SJTU SAIL (Software Architecture and Infrastructure Lab)

SHANGHAI JIAO TONG UNIVERSITY

Serverless Ephemeral Storage

Yalun Lin Hsu

2020-12-14

SAIL

USENIX ATC 2018

Understanding Ephemeral Storage for Serverless Analytics

Ana Klimovic*, Yawen Wang*, Christos Kozyrakis*, Patrick Stuedi+ , Jonas Pfefferle+ , Animesh Trivedi+

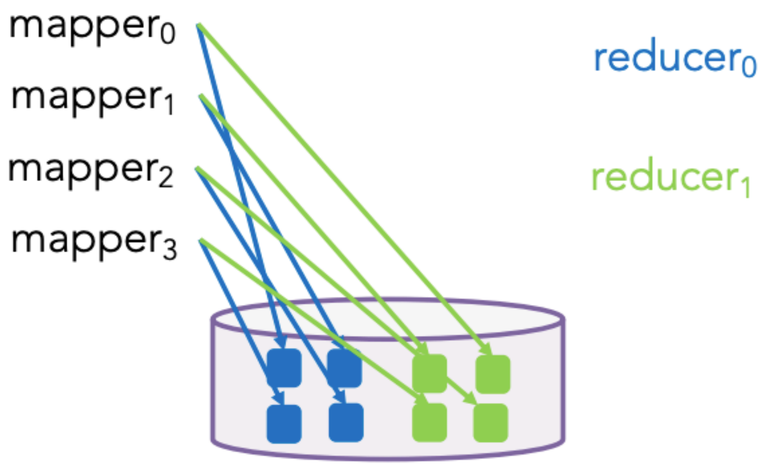
https://www.usenix.org/conference/atc18/presentation/klimovic-serverless

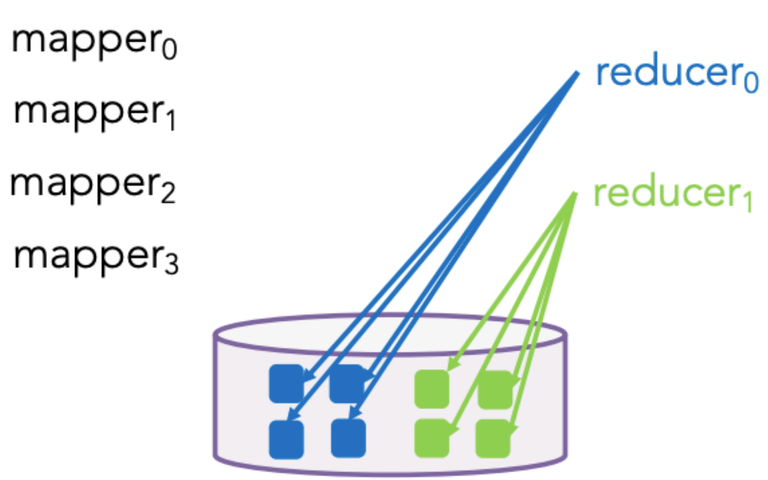
Why Need Ephemeral Storage

- Real applications(Analytics jobs) involve multiple stages of execution

- Serverless tasks need an efficient way to communicate **intermediate data** between different stages of execution

ephemeral data





Questions

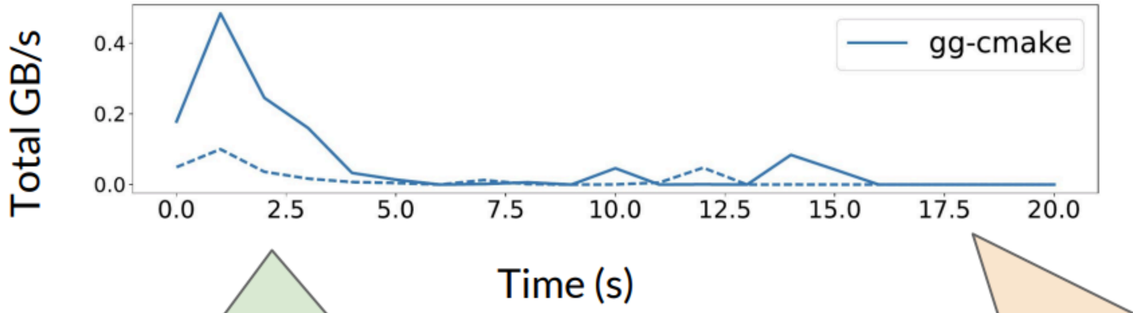
1. What are the ephemeral I/O characteristics of serverless analytics applications?
2. How do applications perform using existing systems (e.g., S3, Redis) for ephemeral I/O?
3. What storage media (DRAM, Flash, HDD) satisfies I/O requirements at the lowest cost?

1. Application Ephemeral I/O Patterns

Application Type

Distributed Compilation

Ephemeral I/O Throughput: Write (dotted), Read (solid)



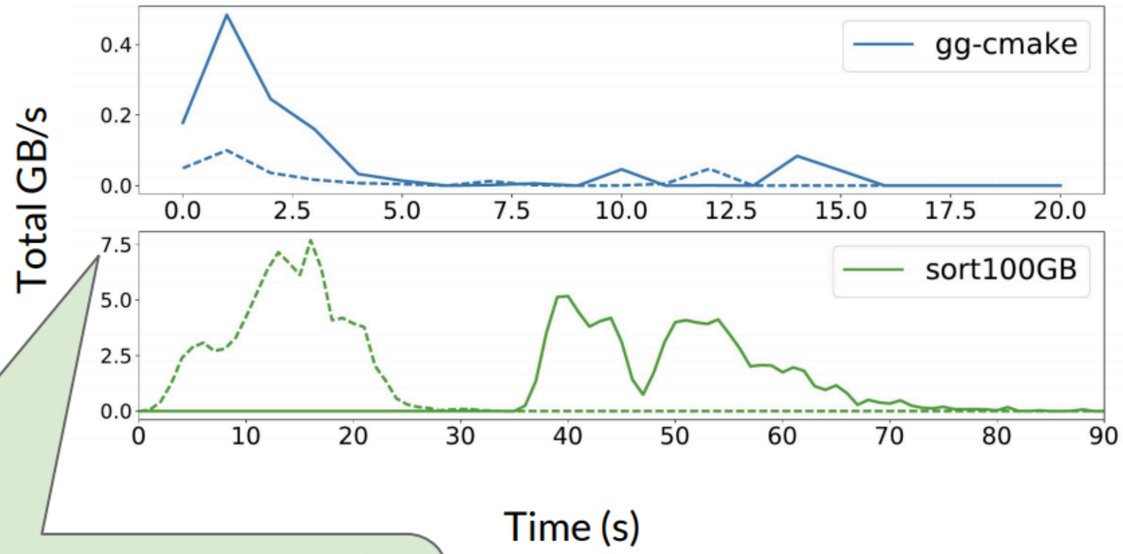
Ephemeral Data Capacity

0.85 GB

High throughput and IOPS due to high parallelism: lambdas each compile independent files

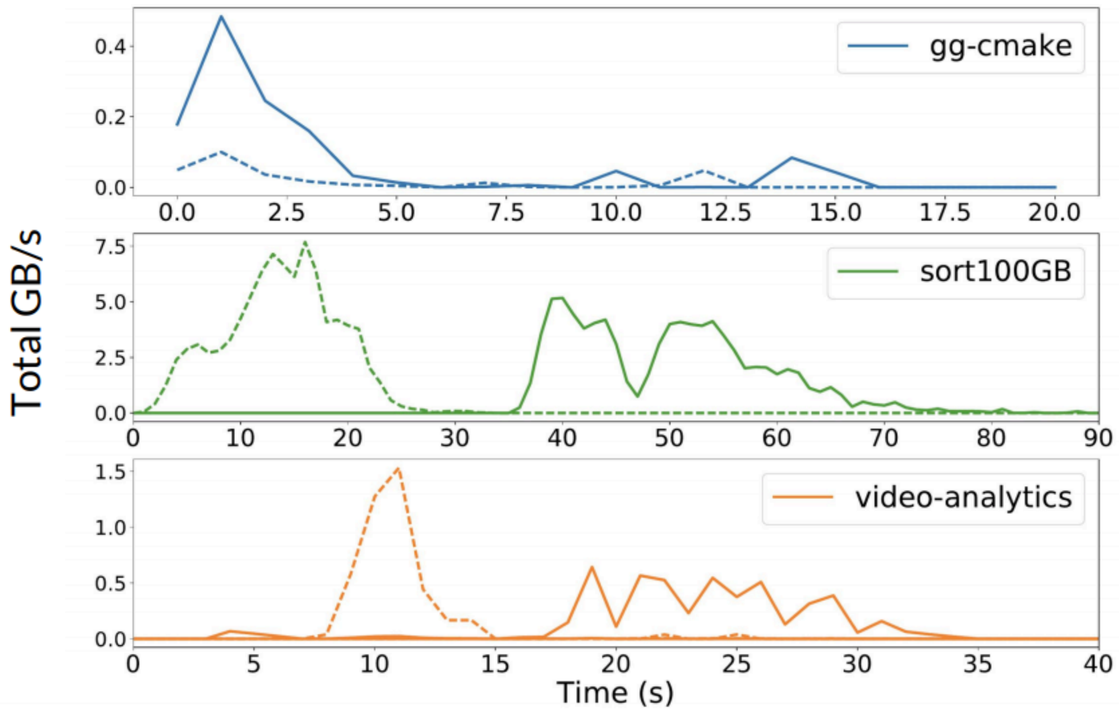
Archiving and linking lambdas are serialized as they depend on previous lambdas → low parallelism, low I/O rate

1. Application Ephemeral I/O Patterns

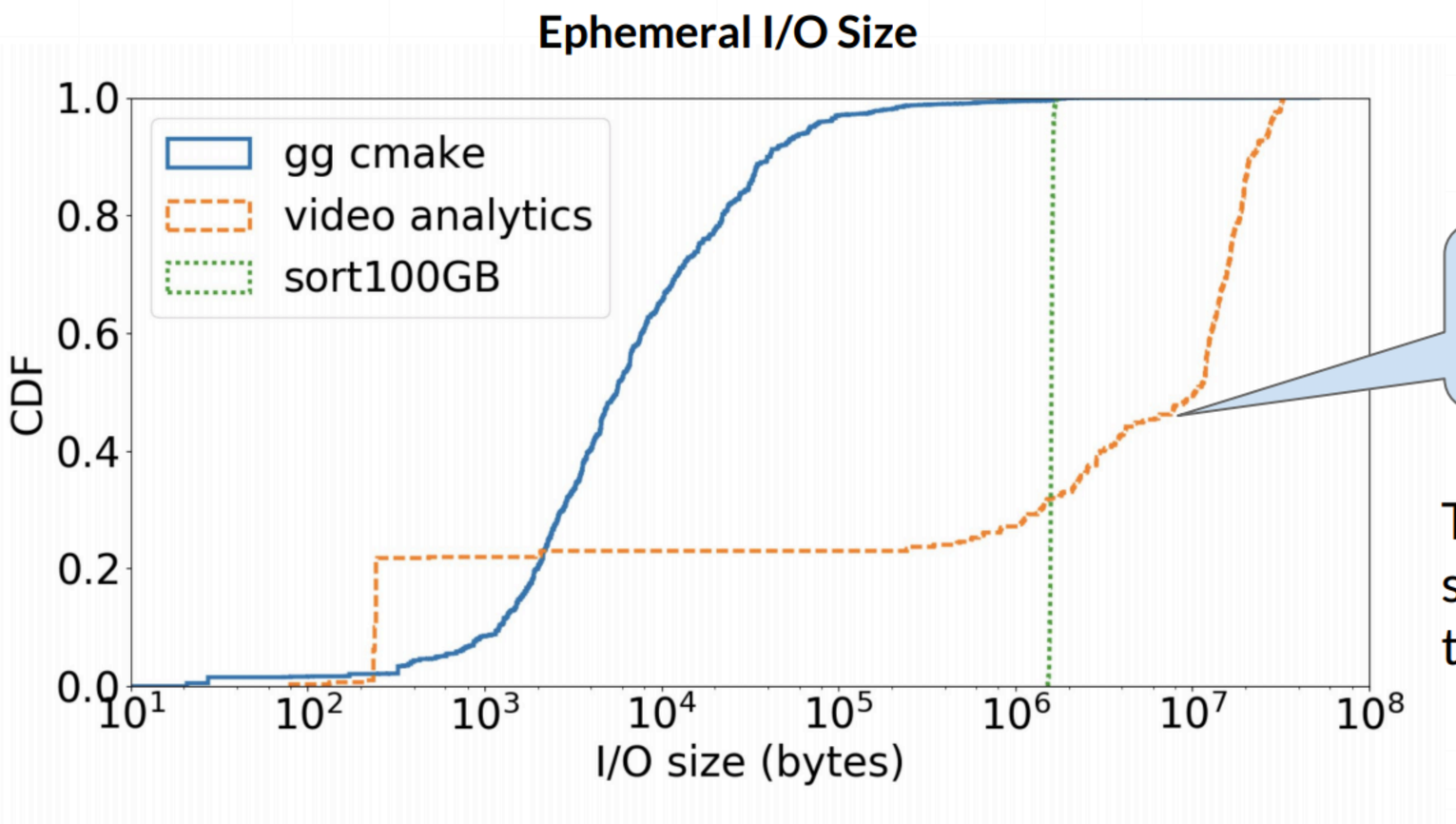
Application Type	Ephemeral I/O Throughput: Write (dotted), Read (solid)	Ephemeral Data Capacity
Distributed Compilation		0.85 GB
MapReduce		100 GB

High throughput due to high I/O intensity and parallelism (up to 7.5 GB/s with 500 concurrent lambdas)

1. Application Ephemeral I/O Patterns

Application Type	Ephemeral I/O Throughput: Write (dotted), Read (solid)	Ephemeral Data Capacity
Distributed Compilation		0.85 GB
MapReduce		100 GB
Video Analytics		6 GB

1. Application Ephemeral I/O Patterns



Wide range of I/O sizes (bytes to 100s of MBs)

Thus, an ephemeral storage system should support high throughput ***and*** low latency.

2. Existing Storage Systems

Cloud object storage system (e.g. Amazon S3)

- Pay only for the capacity and throughput you use
- Resources managed by cloud provider

In-memory key-value store (e.g. Redis)

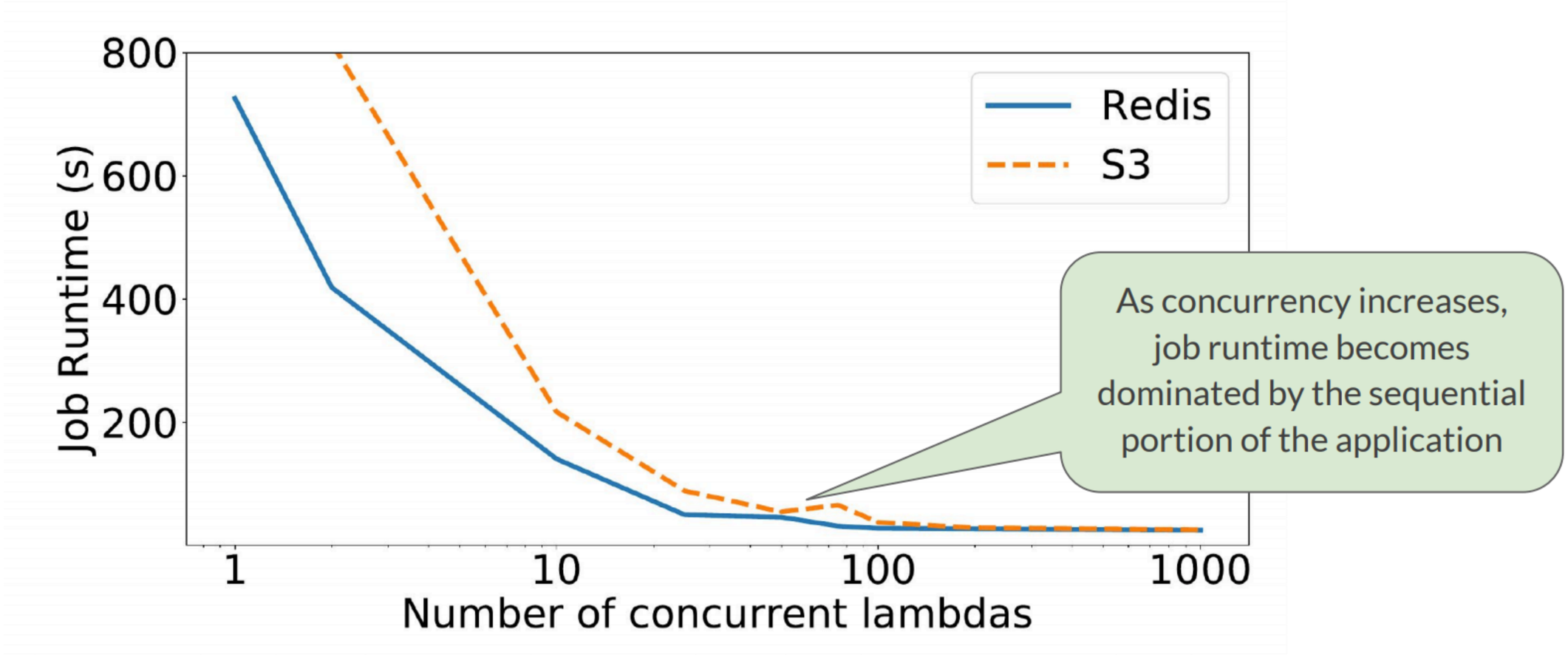
- High performance at the higher cost of DRAM
- Manually select and scale storage instance

Distributed Flash-based data store (e.g. Crail-ReFlex)

- Use Flash for high bandwidth at lower cost
- Manually select and scale storage instances

Latency sensitivity

Distributed compilation job shows some sensitivity to latency due to small I/Os



The impact of application parallelism

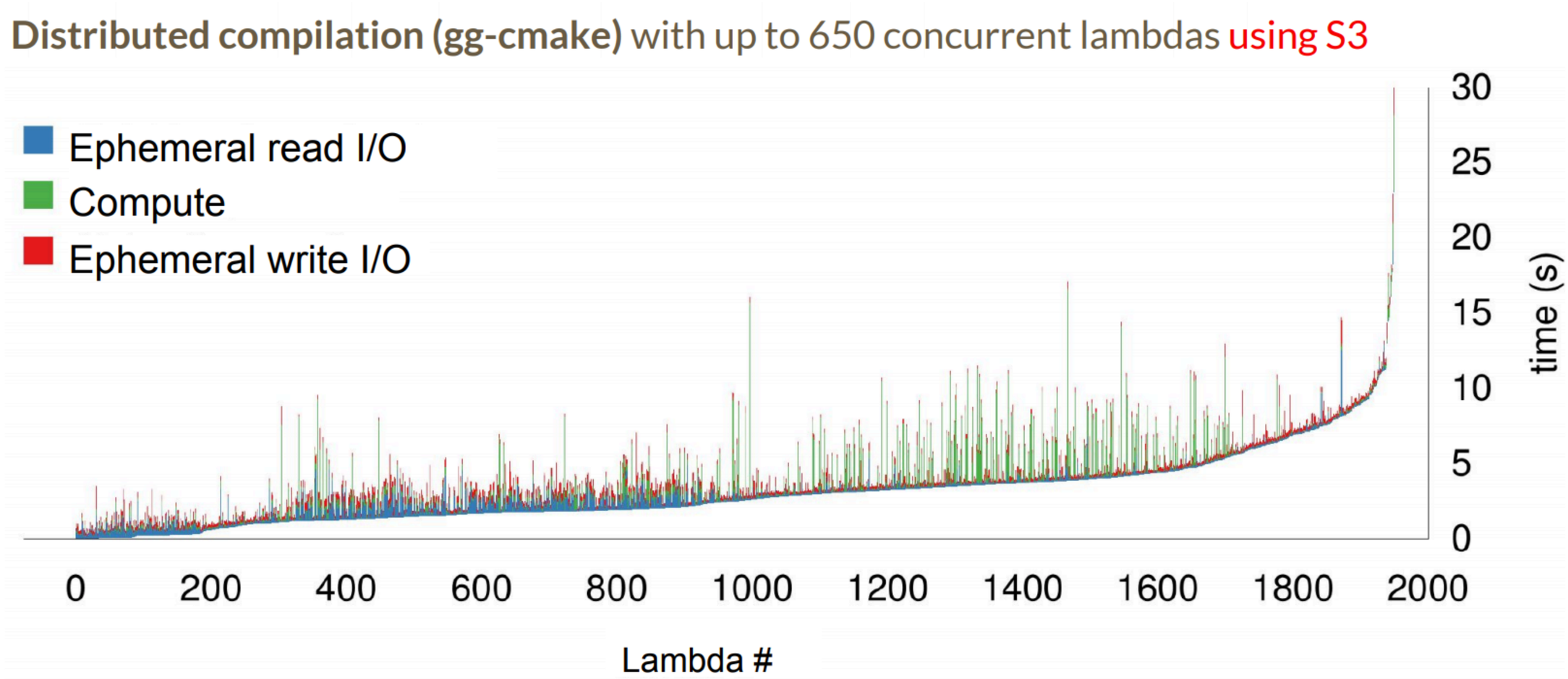


Figure based on Fig. 6 in "A thunk to remember: make -j1000 (and other jobs) on functions-as-a-service infrastructure (preprint)." Fouladi, S., et al.

The impact of application parallelism

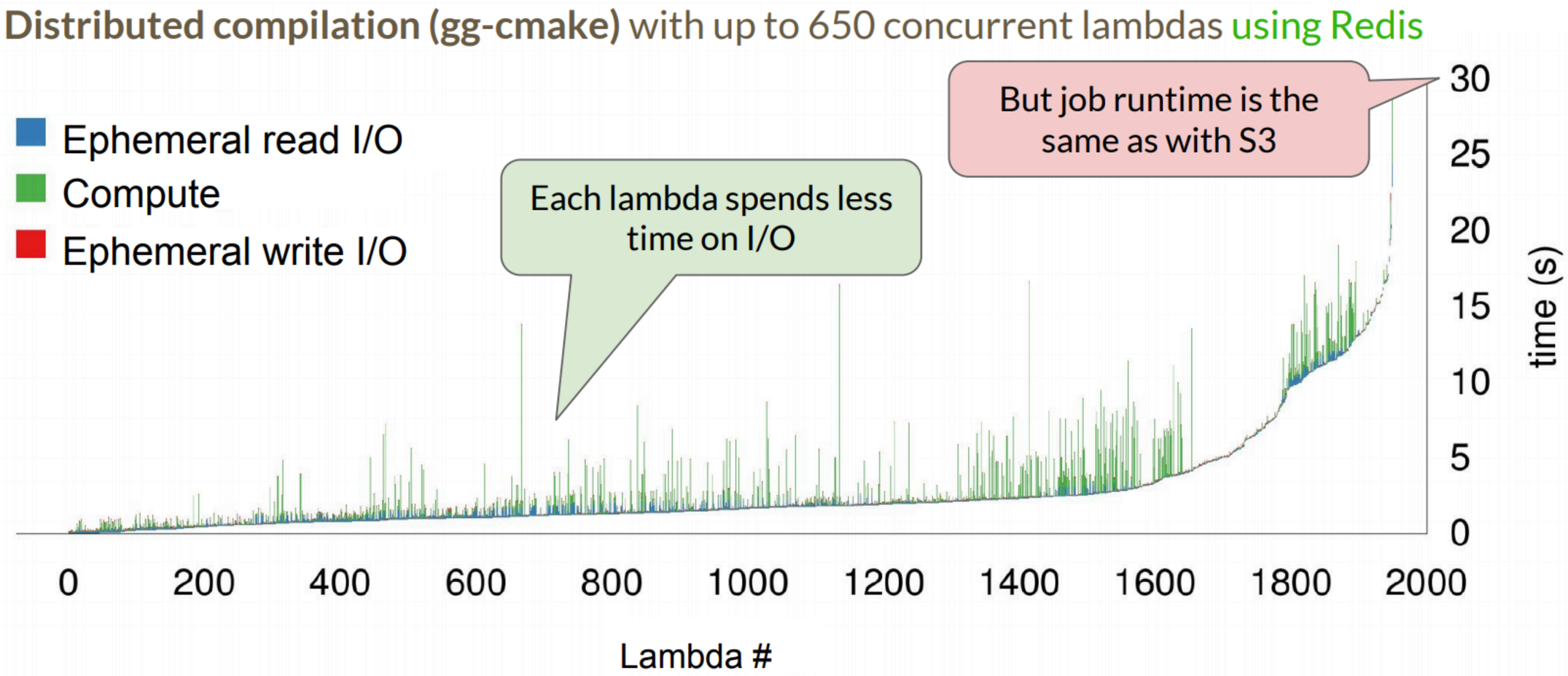


Figure based on Fig. 6 in "A thunk to remember: make -j1000 (and other jobs) on functions-as-a-service infrastructure (preprint)." Fouladi, S., et al.

The impact of application parallelism

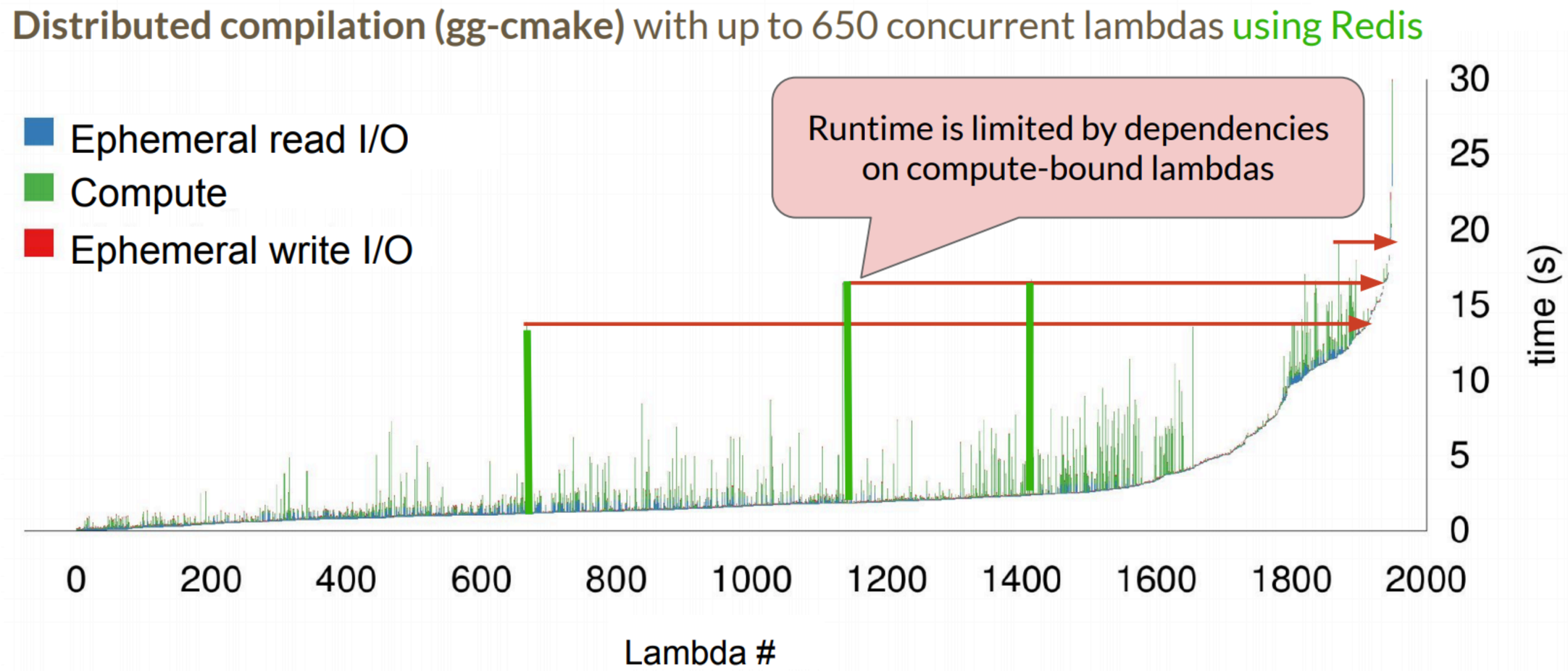


Figure based on Fig. 6 in "A thunk to remember: make -j1000 (and other jobs) on functions-as-a-service infrastructure (preprint)." Fouladi, S., et al.

The impact of application parallelism

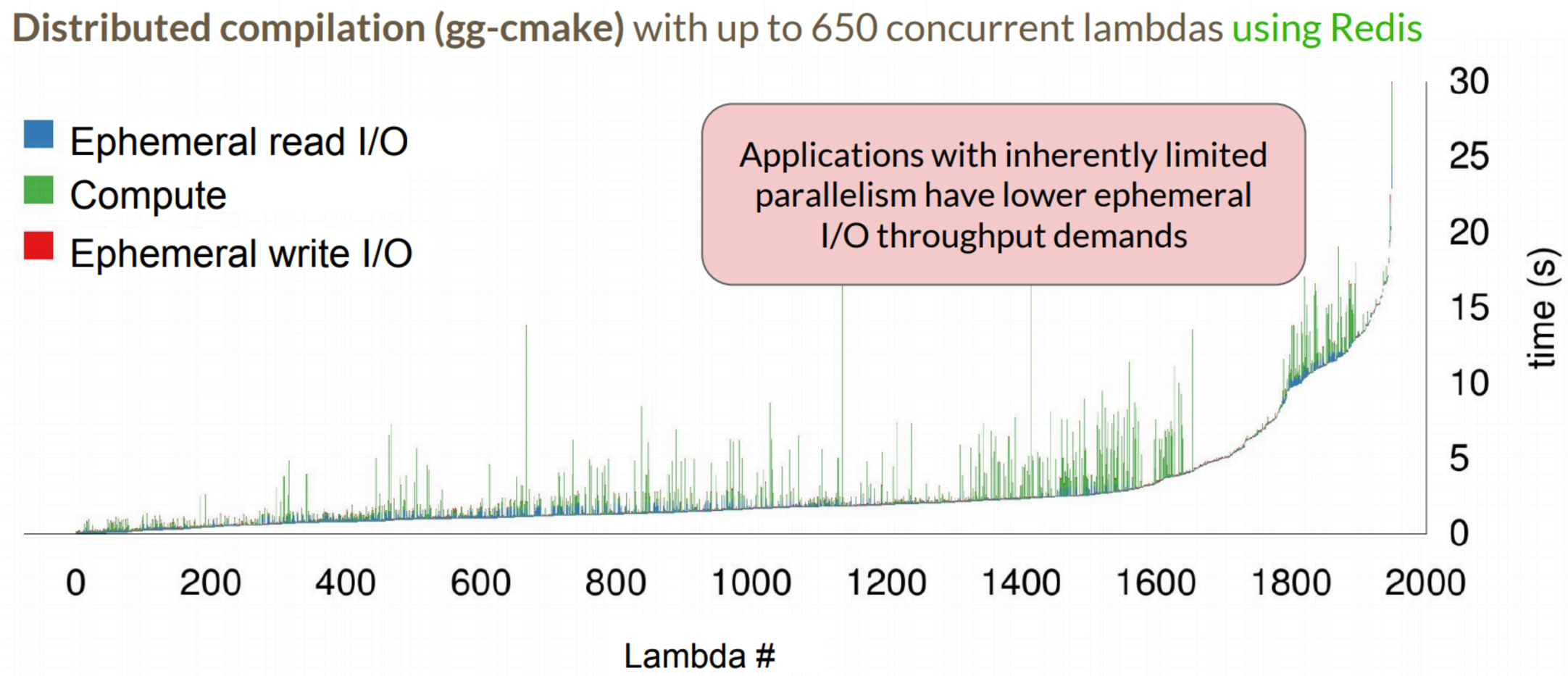
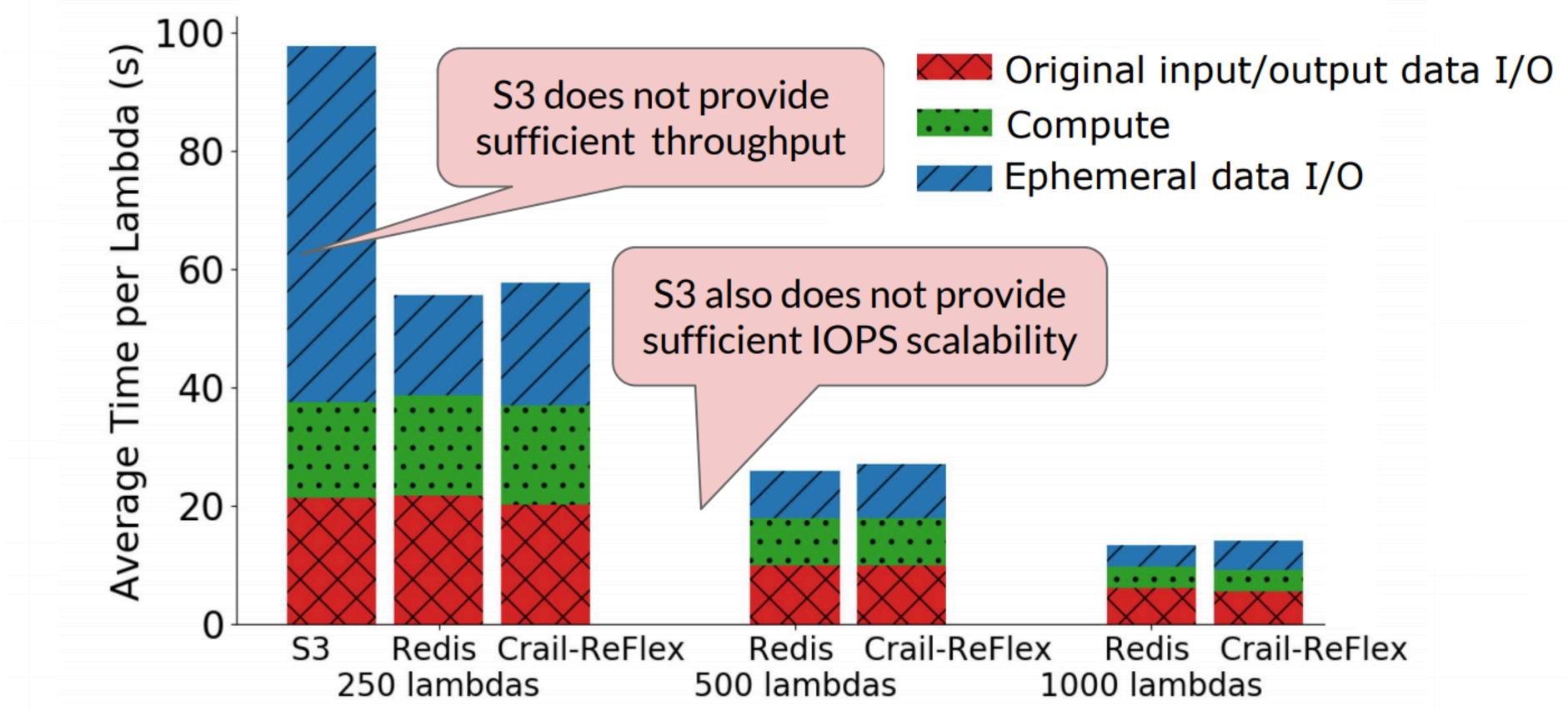


Figure based on Fig. 6 in "A thunk to remember: make -j1000 (and other jobs) on functions-as-a-service infrastructure (preprint)." Fouladi, S., et al.

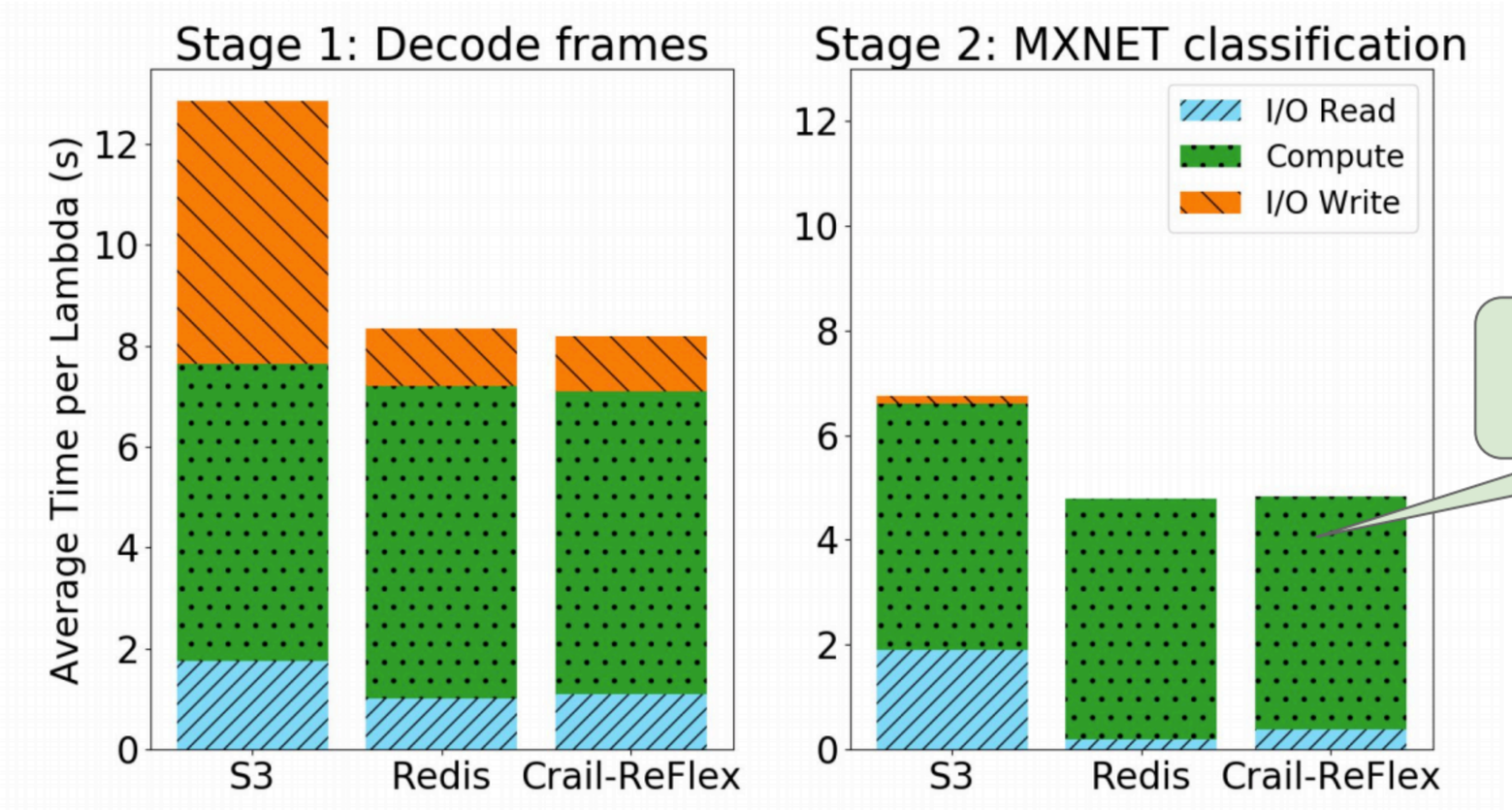
High I/O Intensity

MapReduce sort (100 GB) demands high throughput



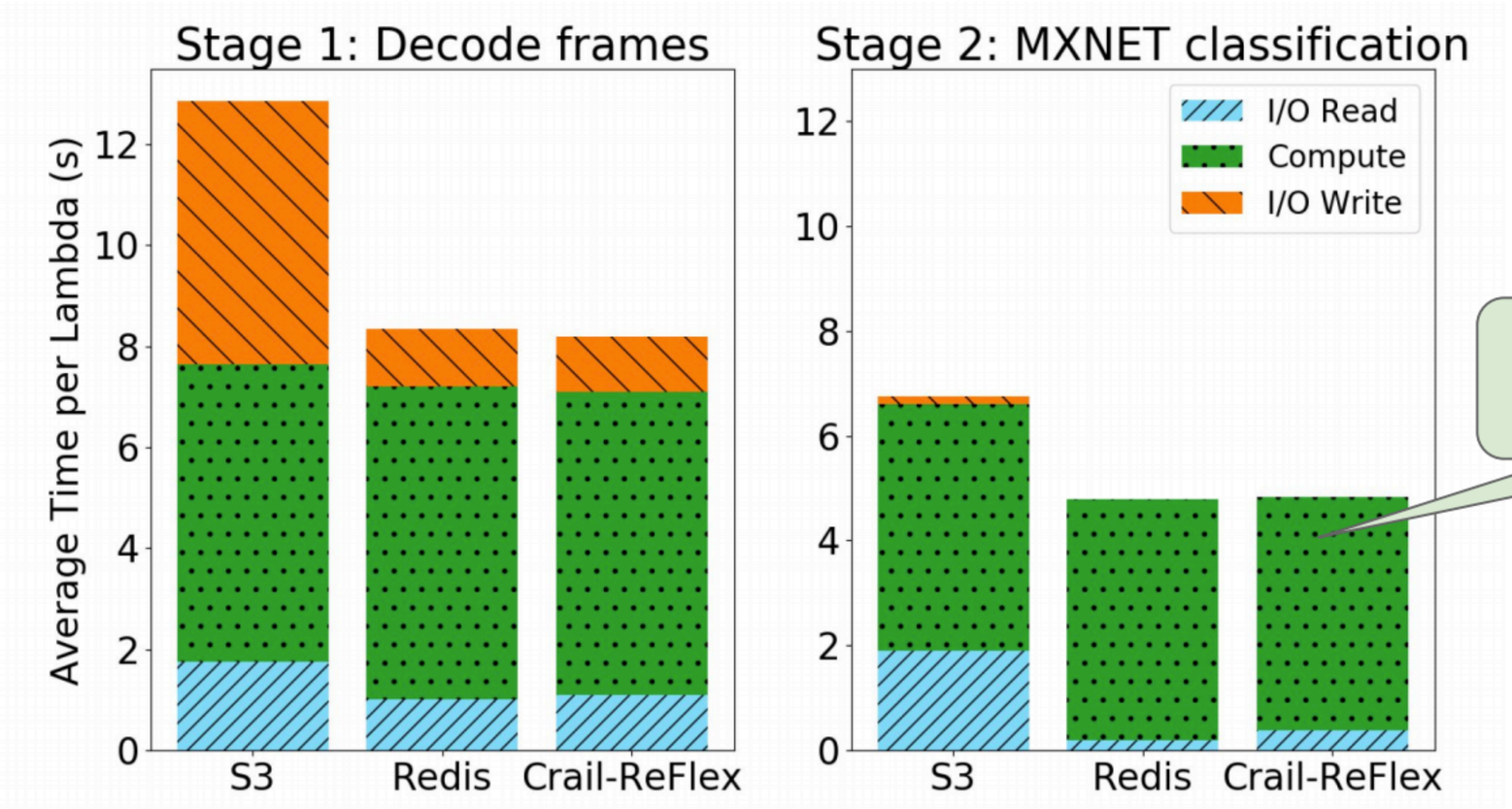
High I/O Intensity

Video analytics has both high I/O and compute intensity



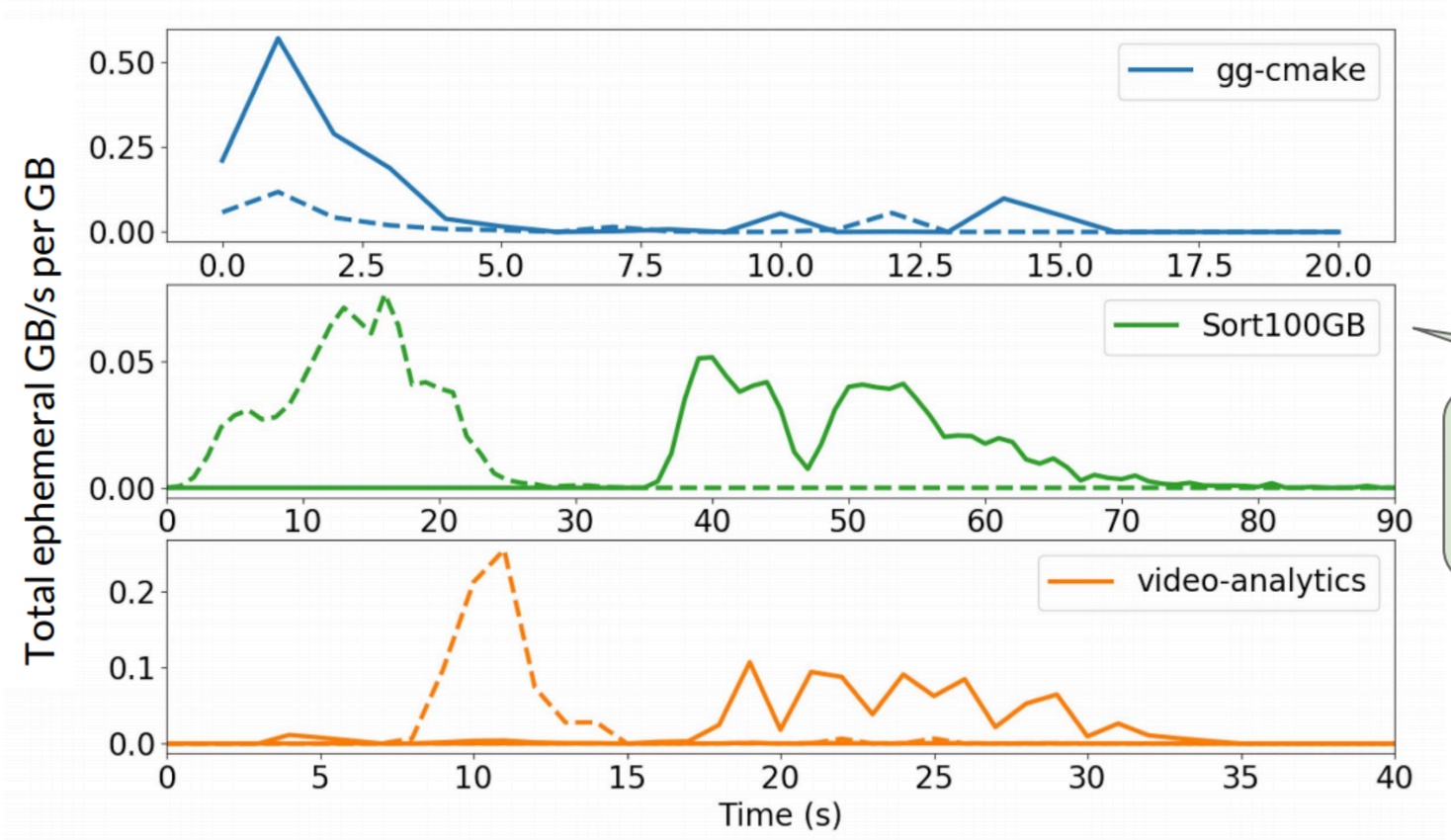
High I/O Intensity

Video analytics has both high I/O and compute intensity



3. Choice of Storage Media

- Compare throughput:capacity ratios of DRAM, Flash, HDD



DRAM: 20 GB/s / 64 GB = 0.3

Flash: 3.2 GB/s / 500 GB = 0.006

Disk: 0.7 GB/s / 6000 GB = 0.0001

Application throughput:capacity ratios are in DRAM - Flash regimes

Using Flash vs. DRAM, jobs achieve similar performance at lower cost per bit

Putting it all together…

- Ephemeral storage wishlist for serverless analytics:
  - ★ **High throughput and IOPS**
  - ★ **Low latency,** particularly important for small requests
  - ★ **Fine-grain, elastic scaling** to adapt to elastic application load
  - ★ **Automatic rightsizing** of resource allocations
  - ★ **Low cost**, pay-what-you-use

- Existing systems provide some but not all of these properties

OSDI 2018

Pocket: Elastic Ephemeral Storage for Serverless Analytics

Ana Klimovic*, Yawen Wang*, Patrick Stuedi+ , Animesh Trivedi+,
Jonas Pfefferle+, Christos Kozyrakis*

https://www.usenix.org/conference/osdi18/presentation/klimovic

Requirements of Ephemeral Storage

- High performance for a wide range of object sizes
- Cost efficiency, i.e., fine-grain, pay-what-you-use resource billing
- ~~Fault-tolerance~~

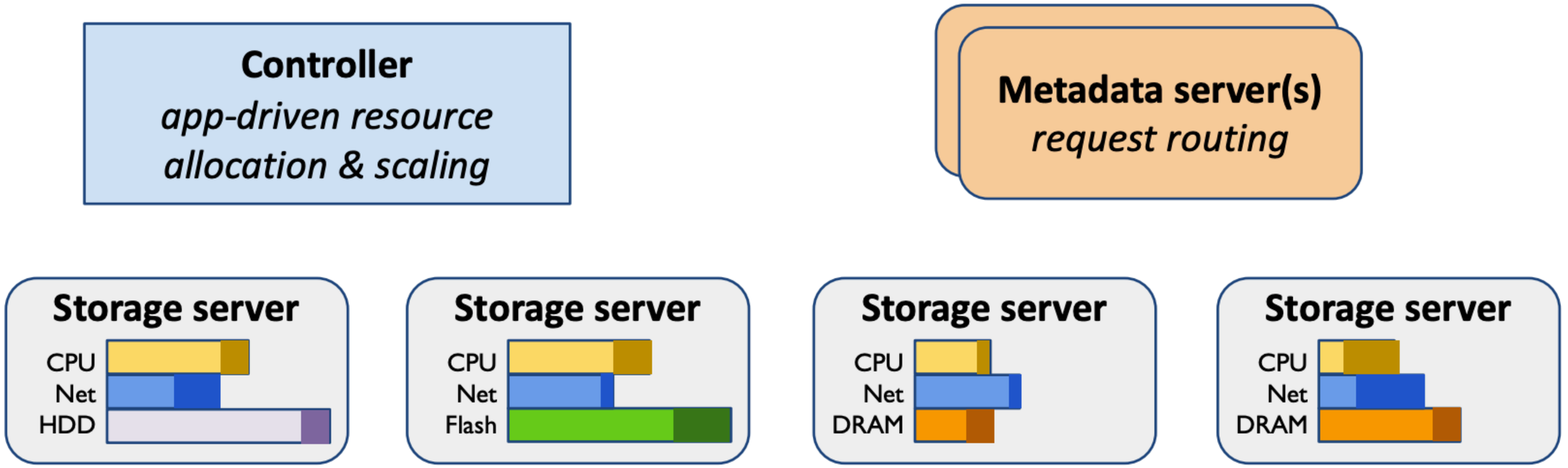
Pocket

- An elastic, distributed data store for ephemeral data sharing in serverless analytics

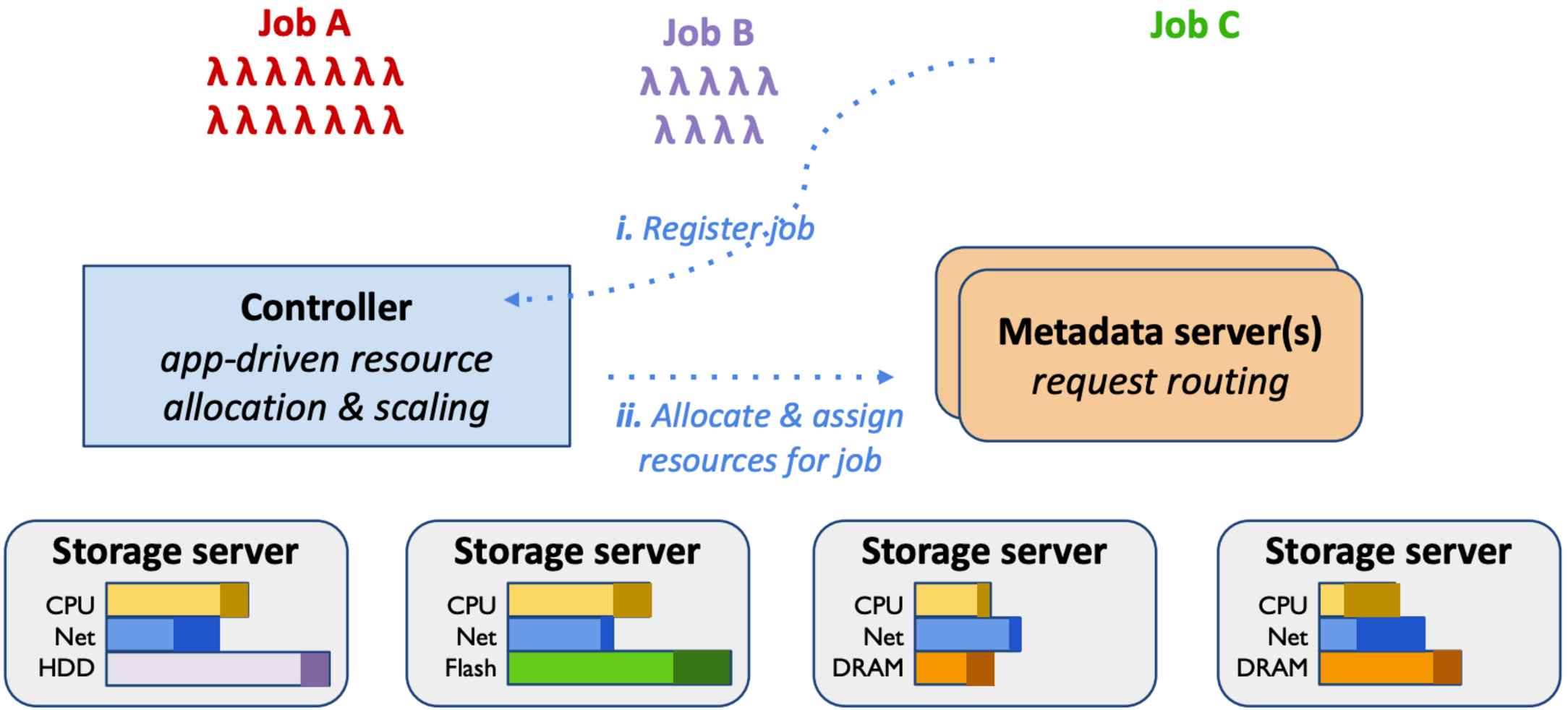
- Pocket achieves high performance and cost efficiency by:
  - Leveraging multiple storage technologies
  - Rightsizing resource allocations for applications
  - Autoscaling storage resources in the cluster based on usage

- Pocket achieves similar performance to Redis, an in-memory key value store, while saving ~60% in cost for various serverless analytics jobs

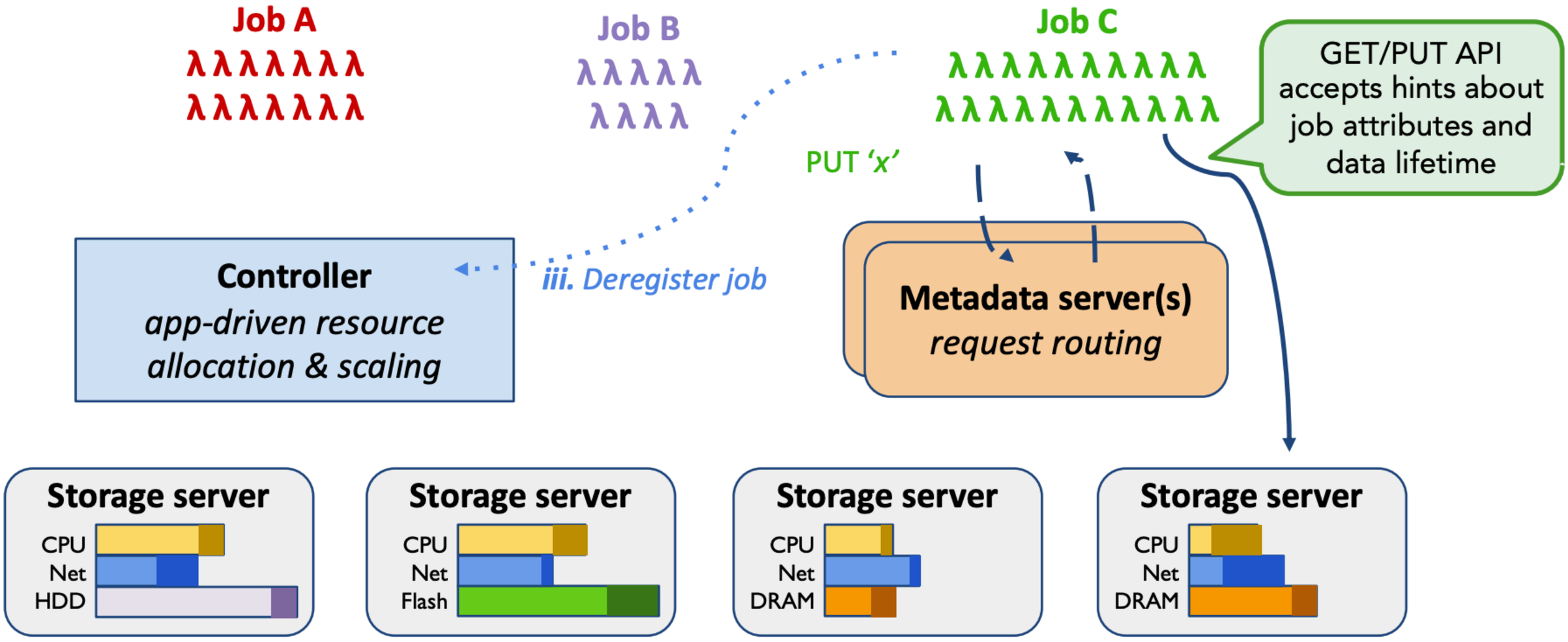
Pocket Design



Using Pocket

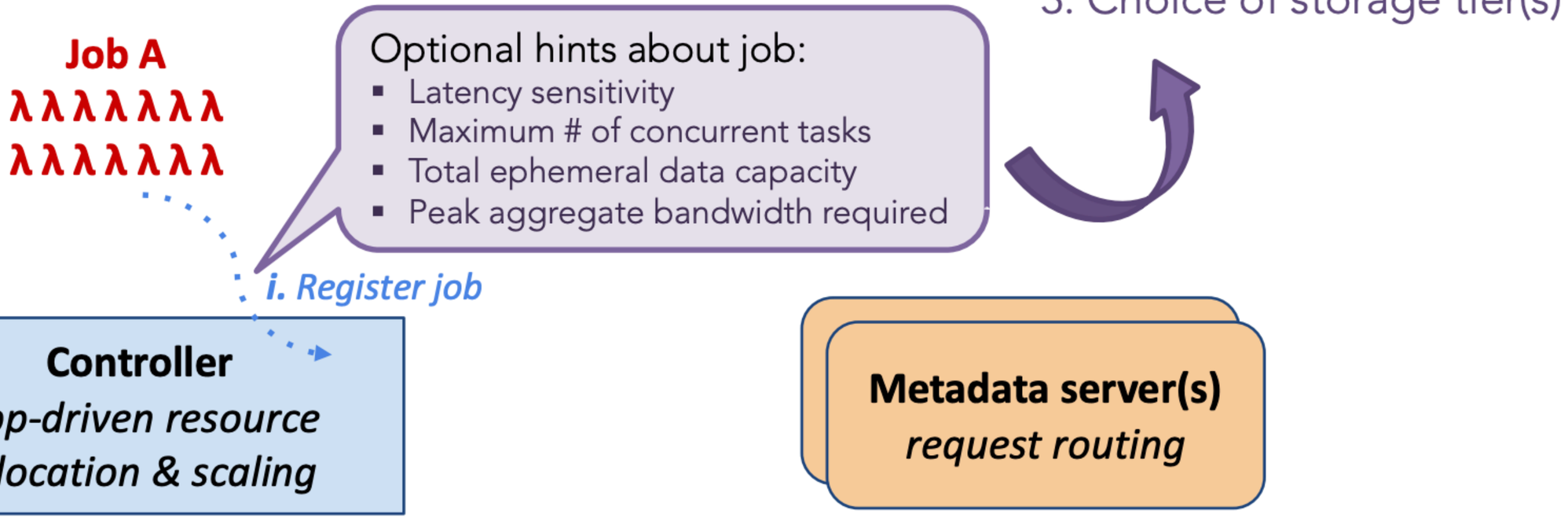


Using Pocket



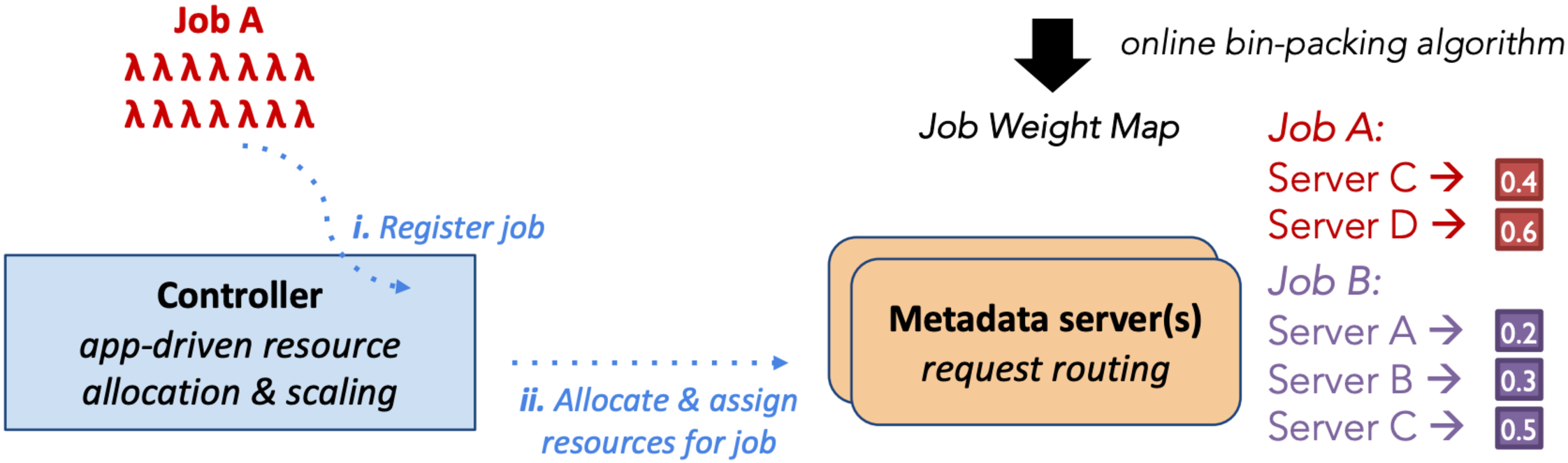
Assigning Resources to Jobs

1. Throughput allocation
2. Capacity allocation
3. Choice of storage tier(s)



Assigning Resources to Jobs

1. Throughput allocation
2. Capacity allocation
3. Choice of storage tier(s)



Autoscaling the Pocket Cluster

- **Goal**: scale cluster resources dynamically based on resource usage

- **Mechanisms**:
  - Monitor CPU, network bandwidth, and storage capacity utilization
  - Add/remove storage & metadata nodes to keep utilization within range
  - Steer data for incoming jobs to active nodes
  - Drain inactive nodes as jobs terminate

- **Avoid migrating data**

Implementation

- Pocket's metadata and storage server implementation is based on the ***Apache Crail*** distributed storage system

- Use ***ReFlex*** for the Flash storage tier

- Pocket runs the storage and metadata servers in containers, orchestrated using ***Kubernetes***

Pocket Evaluation

- We deploy Pocket on Amazon EC2

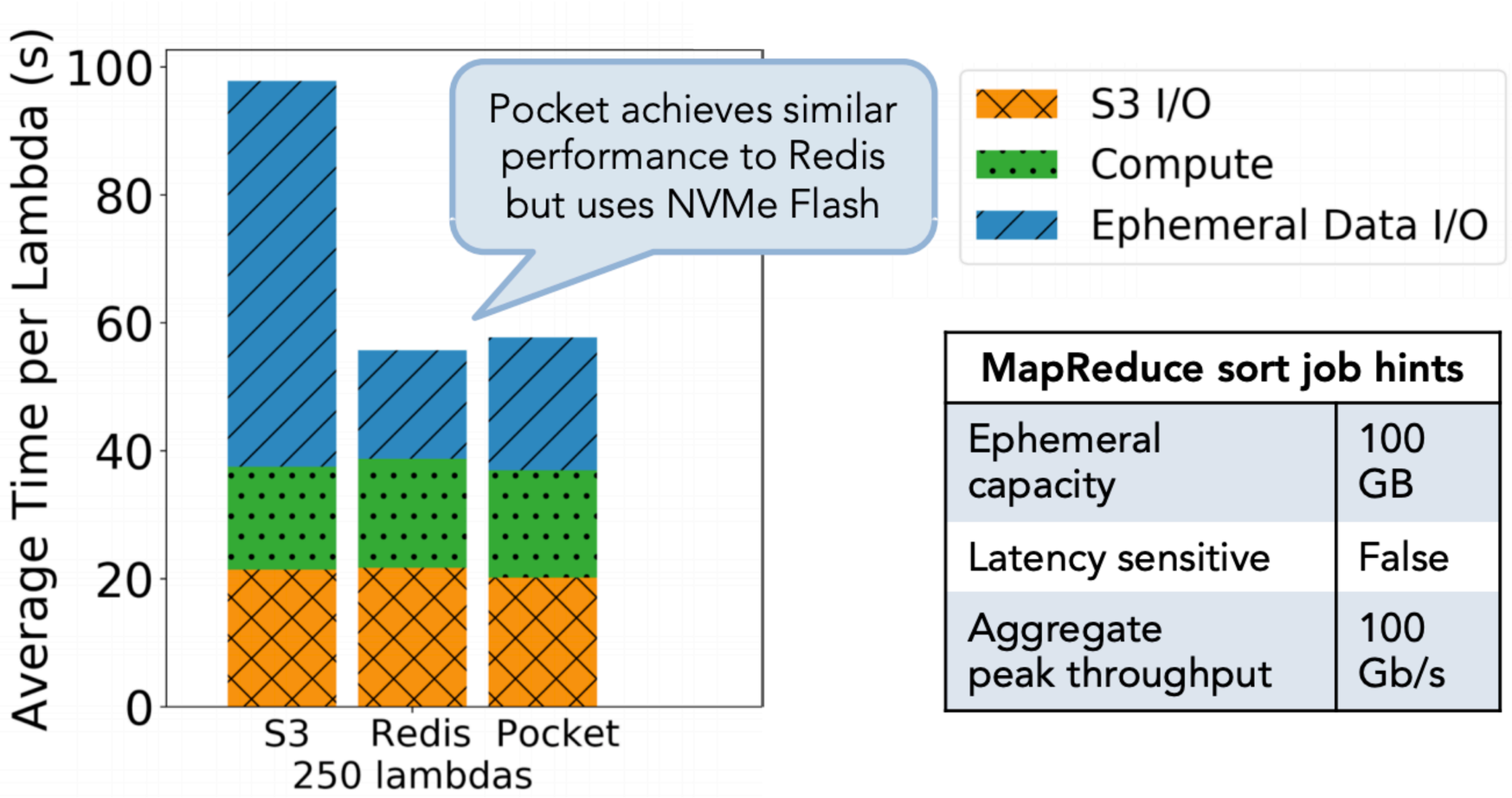
Controller	m5.xlarge
Metadata server	m5.xlarge
DRAM server	r4.2xlarge
NVMe Flash server	i3.2xlarge
SATA/SAS SSD server	i2.2xlarge
HDD server	h1.2xlarge

- We use AWS Lambda as our serverless platform

- Applications: MapReduce sort, video analytics, distributed compilation

Application Performance with Pocket

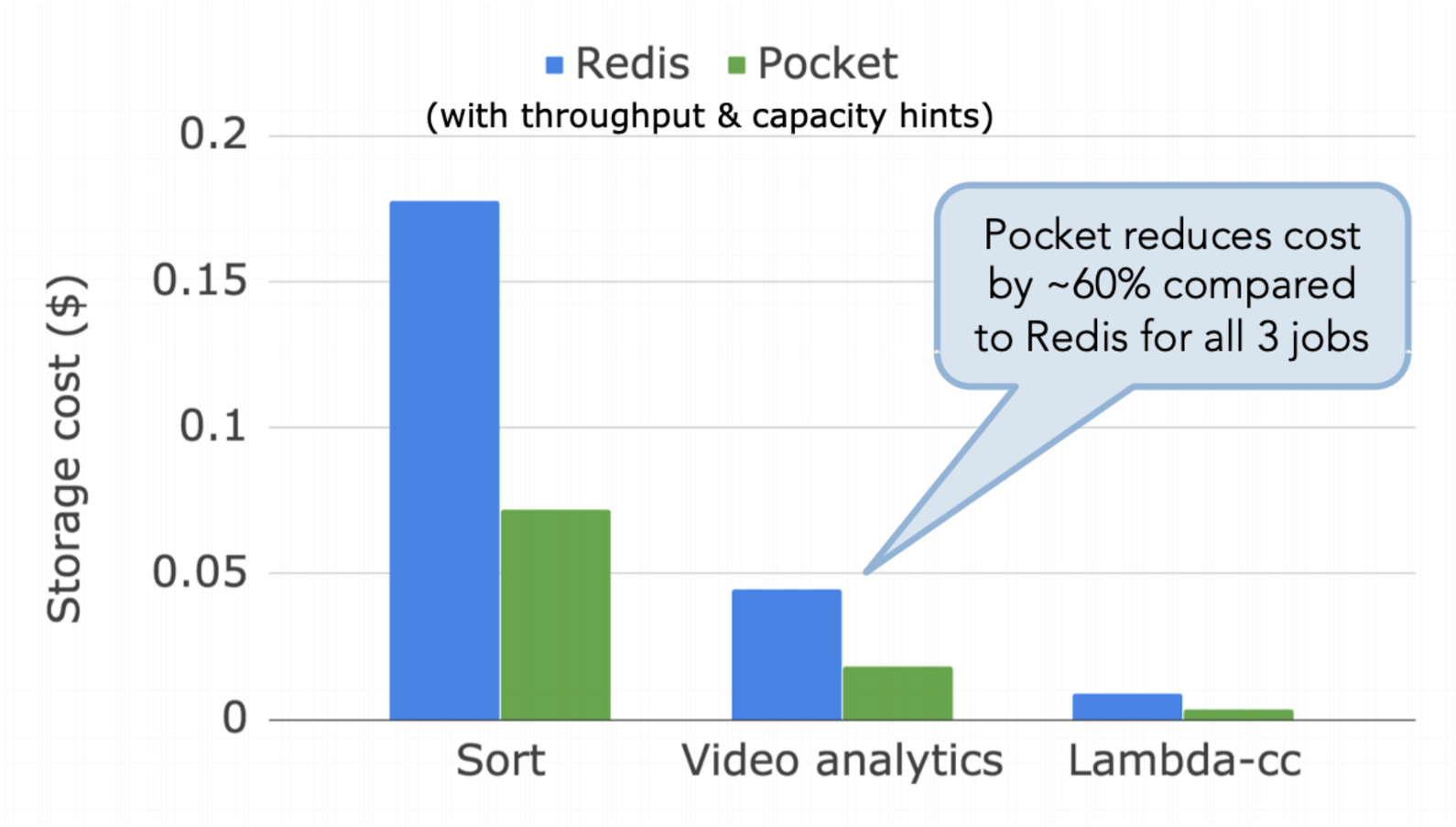
- Compare Pocket to S3 and Redis, which are commonly used today



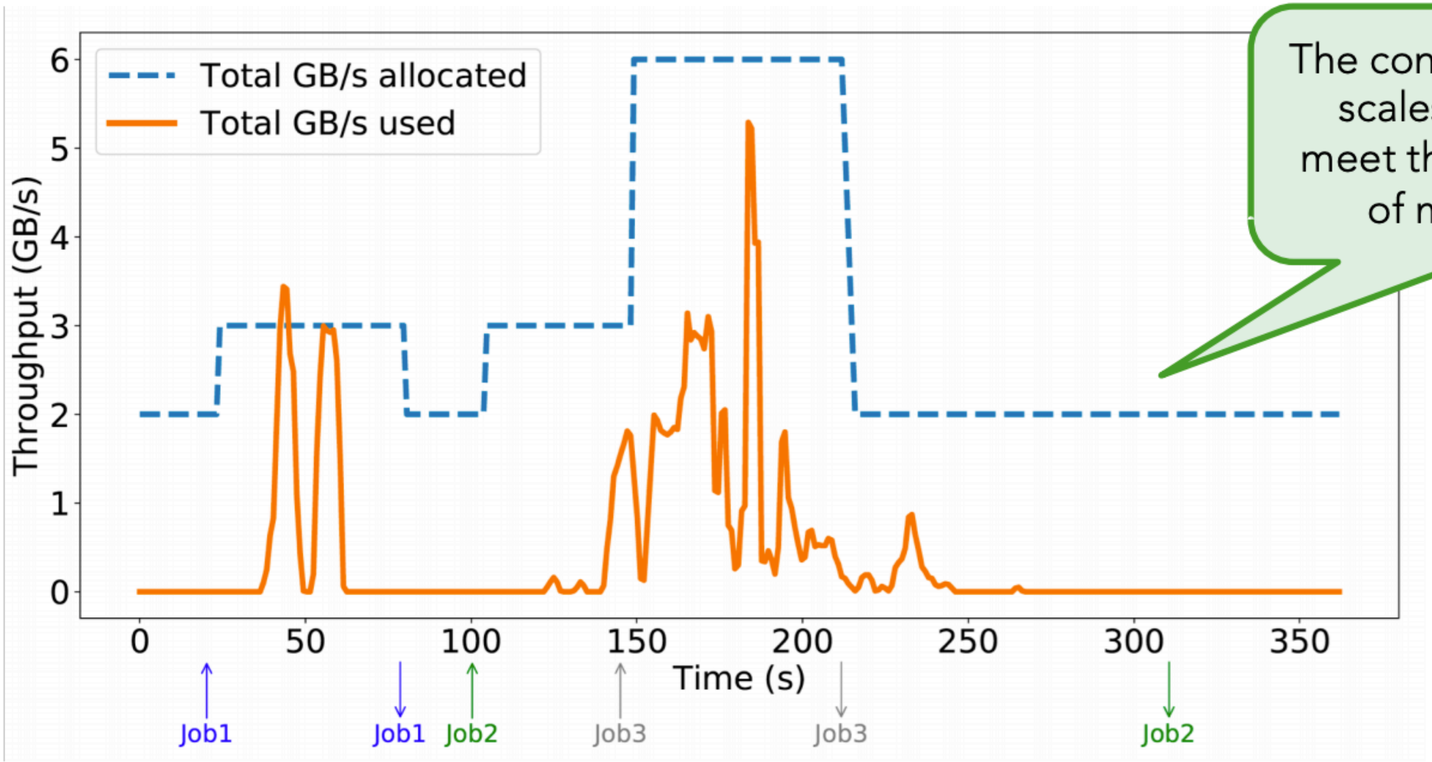
MapReduce sort job hints	
Ephemeral capacity	100 GB
Latency sensitive	False
Aggregate peak throughput	100 Gb/s

Application Performance with Pocket

- Pocket leverages job attribute hints for cost-effective resource allocation and amortizes VM costs across multiple jobs, offering a pay-what-you-use model



Autoscaling the Pocket Cluster



Job hints	Job1: Sort	Job2: Video analytics	Job3: Sort
Latency sensitive	False	False	False
Ephemeral data capacity	10 GB	6 GB	10 GB
Aggregate throughput	3 GB/s	2.5 GB/s	3 GB/s

Conclusion

- Pocket is a distributed ephemeral storage system that:
  - Leverages multiple storage technologies
  - Rightsizes resource allocations for applications
  - Autoscales storage cluster resources based on usage