functions can initiate outbound network connections, they themselves are not directly network-addressable. Function co-ordination and composition is therefore mediated through high-latency low-throughput services object storage, key-value stores or queuing services (S3, DynamoDB, SQS respectively in case of AWS). As observed in [12], Lambdas experience severe I/O bottlenecks as several lambdas are packed on the same VM. [6] reports that the cost of I/O - read+write of 1KB of data to S3 from an AWS Lamdba function is 372x the cost of direct messaging between two AWS EC2 instances using ZeroMQ. Figure 1 demonstrates the cost of externalizing state to remote storage

when compiling large open-source codebases using gg [3] - key-value operations contribute a major portion ($> 50\%$ in some cases) to the total execution time. In addition, clients have no control over the placement of Lambda and there is no "stickiness" for client connections and thereby caching optimizations that exploit data-locality to improve efficiency are impossible. The effect of this *data shipping architecture* is the growing gap between "storage" and "compute" resources in the cloud. The fundamental trade-off made by this paradigm is to increase resource utilization at the cost of efficiency.

Recent proposals in the literature [8, 10, 11] augment the serverless execution framework with low-latency ephemeral stores to store intermediate state to build "stateful" serverless functions. While this optimizes function composition, it doesn't fix the "data-shipping" problem and also gives up on the simplicity of fault-tolerance properties of stateless functions. Systems specifically designed for serverless architecture such as ExCamera [4] are forced work around this limitation by using a coordinator server and thereby compromising scalability. gg [3] also alludes to using a coordinator to optimize the placement of lambdas for efficiency. Alternative ideas from [9, 15] have proposed a low-latency multi-tenant cloud store that allows small units of computation to be performed directly within storage nodes. Although, these solutions can minimize movement of data over the network, computations are limited to a single node. These solutions are insufficient for applications that store large amounts of data distributed over several nodes in the cluster and have severe limitations dealing with scale. State-of-the-art distributed key-value stores [2, 13] shard and replicate data over several replicas to achieve high-throughput, low-latency, fault-tolerance and load balancing. It is limiting to assume that any lambda function accesses data from only one of these replicas.

In this paper, we take the first step towards building a distributed key-value store that has co-located compute resources for accelerated serverless compute. In the sections to follow, we identify the trade-offs required to architect a low-latency scalable serverless execution framework that can run lamdbas written in high-level programming languages. In addition, we outline various techniques such as placement heuristics to minimize data movement over the network. Our experiments demonstrate the promise our design holds in bridging the gap between cloud applications and their data.

2 Motivation and Background

In this section, we survey the workload characteristics of the recent and popular serverless execution frameworks. With this insight at hand, we then examine the various design trade-offs that can be made in building λ-KV.

Our focus is particularly on burst-parallel systems such as gg [3], ExCamera [4] and PyWren [7] that take advantage of cloud-functions platforms to implement low-latency, massively parallel applications. In particular, gg is a powerful abstraction that helps applications use cloud-functions platforms for a broader set of workloads, including irregular execution graphs and ones that change as execution evolves. Several applications including distributed compilation, linear algebra tasks, video encoding and object recognition have been expressed using gg. Therefore, with this generality, we analyze gg to find insights applicable to a broad class of serverless applications.

1. gg expresses jobs in terms of a dependency graph of thunks - hermetically sealed, short-lived containers that may reference the output of other thunks or produce other thunks as output. All data dependencies of a thunk are known apriori and are resolved before execution.

2. Thunks are stateless and fetch all their dependencies from remote storage. As demonstrated by Figure 1, thunks spend a major fraction of their life time interacting with slow, low-throughput cloud storage such as S3, DynamoDB over the network. Minimizing movement of data over the network can result in significant performance improvements.

3. Thunks and other file dependencies are addressed using unique hash values. Key-Value abstraction is suitable for serverless applications interacting with distributed stores.

4. Most input data and the intermediate state generated by thunks is immutable. Distributed storage can relax consistency protocols for such immutable objects to get better performance.

5. Closely-connected thunks in the execution graph exhibit data locality in terms of its dependencies. It must also be observed that state transfer during function composition is intermediated by remote storage. A scheme that optimizes function composition and optimally places the execution of closely related thunks can exploit data locality to improve performance. Furthermore, overlapping the execution phase with communication to remote storage can help hiding latency and improve the utilization of compute resources.

6. Failed thunks and lost objects may be recovered by simply re-executing a portion of the execution graph. Durability is not a main concern for such systems and may be relaxed to gain performance improvements.

We use the above insights as a guide in making the right trade-offs in the design of λ-KV.

3 Design

This section elaborates on the system architecture, design trade-offs and limitations of our implementation.

3.1 Architecture

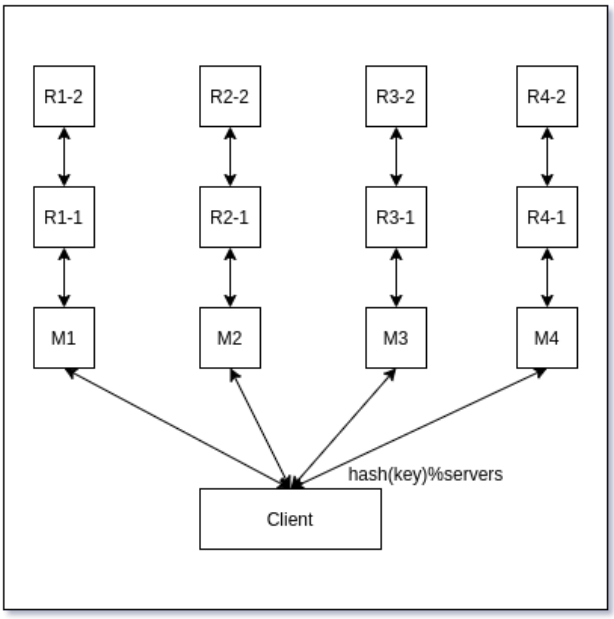

Figure 2: System architecture of λ-KV

Figure 2 depicts the high-level system architecture. A λ-KV cluster consists of several nodes that provide both storage and compute resources to clients. Some nodes are designated as master nodes and each master node is configured with several slave replicas. Nodes are connected to each other and exchange periodic messages to exchange data, detect changes to membership and handle failures.

We elaborate on the salient features of the design in the following sections:

3.1.1 Consistent Hashing

λ-KV uses consistent hashing to shard the data across multiple nodes. By using a hash function to distribute the data, we ensure that data access patterns do not burden any single node and the load is roughly evenly distributed at all times. Similar to DynamoDB [2], the output range of a hash function is treated as a fixed circular space and each node is assigned an even portion of this circular space. Consistent hashing helps in minimizing the number of data transfers triggered by rehashing when new nodes join the system or failure occur. Furthermore, using a consistent hashing scheme helps in significantly reducing the latency as clients can directly communicate with the nodes instead of going through a load-balancer or a meta-data server. Using a consistent hashing scheme also helps clients determine the optimal location for placement of lambda execution based on its data dependencies.

3.1.2 Replication and Fault-tolerance

As depicted in Figure 2, nodes in the cluster follow a master-slave model. Each hash slot has a master node and several replica nodes. Operations are primary handled by the master nodes and are *asynchronously* replicated to slave nodes. Master nodes run a leader election protocol and the leader is responsible for detecting and handling faults. When a majority of nodes determine that a node is unreachable, the leader can direct a slave replica to take over as the master node for the hash slots it is responsible for. Clients when they connect for the first time can query the leader for the current set of master nodes and may cache them in its local memory. Clients may send write requests directly to master nodes and read/execution requests may be directed to both master/slave replicas. Furthermore, when a client is unable to reach a node, it can contact the leader to refresh its local copy of the set of master nodes.

λ-KV is unable to guarantee strong consistency as data is eventually replicated to replicas. Furthermore, we do not provide any durability guarantees in case of failure of master node before it is replicated to the other replica slaves. We trade-off consistency and durability for the sake of performance. We believe that this is the right trade-off to make for a serverless execution framework - Failed executions may be restarted and lost data may be reconstructed by running the execution graph of lamdbas through its lineage [14]. However, the responsibility solely lies on the client to handle failures, inconsistencies and data loss.

3.1.3 Immutable Object Cache

When an execution request is received on a node, all its dependencies must be pooled in before the execution begins. Some of the dependencies may need to be fetched from remote nodes. Similarly, when an execution produces output store requests, some of them may need to be forwarded remote nodes based on the key partitioning dictated by the consistent hash function. Immutable objects present an optimization opportunity to reduce the data movement across the cluster. LRU cache with fixed capacity for remote immutable objects is maintained at each node to exploit the locality patterns in data access. Objects fetched from remote note are inserted into the cache. Infrequently used objects may be replaced based on the LRU policy. Note that cache consistency issues do not arise here because these objects are read-only.

3.1.4 Pipelined Execution

All data dependencies of the lambda execution are declared in the execution request. In our execution model, input and output to lambda are handled by λ-KV. Pipelining these phases can help us overlap computation with communication to make large gains in performance. Clients can pipeline execution requests to λ-KV by making several requests without waiting

for a response. Each node has a small number of executors and a pending execution queue of fixed size. Further, nodes have dedicated threads to setup the dependencies of execution prior to running the program. With this design, we hope to achieve load, execute and store operations concurrently to increase utilization and decrease latency.

3.1.5 Execution Runtime

Lambda functions may be written in C/C++ (high-level programming languages) and built with the lambda runtime to be compatible to run on the system. It is fairly straightforward to transform a program written in any high-level programming language to the interface provided by λ-KV. The runtime takes care of preparing the execution environment, fetching all of the dependencies, provisioning resources and finally executing the lambda handler function. Moreover, runtime ensures that the execution is isolated, secure and does not consume resources over the provisioned capacity.

3.2 Placement Heuristics

Clients are aware of the key partitioning scheme used by λ-KV and are therefore able to direct execution requests to nodes based on heuristics that reduce can potentially reduce the number of remote calls required for data and state transfer. Note that clients may also take into account the caching behaviour of $\lambda - KV$ when making placement decisions - if a recent execution pulled in an immutable remote object on a worker, it is likely that this object will be available in the cache of the worker. In this work, we consider the following heuristics -

Work-Stealing This is a very simple heuristic that places executors on a FIFO queue. First idle executor is assigned to execute the lambda. On receiving the response for execution completion, idle executor is placed at the end of the FIFO queue. This is a baseline heuristic that does not take location of data dependencies into consideration when determining placement.

Most Objects We select the executor the stores most data dependencies by count from the list of idle workers.

Most Objects by Size This is similar to the previous algorithm except that the executors are ranked based on the total size of objects instead of number of objects. While this heuristic is more robust, it also adds an extra overhead of keeping track of size of each object at the client.

Largest Object Pick the worker that stores the largest object from amongst the data dependencies. This heuristic is useful when object size distribution is skewed and the largest object is usually the bottleneck. It may also be the case that very large objects do not fit within the immutable object cache. This is particularly useful in such scenarios.

It is possible to conceive of more sophisticated schemes based on linear programming, query planners and other more recent techniques based on Machine Learning. However, such schemes may impose a computational overhead on the client that might ultimately overwhelm the gains obtained from other architectural features of λ-KV.

3.3 API

Table 3.3 describes the programming interface to λ-KV. `key` is an ASCII string whereas `value` is a byte array and they can be arbitrarily long.

Each request must include a 64-bit identifier generated by the client and this is returned back in the response. A client can issue multiple requests at once without waiting for completion. In fact, pipelining requests increases throughput and is recommended to be used by the client. As previously noted, requests may be executed out-of-order and therefore, the client must use the identifier to associate the response with a request. Generation of the identifier is totally left to the implementation of the client.

A runtime library and header files are provided in order to write new lambda functions. Lamdba functions must be written as C/C++ function as per the prescribed prototype

```
ExecResponse handler(const ExecArgs& args);
```

and compiled into a static binary along with the runtime library. `put()` request with `Execute` argument may be used to store lambda binary code that can be invoked using `execute()`.

Execution requests to lambdas must declare all the data dependencies apriori. The programming model restricts lambdas from performing I/O during execution. Note that the execution request format does not place any restrictions on type, number or name of parameters. Any state or parameters to the function may be encoded in the format prescribed above. Arguments provided to `execute()` are passed to the lambda execution as an `ExecArgs` structure. Output generated by the execution must be encoded as `ExecResponse` structure described below:

```
ExecResponse
    return_code:    Integer
    return_output:  Byte Array
    repeated put_requests {
        key: ASCII String
        value: Byte Array
        permissions: {Read, Write, Execute}
    }
```

Execution may produce multiple `put` requests as a part of the execution response and these are inserted into the store.

API	Parameters	Operation	Output
get	key: ASCII String	Fetch <value>corresponding to <key>from store.	value: Byte Array
put	key: ASCII String value: Byte Array permissions: Bitmask from {Read, Write, Execute} Must be readable.	Store <key, value>pair in store. Also used for Update. If <key>already exists and is read-only, operation will be unsuccessful.	-
delete	key: ASCII String force: Boolean	Delete <key>from store. If <key>is read-only, operation succeeds only if <force=true>.	-
execute	key: ASCII String arguments: Arguments to function immediate: List of ASCII Strings (key, as_file): List of 2-tuples of type (ASCII String, Boolean)	Execute <key>. <immediate>arguments: constant parameters passed to the function. <(key, as_file)>arguments: <value>corresponding to <key>is fetched from store and <(key, value)> is passed as a parameter to the function. If <as_file=true>, <value>is stored in a file and made available to the execution.	return_code: Integer Exit value return_output: Byte Array

Return code and return output are returned to the client as a part of the response.

4 Implementation

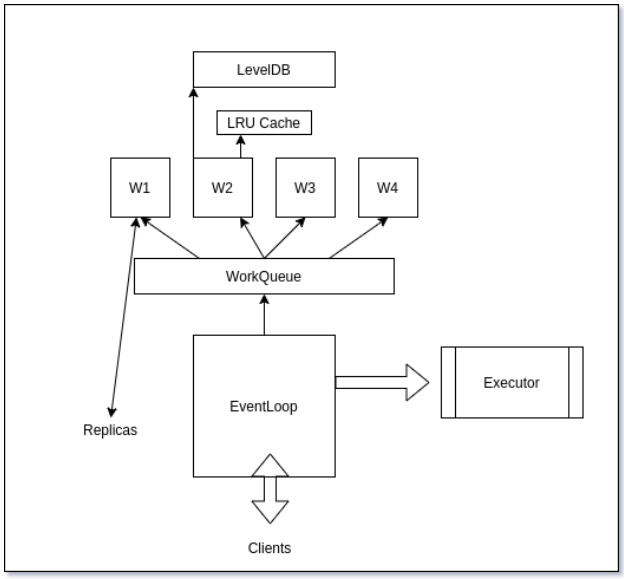

Figure 3: Internal architecture of λ-KV nodes.

4.1 Client Library

Clients connect to λ-KV over a TCP connection. Message formats as described above are implemented using Protobuf. Message formats written in the Protobuf Description Language may be compiled into any programming language headers/packages using the Protobuf compiler. Requests are sent over the connection by first writing the size of the serialized message followed by the serialized message. Similarly, responses are sent by λ-KV as protobuf messages. Using protobuf gives us the flexibility of using any programming language to interface with λ-KV as the protobuf compiler and no special code is required otherwise. Furthermore, a Protobuf stream parser for C++ is also implemented and is available as a library and clients may use this library to encode/decode protobuf messages both over blocking or non-blocking TCP connections.

Key-space is partitioned using consistent hashing depending on the number of available nodes in the cluster. Clients directly talk to the node instead of going through a load balancer. We use CRC16 as the consistent hash function similar to Redis cluster deployments [1].

Furthermore, the client library provides functions for placement heuristics described in Section 3.2 above. Clients have the ability to extend or combine the heuristic functions or supply their own heuristic for placement of execution request.

4.2 λ-KV Nodes

On application startup, a main thread and several worker threads are spawned. Each worker thread connects to the other replicas via a TCP connection using the same mechanism as a client. A single asynchronous event loop implemented using the Linux `epoll()` mechanism runs in the main thread and serves the TCP connections to clients. As requests are received from clients, the main thread parses the requests from the TCP stream and they are enqueued on a work queue. Idle worker threads steal work from the work queue, process the request and generate a response. Main thread is notified of work completion through a callback and it serializes the response over the TCP connection. In this design, the main thread is always I/O bound and CPU bound tasks are offloaded to worker threads.

LevelDB - a high-performance embedded key-value store [5] is used as the storage backend. LevelDB already provides serializability across many different concurrent operations from the workers. In addition, an in-memory Least Recently Used (LRU) cache of fixed size is used to cache immutable objects fetched from remote nodes. Access to the LRU is synchronized via coarse-grained synchronization using a single mutex. More efficient implementations using fine-grained optimistic concurrency are possible but not attempted in this implementation.

For every key-value pair, we store a small amount of metadata in the storage backend to store the access permissions. When a node receives a request with a key outside of its keyspace domain, the request is rejected and an error code is returned. In addition, a put request on an immutable key is rejected. A delete operation on a read-only key without the force flag is also rejected. When a forced delete on a read-only key is received, the request is forwarded to every node in the cluster so that it can be removed from the object cache. Note that this process happens asynchronously and the client should not assume that the object has been deleted from the cache at the other nodes when it receives a success response to the delete operation. Furthermore, no guarantees are provided in the event of failures and we rely on the LRU mechanism to clean-up objects that are unused for a long time. Immutable objects are only to be used when the application is certain that the value does not change during the life cycle of the application. When immutable objects are force deleted, it is assumed that there are no active consumers to this object any longer.

When an execution request is received, worker thread prepares the execution environment. First, the input to the lambda is gathered from local store/cache and remote nodes. Then, input required as files specified in the execution arguments are copied into a temporary directory. After that, an idle executor is then chosen to execute the function. The executor invokes the lambda function using `fork()`, passes the input parameters through a UNIX pipe. Runtime built into the lambda binary parses the input and invokes the lambda handler function. Output generated by the lambda handler function is communicated back by the runtime to the main thread on the UNIX pipe. Any `put()` operations generated by the execution are handled by a worker thread and a response is generated. Lambdas are executed as Linux processes and we rely on the CPU and Memory virtualization provided by the operating system to isolate the process. The execution runtime also ensures that the lambda execution is hermetically sealed i.e. program is not allowed to perform any network I/O and interact only with a small set of dependency files bound at runtime. The executed program is traced using `ptrace()` and any any illegal system calls are caught and the execution is terminated. Stronger isolation and more secure execution mechanisms using Chrome V8 engine or Containers may be implemented in future versions of the implementation. Furthermore, a runtime based on WebAssembly may be used to support several high-level languages in the future.

4.3 gg Storage Backend + Execution Engine

A storage backend for `gg` is built using the client library described above. Uploading and downloading files from λ-KV are implemented using multiple threads - each thread connects to a single replica and the operations are batched to achieve maximum performance. Execution engine also uses the client library to connect λ-KV. Number of nodes and number of pipelined jobs per node may be configured. Furthermore, the heuristics described above are implemented and may be configured to optimize lambda placement.

5 Evaluation

In our evaluation setup, we use a cluster of 12 Google Cloud VM Instances - 8 of type `n1-standard-1` and 4 of type `n1-standard-2`.[1] Each `n1-standard-1` machine houses 1 vCPU (implemented as a single hardware Hyperthread on Intel Skylake architecture) with 3.75GB of memory. `n1-standard-2` is a more powerful machine with 2 vCPUs and 7.5GB of memory. Further, each instance has 250GB of SSD backed persistent storage. These machines run Ubuntu 18.04 LTS with GCP optimized Linux kernel version 5.3.0-1018-gcp.

We setup the baseline as a dis-aggregated compute and storage cluster. Storage nodes run bare-bones λ-KV without the execution runtime and simply acts as a distributed key-value store. Compute nodes work as a `gg-remote` execution servers. Compute nodes are configured to execute lambdas in a purely stateless mode without caching state on local storage. Baseline storage and compute nodes run on nodes with

[1] Google Cloud Platform restricts free accounts from running more than 8 vCPUs simultaneously. Tests using more than 8 nodes was not possible without spending significant amount of money to run the VMs. Therefore, we only consider upto 8 nodes in all our tests.

`n1-standard-1` configuration. Since each of these machines have a single vCPU, there is no advantage in running more than one executor at the same time. Therefore, a $N-$node compute cluster provides N-way parallelism in job execution. We run the λ-KV serverless execution framework on four `n1-standard-2` nodes each with 2 vCPUs. This is necessary for a fair comparison because each node also runs an executor along with the key-value service. Both the storage nodes and λ-KV nodes are backed by LevelDB instance as described in Section 4.2. LevelDB durably stores the records on secondary storage and is also configured with a 100 MB in-memory cache. Note that the above mentioned cache is only for local objects and is not the same as the Immutable object cache described earlier. Furthermore, we do not configure any replica nodes in this setup.

Our evaluation workload includes compiling mosh-shell and llvm - two popular open-source applications using `gg` on λ-KV clusters. As described in Section 2, gg breaks down compilation into a collection of thunks and an execution graph. At the heart of gg is a Directed Acyclic Graph (DAG) scheduler that can execute thunks in parallel on multiple execution engines. As thunks are executed, dependencies are further resolved in the execution graph and more thunks may be executed. This process continues until the desired output file is generated. Mosh-shell is a very small code base and has just 88 thunks. On the other hand, LLVM is at least 10x larger than mosh-shell with 715 thunks to be resolved. Both of these workloads have high density of dependency between different thunks and are therefore suitable for modelling execution workloads with dominant KV-operations cost.

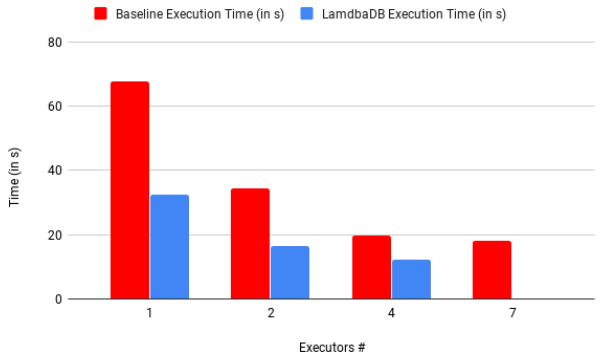

Figure 4: Comparison between Baseline and LamdbaDB: Total execution time when building mosh-shell using gg. Each have the same number of storage and executors configured as indicated by the X axis

Figures 4 and 5 plot the time taken to compile mosh and llvm respectively using both baseline and λ-KV execution engines.[2] Both the execution engines are configured with

[2]LambdaDB could not be executed with 7 executors on a free GCP account for reasons stated previously.

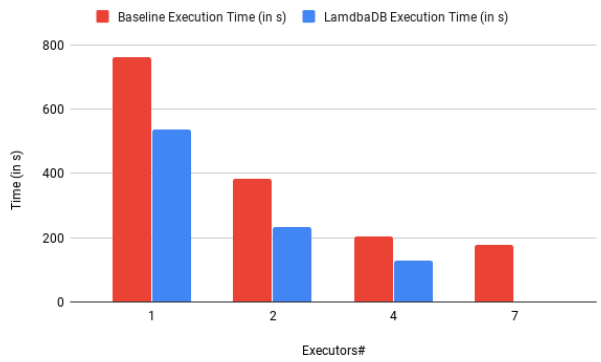

Figure 5: Comparison between Baseline and LamdbaDB: Total execution time when building llvm using gg. Each have the same number of storage and executors configured as indicated by the X axis

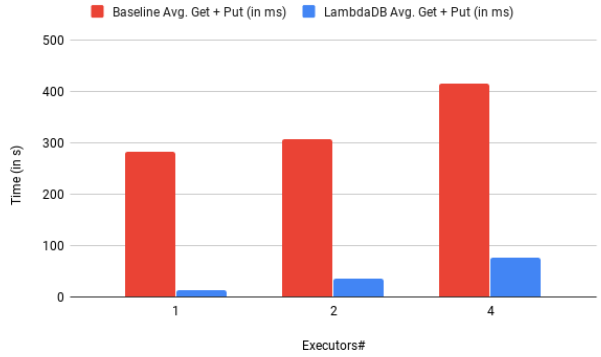

Figure 6: Comparison between Baseline and LamdbaDB: Average latency incurred on KV-operations per Lambda when building mosh-shell using gg. Each have the same number of storage and executors configured as indicated by the X axis

equal number of storage and compute nodes for comparison. However, the major difference is that storage and compute are co-located on the same node in λ-KV. In addition, λ-KV is also configured with 100MB of immutable object cache and execution is pipelined with a count of 2 - i.e. at most one job is enqueued at each node in the pipeline queue when an execution is ongoing. We observe 1.6x improvement in overall execution time with 4 executors. In order to explain this, we examine the average time spent in KV-operations by the lamdba functions in Figures 6 and 7 for mosh and llvm respectively. $\lambda - DB$ reduces the time spent in get or put operations by a factor of 5x with 4 nodes and up to 20x for a single node! Comparing with the data in Figure 1 where see that the fraction of get + put to execution has dropped to

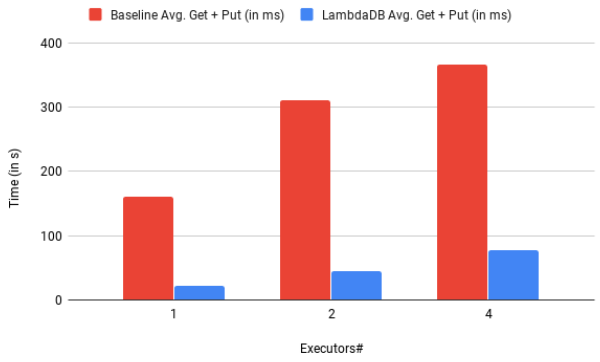

Figure 7: Comparison between Baseline and LamdbaDB: Average latency incurred on KV-operations per Lambda when building llvm using gg. Each have the same number of storage and executors configured as indicated by the X axis

$\approx$ 20% drastically bringing the overall execution time.

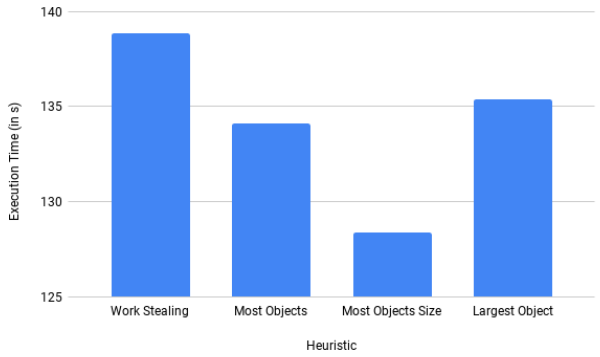

Figure 8: Comparison of different placement heuristics with respect to total execution time.

Next, we compare the performance of different placement heuristics. In this experiment, we compile llvm over a 4-node λ-KV cluster with 100M of immutable object cache with a pipeline factor of 2 as described above. As seen in Figure 8, we observe a significant reduction in execution time when scheduling decisions are taken based on the location of the objects in the cluster. Most Objects By Size yields the best results with a 7.5% gain over the naive work stealing algorithm. Other heuristics also show a notice gain of about 3% over the baseline.

In the next experiment, we analyze the effect of cache size on the average KV-operations latency per lambda. We setup a similar 4-node cluster with varying cache size and compile llvm on this cluster. Pipelining is disabled in this experiment. Figure 9 shows that as the cache size is gradually increased, we see a steep reduction in the overheads. Beyond a certain

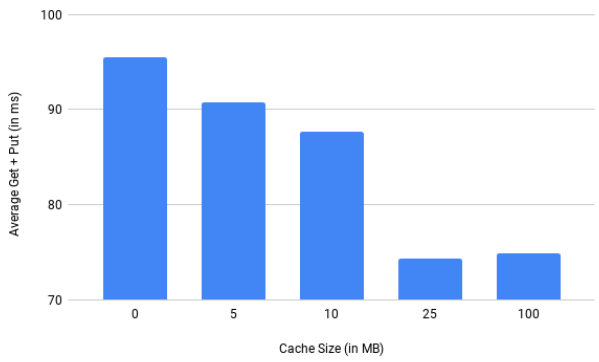

Figure 9: Latency of KV-operations as a function of Immutable Object Cache size. As cache size increases, the need to fetch data from remote servers dramatically decreases.

cache size, no further improvements are seen as we have reached a situation where we have fully exploited any locality available in the workload.

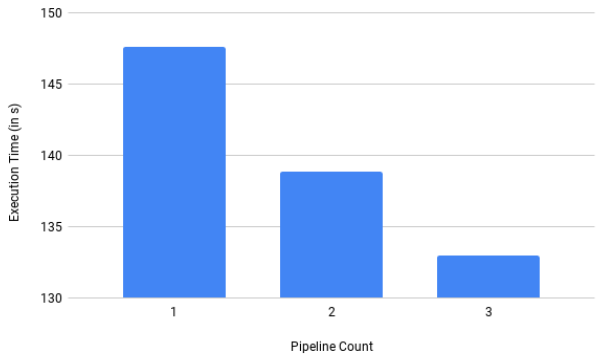

Figure 10: Number of pipelined jobs at each worker compared with total execution time.

Next experiment depicted in Figure 10 shows the effect of pipelining on total exection time. Notice that we could not experiment beyond pipelining level of 3 on our setup because each of our node has only 2 vCPUs. We will not see any benefits by increasing the pipelining further due to this. With a small modification to the design by using separate event loops in worker threads to fetch remote objects, the pipelining factor may be increased beyond the current limit and this may be implemented in the future.

6 Acknowledgments

I would like to thank Prof. Simon Peter for his suggestions. His insights were crucial in helping me crystallize the ideas

behind this work. I would also like to thank the research group at Stanford behind `gg` for choosing to open-source the project.

7 Source Code

Source code for **λ-KV** is publicy available and open-source at https://github.com/rajathshashidhara/SimpleDB.
Furthermore, our fork of **gg** is also available on github: https://github.com/rajathshashidhara/gg.

8 Conclusion

Present cloud deployments avoid coupling compute and storage to ease scaling and fault-tolerance and to drive high-utilization. Applications incur significant overheads due to this ever widening gap. λ-KV attempts to bridge this gap by devising an architecture that allows applications to embed functions within the key-value store. λ-KV aggressively attempts to reduce data movement across the network by making it possible to execute lambda functions directly on the storage nodes. Together with the placement heuristics, pipelining and caching, λ-KV demonstrates orders of magnitude improvement over conventional deployments.

References

[1] Redis cluster: Documentation.

[2] 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. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007.

[3] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *2019 USENIX Annual Technical Conference (ATC 19)*, pages 475–488, 2019.

[4] 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 *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 363–376, 2017.

[5] Sanjay Ghemawat and Jeff Dean. Leveldb: fast key-value storage library.

[6] Joseph M Hellerstein, Jose Faleiro, Joseph E Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back. *arXiv preprint arXiv:1812.03651*, 2018.

[7] 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*, pages 445–451, 2017.

[8] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, 2018.

[9] Chinmay Kulkarni, Sara Moore, Mazhar Naqvi, Tian Zhang, Robert Ricci, and Ryan Stutsman. Splinter: Bare-metal extensions for multi-tenant low-latency storage. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 627–643, 2018.

[10] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 193–206, 2019.

[11] Vikram Sreekanti, Chenggang Wu Xiayue Charles Lin, Jose M Faleiro, Joseph E Gonzalez, Joseph M Hellerstein, and Alexey Tumanov. Cloudburst: Stateful functions-as-a-service. *arXiv preprint arXiv:2001.04592*, 2020.

[12] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 133–146, 2018.

[13] Sage A Weil, Andrew W Leung, Scott A Brandt, and Carlos Maltzahn. 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*, pages 35–44, 2007.

[14] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[15] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. Narrowing the gap between serverless and its state with storage functions. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–12, 2019.