Occupy the Cloud: Distributed Computing for the 99%

Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, Benjamin Recht
University of California, Berkeley
{jonas, qifan, shivaram, istoica, brecht}@eecs.berkeley.edu

ABSTRACT

Distributed computing remains inaccessible to a large number of users, in spite of many open source platforms and extensive commercial offerings. While distributed computation frameworks have moved beyond a simple map-reduce model, many users are still left to struggle with complex cluster management and configuration tools, even for running simple embarrassingly parallel jobs. We argue that stateless functions represent a viable platform for these users, eliminating cluster management overhead, fulfilling the promise of elasticity. Furthermore, using our prototype implementation, PyWren, we show that this model is general enough to implement a number of distributed computing models, such as BSP, efficiently. Extrapolating from recent trends in network bandwidth and the advent of disaggregated storage, we suggest that stateless functions are a natural fit for data processing in future computing environments.

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**; • **Computing methodologies** → *Distributed programming languages*;

KEYWORDS

Serverless, Distributed Computing, AWS Lambda, PyWren

ACM Reference Format:
Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, Benjamin Recht University of California, Berkeley {jonas, qifan, shivaram, istoica, brecht}@eecs.berkeley.edu. 2017. Occupy the Cloud: Distributed Computing for the 99%. In *Proceedings of SoCC '17, Santa Clara, CA, USA, September 24–27, 2017,* 7 pages.
https://doi.org/10.1145/3127479.3128601

1 INTRODUCTION

Despite a decade of availability, the twin promises of scale and elasticity [2] remain out of reach for a large number of cloud computing users. Academic and commercially-successful platforms (Apache Hadoop, Apache Spark) with tremendous corporate backing (Amazon, Microsoft, Google) still present high barriers to entry for the average data scientist or scientific computing user. In fact, taking advantage of elasticity remains challenging for even sophisticated users, as the majority of these frameworks were designed to first target on-premise installations at large scale. On commercial cloud platforms, a novice user confronts a dizzying array of potential decisions: one must ahead of time decide on instance type, cluster size, pricing model, programming model, and task granularity.

Such challenges are particularly surprising considering that the vast number of data analytic and scientific computing workloads remain embarrassingly parallel. Hyperparameter tuning for machine learning, Monte Carlo simulation for computational physics, and featurization for data science all fit well into a traditional map-reduce framework. Yet even at UC Berkeley, we have found via informal surveys that the majority of machine learning graduate students have never written a cluster computing job due to complexity of setting up cloud platforms.

In this paper we argue that a *serverless* execution model with *stateless* functions can enable radically-simpler, fundamentally elastic, and more user-friendly distributed data processing systems. In this model, we have one simple primitive: users submit functions that are executed in a remote container; the functions are stateless as all the state for the function, including input, output is accessed from shared remote storage. Surprisingly, we find that the performance degradation from using such an approach is negligible for many workloads and thus, our simple primitive is in fact general enough to implement a number of higher-level data processing abstractions, including MapReduce and parameter servers.

Recently cloud providers (e.g., AWS Lambda, Google Cloud Functions) and open source projects (e.g., OpenLambda [16], OpenWhisk [31]) have developed infrastructure to run event-driven, stateless functions as micro-services. In this model, a function is deployed once and is invoked repeatedly whenever new inputs arrive and elastically scales with input size. Our key insight is that we can dynamically inject code into these functions, which combined with remote storage, allows us to build a data processing system that inherits the elasticity of the serverless model while addressing the simplicity for end users.

We describe a prototype system, PyWren[1], developed in Python with AWS Lambda. By employing only stateless functions, PyWren helps users avoid the significant developer and management overhead that has until now been a necessary prerequisite. The complexity of state management can instead be captured by a global scheduler and fast remote storage. With PyWren, we seek to understand the trade-offs of using stateless functions for large scale data analytics and specifically what is the impact of solely using remote storage for inputs and outputs. We find that we can achieve around 30-40 MB/s write and read performance per core to a remote bulk object store (S3), matching the per-core performance of a single local SSD on typical EC2 nodes. Further we find that this scales to 60-80 GB/s to S3 across 2800 simultaneous functions, showing that existing remote storage systems may not be a significant bottleneck.

[1] PyWren is available at https://pywren.io

SoCC '17, September 24–27, 2017, Santa Clara, CA, USA

ACM ISBN 978-1-4503-5028-0/17/09…$15.00
https://doi.org/10.1145/3127479.3128601

Using this as a building block we implement image processing pipelines where we extract per-image features during a map phase via unmodified Python code. We also show how we can implement BSP-style applications on PyWren and that a word count job on 83M items is only 17% slower than PySpark running on dedicated servers. Shuffle-intensive workloads are also feasible as we show PyWren can sort 1TB data in 3.4 minutes. However, we do identify storage throughput as a major bottleneck for larger shuffles. Finally we discuss how parameter servers, a common construct in distributed ML [23] can be used with this model. We conclude the paper with some remaining systems challenges, including launch overhead, storage performance and scalable scheduling.

2 IS THE CLOUD USABLE?

The advent of elastic computing has greatly simplified access to computing resources, as the complexity of management is now handled by cloud providers. Thus the complexity has now shifted to applications or programming frameworks. However most software, especially in scientific and analytics applications, is not written by computer scientists [18, 26], and it is many of these users who have been left out of the cloud revolution.

The layers of abstraction present in distributed data processing platforms are complex and difficult to correctly configure. For example, PySpark, arguably one of the easier to use platforms, runs on top of Spark [49] (written in Scala) which interoperates and is closely coupled with HDFS [42] (written in Java), Yarn [46] (Java again), and the JVM. The JVM in turn is generally run on virtualized Linux servers. Merely negotiating the memory limit interplay between the JVM heap and the host operating system is an art form [10, 44, 45]. These systems often promote "ease of use" by showing powerful functionality with a few lines of code, but this ease of use means little without mastering the configuration of the layers below.

In addition to the software configuration issues, cloud users are also immediately faced with tremendous planning and workload management before they even begin running a job. AWS offers 70 instances types across 14 geographical datacenters – all with subtly different pricing. This complexity is such that recent research has focused on algorithmic optimization of workload trade-offs [17, 47]. While several products such as Databricks and Qubole simplify cluster management, the users still need to explicitly start and terminate clusters, and pick the number and type of instances.

Finally, the vast majority of scientific workloads could take advantage of dynamic market-based pricing of servers, such as AWS spot instances – but computing spot instance pricing is challenging, and additionally most of the above-mentioned frameworks make it difficult to handle machine preemption. To avoid the risk of losing intermediate data, users must be careful to either regularly checkpoint their data or run the master and a certain number of workers on non-spot instances. This adds another layer of management complexity which makes elasticity hard to obtain in practice.

What users want: Our proposal in this paper was motivated by a professor of computer graphics at UC Berkeley asking us "Why is there no cloud button?" He outlined how his students simply wish they could easily "push a button" and have their code – existing, optimized, single-machine code – running on the cloud. Thus, our fundamental goal here is to allow as many users as possible to take existing, legacy code and run it in parallel, exploiting elasticity. In an ideal world, users would simply be able to run their desired code across a large number of machines, bottlenecked only by serial performance. Executing 100 or 10000 five-minute jobs should take roughly five minutes, with minimal start-up and tear-down overhead.

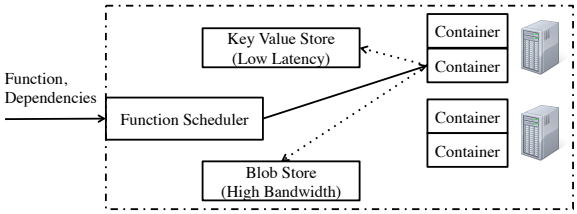

Figure 1: System architecture for stateless functions

Further, in our experience far more users are capable of writing reasonably-performant single-threaded code, using numerical linear algebra libraries (e.g., OpenBLAS, Intel's MKL), than writing complex distributed-systems code. Correspondingly the goal for these users is not to get the best parallel performance, but rather to get vastly better performance than available on their laptop or workstation while taking *minimal development time*.

For compute-bound workloads, it becomes more useful to parallelize across functions for many cases; to say sweep over a wide range of parameters (such as machine learning hyperparameter optimization) or try a large number of random initial seeds (Monte Carlo simulations of physical systems). In these cases, exposing function-level parallelism is more worthwhile than having complex interfaces for intra-function optimization. Therefore, a simple function interface that captures sufficient local state, performs computation remotely, and returns the result is more than adequate. For data-bound workloads, a large number of users would be served by a simpler version of the existing map-reduce framework where outputs can be easily persisted on object storage.

Thus, a number of compute-bound and data-bound workloads can be captured by having a simple abstraction that allows users to run arbitrary functions in the cloud without setting up and configuring servers/frameworks etc. We next discuss why such an abstraction is viable now and the components necessary for such a design.

3 A MODEST PROPOSAL

Many of the problems with current cloud computing abstractions stem from the fact that they are designed for a server-oriented resource model. Having servers as the unit of abstraction ties together multiple resources like memory, CPU and network bandwidth. Further servers are also often long running and hence require DevOps support for maintenance. Our proposal is to instead use a serverless architecture with *stateless functions* as the unifying abstraction for data processing. Using stateless functions will simplify programming and deployment for end users. In this section we present the high level components for designing data processing systems on a serverless architecture. While other proposals [4] have looked at implementing data processing systems on serverless infrastructure, we propose a simple API that is tightly integrated with existing libraries and also study performance trade-offs of this approach by using our prototype implementation on a number of workloads.

Table 1: Comparison of single-machine write bandwidth to instance local SSD and remote storage in Amazon EC2. Remote storage is faster than single SSD on the standard `c3.8xlarge` instance and the storage-optimized `i2.8xlarge` instance.

Storage Medium	Write Speed (MB/s)
SSD on `c3.8xlarge`	208.73
SSD on `i2.8xlarge`	460.36
4 SSDs on `i2.8xlarge`	1768.04
S3	501.13

3.1 Systems Components

The main components necessary for executing stateless functions include a low overhead execution runtime, a fast scheduler and high performance remote storage as shown in Figure 1. Users submit single-threaded functions to a global scheduler and while submitting the function they can also annotate the runtime dependencies required. Once the scheduler determines where a function is supposed to run, an appropriate container is created for the duration of execution. While the container maybe reused to improve performance none of the state created by the function will be retained across invocations. Thus, in such a model all the inputs to functions and all output from functions need to be persisted on remote storage and we include client libraries to access both high-throughput and low latency shared storage systems.

Fault Tolerance: Stateless functions allow simple fault tolerance semantics. When a function fails, we restart it (at possibly a different location) and execute on the same input. We only need atomic writes to remote storage for tracking which functions have succeeded. Assuming that functions are idempotent we obtain similar fault tolerance guarantees as existing systems.

Simplicity: As evidenced by our discussion above, our architecture is very simple and only consists of the minimum infrastructure required for executing functions. We do not include any distributed data structures or dataflow primitives in our design. We believe that this simplicity is necessary in order to make simple workloads like embarrassingly parallel jobs easy to use. More complex abstractions like dataflow or BSP can be implemented on top and we discuss this in Section 3.3.

Why now? The model described above is closely related to systems like Linda [6], Celias [15] and database trigger-based systems [34, 35]. While their ideas are used in work-stealing queues and shared file system, the specific programming model has not been widely adopted. We believe that this model is viable now given existing infrastructure and technology trends. While the developer has no control of where a stateless function runs (e.g., the developer cannot specify that a stateless function should run on the node storing the function's input), the benefits of colocating computation and data – a major design goal for prior systems like Hadoop, Spark and Dryad – have diminished.

Prior work has shown that hard disk locality does not provide significant performance benefits [12]. To see whether the recent datacenter migration from hard disks to SSDs has changed this conclusion, we benchmarked the I/O throughput of storing data on a local SSD of an AWS EC2 instance vs. storing data on S3. Our results, in Table 1, show that currently that writing to remote storage is faster than a single SSD but using multiple SSDs can yield better performance. However, technology trends [9, 14, 41] indicate that the gap between network bandwidth and storage I/O bandwidth is narrowing, and many recently published proposals for rack-scale computers feature disaggregated storage [3, 20] and even disaggregated memory [13]. All these trends suggest diminishing performance benefits from colocating compute with data in the future.

3.2 PyWren: A Prototype

We developed PyWren[2] to rapidly evaluate these ideas, seamlessly exposing a map primitive from Python on top of AWS Lambda. While Lambda was designed to run event-driven microservices (such as resizing a single user-uploaded image) with a fixed funtion, by extracting new code from S3 during runtime we make each Lambda invocation run a different function. Currently AWS Lambda provides a very restricted containerized runtime with a maximum 300 seconds of execution time, 1.5 GB of RAM, 512 MB of local storage and no root access, but we believe these limits will be increased as AWS Lambda is used for more general purpose applications.

PyWren serializes a Python function using `cloudpickle` [7], capturing all relevant information as well as most modules that are not present in the server runtime[3]. This eliminates the majority of user overhead about deployment, packaging, and code versioning. We submit the serialized function along with each serialized datum by placing them into globally unique keys in S3, and then invoke a common Lambda function. On the server side, we invoke the relevant function on the relevant datum, both extracted from S3. The result of the function invocation is serialized and placed back into S3 at a pre-specified key, and job completion is signaled by the existence of this key. In this way, we are able to reuse one registered Lambda function to execute different user Python functions and mitigate the high latency for function registration, while executing functions that exceed Lambda's code size limit.

Map for everyone: As discussed in Section 2, many scientific and analytic workloads are embarrassingly parallel. The `map` primitive provided by PyWren makes addressing these use cases easy – serializing all local state necessary for computation, transparently invoking functions remotely and returning when complete. Calling `map` launches as many stateless functions as there are elements in the list that one is mapping over. An important aspect to note here is that this API mirrors the existing Python API for parallel processing and thus, unlike other serverless MapReduce frameworks [4], this integrates easily with existing libraries for data processing and visualization.

Microbenchmarks: Using PyWren we ran a number of benchmarks(Figures 2,3,4) to determine the impact of solely using remote storage for IO, and how this scales with worker count. In terms of compute, we ran a matrix multiply kernel within each Lambda and find that we get 18 GFLOPS per core and that this unsurprisingly scales to more than 40 TFLOPS while using 2800 workers. To measure remote I/O throughput we benchmarked the read, write bandwidth to S3 and our benchmarks show that we can get on average 30 MB/s write and 40 MB/s read per Lambda and that this

[2] A wren is much smaller than a Condor

[3] While there are limitations in the serialization method (including an inability to transfer arbitrary Python C extensions), we find this can be overcome using libraries from package managers such as Anaconda.

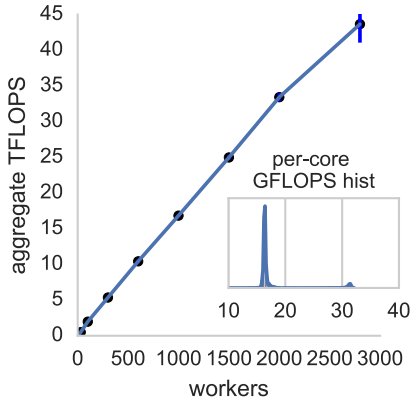

Figure 2: Running a matrix multiplication benchmark inside each worker, we see a linear scalability of FLOPs across 3000 workers.

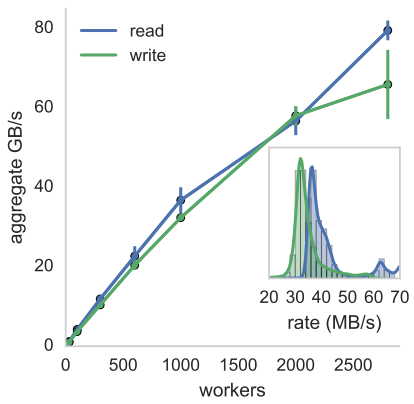

Figure 3: Remote storage on S3 linearly scales with each worker getting around 30 MB/s bandwidth (inset histogram).

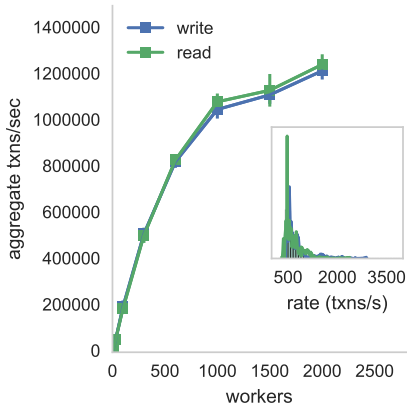

Figure 4: Remote key-value operations to Redis scales up to 1000 workers. Each worker gets around 700 synchronous transactions/sec.

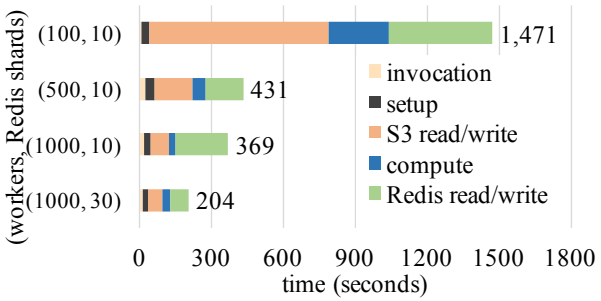

Figure 5: Performance breakdown for sorting 1TB data by how task time is spent on average.

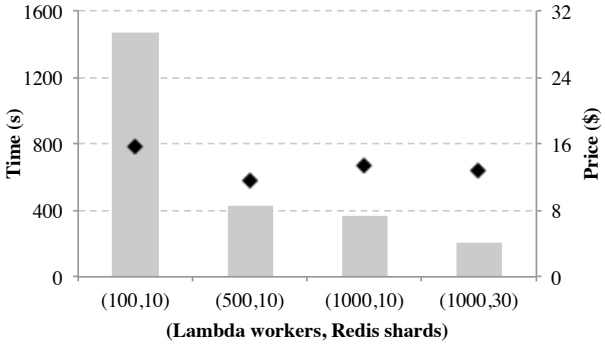

Figure 6: Prorated cost and performance for running 1TB sort benchmark while varying the number of Lambda workers and Redis shards.

also scales to more than 60 GB/s write and 80 GB/s read. Assuming that 16 such Lambdas are as powerful as a single server, we find that the performance from Lambda matches the S3 performance shown in Table 1. To measure the overheads for small updates, we also benchmarked 128-byte synchronous put/gets to two `c3.8xlarge` instances running in-memory Redis. We match the performance reported in prior benchmarks [37] and get less than 1ms latency up to 1000 workers.

Applications: In our research group we have had students use PyWren for applications as diverse as computational imaging, scientific instrument design, solar physics, and object recognition. Working with heliphysicists at NASA's Solar Dynamics Observatory, we have used PyWren for extracting relevant features across 16TB of solar imaging data for solar flare prediction. Working with applied physics colleagues, we have used PyWren to design novel types of microscope point-spread functions for 3d superresolution microscopy. This necessitates rapid and repeat evaluation of a complex physics-based optical model inside an inner loop.

3.3 Generality for the rest of us ?

While the map primitive in PyWren covers a number of applications, it prohibits any coordination among the various tasks. We next look at how stateless functions along with high performance storage can also be used as a flexible building block to develop more complex abstractions.

Map + monolithic Reduce The first abstraction we consider is one where output from all the map operations is collected on to one machine (similar to `gather` in HPC literature) for further processing. We find this pattern covers a number of classical machine learning workloads which consist of a featurization (or ETL) stage that converts large input data into features and then a learning stage where the model is built using SVMs or linear classifiers. In such workloads, the featurization requires parallel processing but the generated features are often small and fit on a single large machine [5]. These applications can be implemented using a *map* that runs using stateless functions followed by a learning stage that runs on a single multi-core server using efficient multi-core libraries [28]. The wide array of machine choices in the cloud means that this approach can handle learning problems with features up to 2TB in size [48].

As an example application we took off-the-shelf image featurization code [8] and performed cropping, scaling, and GIST image featurization [29] of the 1.28M images in the ImageNet LargeScale Visual Recognition Challenge [39]. We run the end-to-end featurization using 3000 workers on AWS Lambda. and store the features on S3. This takes 113 seconds and following that we run a monolithic

Table 2: Time taken for featurization and classification

phase	mean	std
lambda start latency	9.7s	29.1s
lambda setup time	14.2s	5.2s
featurization	112.9s	10.2s
result fetch	22.0s	10.0s
fit linear classifier	4.3s	0.5s

reduce on a single `r4.16xlarge` instance. Fetching the features from S3 to this instance only takes 22s and building a linear classifier using NumPy and Intel MKL libraries takes 4.3s. Thus, we see that this model is a good fit where a high degree of parallelism is initially required to do ETL / featurization but a single node is sufficient (and most efficient [25]) for model building.

MapReduce: For more general purpose coordination, a commonly used programming model is the bulk-synchronous processing (BSP) model. To implement the BSP model, in addition to parallel task execution, we need to perform data shuffles across stages. The availability of high-bandwidth remote storage provides an natural mechanism to implement such shuffles. Using S3 to store shuffle data, we implemented a word count program in PyWren. On the Amazon reviews [24] dataset consisting of 83.68M product reviews split across 333 partitions, this program took 98.6s. We ran a similar program using PySpark. Using 85 `r3.xlarge` instances, each having 4 cores to match the parallelism we had with PyWren, the Spark job took 84s. The slow down is from the lack of parallel shuffle block reads in PyWren and some stragglers while writing/reading from S3. Despite that we see that PyWren is only around 17% slower than Spark and our timings do not include the 5-10 minutes it takes to start the Spark instances.

We also run the Daytona sort benchmark [43] on 1TB input, to see how PyWren handles a shuffle-intensive workload. We implemented the Terasort [30] algorithm to perform sort in two stages: a partition stage that range-partitions the input and writes out to intermediate storage, and a merge stage that, for each partition, merges and sorts all intermediate data for that partition and writes out the sorted output. Due to the resource limitation on each Lambda worker, we need at least 2500 tasks for each stage. This results in 2500^2, or 6,250,000 intermediate files (each 160Kb) to shuffle in between. While S3 does provide abundant I/O bandwidth to Lambda for this case, it is not designed to sustain high request rate for small objects. Also as S3 is a multi-tenant service, there is an imposed limit on request throughput per S3 bucket for the benefit of overall availability. Therefore, we use S3 only for storing input and writing final output, and deploy a Redis cluster with `cache.m4.10xlarge` nodes for intermediate storage.[4] Figure 5 shows the end-to-end performance with varying numbers of concurrent Lambda workers and Redis shards, with breakdown of task time. We see that higher level of parallelism does greatly improve job performance (up to 500 workers) until Redis throughput becomes a bottleneck. From 500 to 1000 workers, the Redis I/O time increases by 42%. Fully leveraging this parallelism requires more Redis shards, as shown by the 44% improvement with 30 shards. Interestingly, adding more resources does not necessarily increase total cost due to the reduction in latency with scale (Figure 6).[5] Supporting a larger sort, e.g., 100TB, does become quite challenging, as the number of intermediate files increases quadratically. We plan to investigate more efficient solutions.

Parameter Servers: Finally using low-latency, high throughput key-value stores like Redis, RAMCloud [38] we can also implement parameter-server [1, 23] style applications in PyWren. For example, we can implement HOGWILD! stochastic gradient descent by having each function compute the gradients based on the latest version of shared model. Since the only coordination across functions happens through the parameter server, such applications fit very well into the stateless function model. Further we can use existing support for server-side scripting [36] in key value stores to implement features like range updates and flexible consistency models [23]. However, currently this model is not easy to use as unlike S3, the ElasticCache service requires users to select a cache server type and capacity. To deploy more performant parameter servers [23] that go beyond simple key-value store would involve more complexity, e.g., setting up on a EC2 cluster or requiring a new hosted service, leaving the economic implications for further investigation.

4 DISCUSSION

While we studied the performance provided by existing infrastructure in the previous section, there are a number of systems aspects that need to be addressed to enable high performance data processing.

Resource Balance: One of the primary challenges in a serverless design is in how a function's resource usage is allocated and as we mentioned in §3.2, the existing limits are quite low. The fact that the functions are stateless and need to transfer both input and output over the network can help cloud providers come up with some natural heuristics. For example if we consider the current constraints of AWS Lambda we see that each Lambda has around 35 MB/s bandwidth to S3 and can thus fill up its memory of 1.5GB in around 40s. Assuming it takes 40s to write output, we can see that the running time of 300s is appropriately proportioned for around 80s of I/O and 220s of compute. As memory capacity and network bandwidths grow, this rule can be used to automatically determine memory capacity given a target running time.

Pricing The simplicity of elastic computing comes with a premium that the users pay to the cloud providers. At the time of writing Lambda is priced at ~$0.06 per GB-hour of execution, measured in 100ms-increments. Lambda is thus only ~2× more expensive than on-demand instances. This cost premium seems worthwhile given substantially finer-grained billing, much greater elasticity, and the fact that many dedicated clusters are often running at 50% utilization. Another benefit that stems from PyWren's disaggregated architecture is that cost estimation or even cost prediction becomes much simpler. In the future we plan to explore techniques that can automatically predict the cost of a computation.

Scalable Scheduling: A number of cluster scheduling papers [21, 32, 33, 40] have looked at providing low latency scheduling for data parallel frameworks running on servers. However, to implement such scheduling frameworks on top of stateless functions, we need

[4] Redis here can be replaced by any other key-value store, e.g., memcached, as we were only using the simple set/get API.

[5] Lambda bills in 100ms increments. Redis is charged per hour and is prorated here to seconds per CloudSort benchmark rules [43].

to handle the fact that information about the cluster status (i.e., which containers are free, input locations, resource heterogeneity) is only available to the infrastructure provider, while the structure of the job (i.e. how functions depend on each other) is only available to the user. In the future we plan to study what information needs to exposed by cloud providers and if scheduling techniques like offers [19] can handle this separation.

Debugging: Debugging can be a challenge as PyWren is composed of multiple system components. For monitoring Lambda execution we rely on service tools provided by AWS. For example, CloudWatch saves off-channel logs from Lambda workers which can be browsed through a cloud-viewer. S3 is another place to track as it contains metadata about the execution. To understand a job execution comprehensively, e.g., caclulating how much time is spent at each stage, however, would require tools to align events from both the host and Lambda.

Distributed Storage: With the separation of storage and compute in the PyWren programming model, a number of performance challenges translate into the need for more efficient distributed storage systems. Our benchmarks in §3.2 showed the limitations of current systems, especially for supporting large shuffle-intensive workloads, and we plan to study how we can enable a flat-datacenter storage system in terms of latency and bandwidth [27]. Further, our existing benchmarks also show the limitation of not lacking API support for append in systems like S3 and we plan to develop a common API for storage backends that power serverless computation.

Launch Overheads: Finally one of the main drawbacks in our current implementation is that function invocation can take up to 20-30 seconds (~10% of the execution time) without any caching. This is partly due to lambda invocation rate limits imposed by AWS and partly due to the time taken to setup our custom Python runtime. We plan to study if techniques used to make VM forks cheaper [22], like caching containers or layering filesystems can be used to improve latency. We also plan to see if the scheduler can be modified to queue functions before their inputs are ready to handle launch overheads.

Other Applications: While we discussed data analytics applications that fit well with the serverless model, there are some applications that do not fit today. Applications that use specialized hardware like GPUs or FPGAs are not supported by AWS Lambda, but we envision that more general hardware support will be available in the future. However, for applications like particle simulations, which require a lot of coordination between long running processes, the PyWren model of using stateless functions with remote storage might not be a good fit. Finally, while we primarily focused on existing analytics applications in this paper, the serverless model has also been used successfully in other domains like video compression [11].

5 CONCLUSION

The server-oriented focus of existing data processing systems in the cloud presents a high barrier for a number of users. In this paper we propose that using stateless functions with remote storage, we can build a data processing system that inherits the elasticity, simplicity of the serverless model while providing a flexible building block for more complex abstractions.

ACKNOWLEDGEMENT

We want to thank Vaishaal Shankar for his contribution to the project, the various anonymous reviewers and our shepherd Ymir Vigfusson for their insightful comments and suggestions. This research is supported in part by DHS Award HSHQDC-16-3-00083, NSF CISE Expeditions Award CCF-1139158, and gifts from Ant Financial, Amazon Web Services, CapitalOne, Ericsson, GE, Google, Huawei, Intel, IBM, Microsoft and VMware. EJ and BR are generously supported by ONR award N00014-17-1-2401 and research grants from Amazon. EJ is additionally supported by a grant from Microsoft. BR is also generously supported by NSF award CCF-1359814, ONR awards N00014-14-1-0024 and N00014-17-1-2191, a Sloan Research Fellowship, and a Google Faculty Award.

REFERENCES

[1] ABADI, M., BARHAM, P., CHEN, J., CHEN, Z., DAVIS, A., DEAN, J., DEVIN, M., GHEMAWAT, S., IRVING, G., ISARD, M., ET AL. Tensorflow: A system for large-scale machine learning. In *OSDI* (2016).

[2] ARMBRUST, M., FOX, A., GRIFFITH, R., JOSEPH, A. D., KATZ, R., KONWINSKI, A., LEE, G., PATTERSON, D., RABKIN, A., STOICA, I., ET AL. A view of cloud computing. *CACM 53*, 4 (2010), 50–58.

[3] ASANOVIC, K., AND PATTERSON, D. Firebox: A hardware building block for 2020 warehouse-scale computers. In *FAST* (2014).

[4] Serverless Reference Architecture: MapReduce. https://github.com/awslabs/lambda-refarch-mapreduce.

[5] CANNY, J., AND ZHAO, H. Big data analytics with small footprint: Squaring the cloud. In *KDD* (2013).

[6] CARRIERO, N., AND GELERNTER, D. Linda in context. *CACM 32*, 4 (Apr. 1989).

[7] cloudpickle: Extended pickling support for python objects. https://github.com/cloudpipe/cloudpickle.

[8] DOUZE, M., JÉGOU, H., SANDHAWALIA, H., AMSALEG, L., AND SCHMID, C. Evaluation of gist descriptors for web-scale image search. In *ACM International Conference on Image and Video Retrieval* (2009).

[9] IEEE P802.3ba, 40Gb/s and 100Gb/s Ethernet Task Force. http://www.ieee802.org/3/ba/.

[10] FANG, L., NGUYEN, K., XU, G., DEMSKY, B., AND LU, S. Interruptible tasks: Treating memory pressure as interrupts for highly scalable data-parallel programs. In *SOSP* (2015).

[11] FOULADI, S., WAHBY, R. S., SHACKLETT, B., BALASUBRAMANIAM, K. V., ZENG, W., BHALERAO, R., SIVARAMAN, A., PORTER, G., AND WINSTEIN, K. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *NSDI* (2017).

[12] G. ANANTHANARAYANAN, A. GHODSI, S. SHENKER, I. STOICA. Disk-Locality in Datacenter Computing Considered Irrelevant. In *Proc. HotOS* (2011).

[13] GAO, P. X., NARAYAN, A., KARANDIKAR, S., CARREIRA, J., HAN, S., AGARWAL, R., RATNASAMY, S., AND SHENKER, S. Network requirements for resource disaggregation. In *OSDI* (2016).

[14] HAN, S., EGI, N., PANDA, A., RATNASAMY, S., SHI, G., AND SHENKER, S. Network support for resource disaggregation in next-generation datacenters. In *HotNets* (2013).

[15] HAN, S., AND RATNASAMY, S. Large-scale computation not at the cost of expressiveness. In *HotOS* (2013).

[16] HENDRICKSON, S., STURDEVANT, S., HARTER, T., VENKATARAMANI, V., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Serverless computation with OpenLambda. In *HotCloud* (2016).

[17] HERODOTOU, H., LIM, H., LUO, G., BORISOV, N., DONG, L., CETIN, F. B., AND BABU, S. Starfish: A self-tuning system for big data analytics. In *CIDR* (2011).

[18] HETTRICK, S., ANTONIOLETTI, M., CARR, L., CHUE HONG, N., CROUCH, S., DE ROURE, D., EMSLEY, I., GOBLE, C., HAY, A., INUPAKUTIKA, D., JACKSON, M., NENADIC, A., PARKINSON, T., PARSONS, M. I., PAWLIK, A., PERU, G., PROEME, A., ROBINSON, J., AND SUFI, S. Uk research software survey 2014. https://doi.org/10.5281/zenodo.14809, Dec. 2014.

[19] HINDMAN, B., KONWINSKI, A., ZAHARIA, M., GHODSI, A., JOSEPH, A., KATZ, R., SHENKER, S., AND STOICA, I. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proc. NSDI* (2011).

[20] HP The Machine: Our vision for the Future of Computing. https://www.labs.hpe.com/the-machine.

[21] ISARD, M., PRABHAKARAN, V., CURREY, J., WIEDER, U., TALWAR, K., AND GOLDBERG, A. Quincy: Fair Scheduling for Distributed Computing Clusters. In *Proc. SOSP* (2009), pp. 261–276.

[22] LAGAR-CAVILLA, H. A., WHITNEY, J. A., SCANNELL, A. M., PATCHIN, P., RUMBLE, S. M., DE LARA, E., BRUDNO, M., AND SATYANARAYANAN, M. Snowflock: Rapid virtual machine cloning for cloud computing. In *EuroSys* (2009).

[23] LI, M., ANDERSEN, D. G., PARK, J. W., SMOLA, A. J., AHMED, A., JOSIFOVSKI, V., LONG, J., SHEKITA, E. J., AND SU, B.-Y. Scaling distributed machine learning with the parameter server. In *OSDI* (2014).

[24] MCAULEY, J., TARGETT, C., SHI, Q., AND VAN DEN HENGEL, A. Image-based recommendations on styles and substitutes. In *SIGIR* (2015).

[25] MCSHERRY, F., ISARD, M., AND MURRAY, D. G. Scalability! but at what COST? In *HotOS* (2015).

[26] MOMCHEVA, I., AND TOLLERUD, E. Software Use in Astronomy: an Informal Survey. *arXiv 1507.03989* (2015).

[27] NIGHTINGALE, E. B., ELSON, J., FAN, J., HOFMANN, O., HOWELL, J., AND SUZUE, Y. Flat datacenter storage. In *OSDI* (2012).

[28] NIU, F., RECHT, B., RE, C., AND WRIGHT, S. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS* (2011).

[29] OLIVA, A., AND TORRALBA, A. Modeling the shape of the scene: A holistic representation of the spatial envelope. *International Journal of computer vision 42*, 3 (2001), 145–175.

[30] O'MALLEY, O. TeraByte Sort on Apache Hadoop. http://sortbenchmark.org/YahooHadoop.pdf.

[31] OpenWhisk. https://developer.ibm.com/openwhisk/.

[32] OUSTERHOUT, K., PANDA, A., ROSEN, J., VENKATARAMAN, S., XIN, R., RATNASAMY, S., SHENKER, S., AND STOICA, I. The case for tiny tasks in compute clusters. In *HotOS* (2013).

[33] OUSTERHOUT, K., WENDELL, P., ZAHARIA, M., AND STOICA, I. Sparrow: distributed, low latency scheduling. In *SOSP* (2013).

[34] PENG, D., AND DABEK, F. Large-scale incremental processing using distributed transactions and notifications. In *OSDI* (2010).

[35] POWER, R., AND LI, J. Piccolo: Building fast, distributed programs with partitioned tables. In *OSDI* (2010).

[36] Redis server side scripting. https://redis.io/commands/eval.

[37] Redis benchmarks. https://redis.io/topics/benchmarks.

[38] RUMBLE, S. M., ONGARO, D., STUTSMAN, R., ROSENBLUM, M., AND OUSTERHOUT, J. K. It's Time for Low Latency. In *Proc. HotOS* (2011).

[39] RUSSAKOVSKY, O., DENG, J., SU, H., KRAUSE, J., SATHEESH, S., MA, S., HUANG, Z., KARPATHY, A., KHOSLA, A., BERNSTEIN, M., BERG, A. C., AND LI, F.-F. ImageNet Large Scale Visual Recognition Challenge. *IJCV 115*, 3 (2015), 211–252.

[40] SCHWARZKOPF, M., KONWINSKI, A., ABD-EL-MALEK, M., AND WILKES, J. Omega: flexible, scalable schedulers for large compute clusters. In *Proc. EuroSys* (2013).

[41] SCOTT, C. Latency trends. http://colin-scott.github.io/blog/2012/12/24/latency-trends/.

[42] SHVACHKO, K., KUANG, H., RADIA, S., AND CHANSLER, R. The Hadoop Distributed File System. In *Mass storage systems and technologies (MSST)* (2010).

[43] Sort Benchmark. http://sortbenchmark.org.

[44] Tuning Java Garbage Collection for Apache Spark Applications. https://goo.gl/SIWlqx.

[45] Tuning Spark. https://spark.apache.org/docs/latest/tuning.html#garbage-collection-tuning.

[46] VAVILAPALLI, V. K., MURTHY, A. C., DOUGLAS, C., AGARWAL, S., KONAR, M., EVANS, R., GRAVES, T., LOWE, J., SHAH, H., SETH, S., ET AL. Apache Hadoop YARN: Yet another resource negotiator. In *SoCC* (2013).

[47] VENKATARAMAN, S., YANG, Z., FRANKLIN, M., RECHT, B., AND STOICA, I. Ernest: Efficient performance prediction for large-scale advanced analytics. In *NSDI* (2016).

[48] X1 instances. https://aws.amazon.com/ec2/instance-types/x1/.

[49] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M., SHENKER, S., AND STOICA, I. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proc. NSDI* (2011).