UNIVERSITY OF CALIFORNIA, SAN DIEGO

System and Analysis for Low Latency Video Processing using Microservices

A thesis submitted in partial satisfaction of the
requirements for the degree
Master of Science

in

Computer Science

by

Karthikeyan Vasuki Balasubramaniam

Committee in charge:

Professor George M. Porter, Chair
Professor Aaron Schulman
Professor Geoffrey M. Voelker

2017

DEDICATION

I dedicate this thesis to the researchers who worked with me throughout the course of this project, without whom this success would not have been possible. Their guidance and constant support is the key factor for the success of the thesis.

EPIGRAPH

ADDICTED to SPEED?
Is your single processor too slow? The Concoction Machine Mark-1 parallel computer is just for you! No matter what your problem, solve it faster with the Mark-1's many processors working in parallel. In fact, we promise that with enough processors, any reasonable problem will run exponentially faster on our machine. Send us e-mail for more information.

–Inconsequent Machine Co
"We trade processors for speed."

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank my advisor Professor George M. Porter for his consistent and invaluable advice and support throughout my research and thesis work. His continuous support and feedback helped me in conducting this research and in the writing of this thesis.

Besides my advisor, I would like to thank Professor Keith Winstein and Riad S. Wahby from Stanford University, Rahul Bhalerao from Google who worked in this research for their immense knowledge and guidance.

I would like to extend my thanks to my thesis committee members, Professor Geoffrey M. Voelker and Professor Aaron Schulman for their inputs and support during this research.

The mu framework discussed in Chapter 4 is implemented by the co-authors Riad S. Wahby and Prof. Keith Winstein of the submitted publication. Material from Chapter 4 in part is currently being prepared for submission for the publication.

VITA

1991	Born in Erode, India
2009-2013	B.E., Computer Science and Engineering, PSG College of Technology, Coimbatore, India
2013-2015	Member Technical Staff, NetApp Systems India Private Ltd., Bangalore, India
2015-2017	M.S., Computer Science, University of California, San Diego, USA

PUBLICATIONS

Sathiya K Priya, Sudha G Sadasivam and **V B Karthikeyan**. “Article: A New Method for preserving privacy in Quantitative Association Rules using Genetic Algorithm” *International Journal of Computer Applications* 60(12):12-19, December 2012

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” *NSDI17*, March 2017

ABSTRACT OF THE THESIS

System and Analysis for Low Latency Video Processing using Microservices

by

Karthikeyan Vasuki Balasubramaniam

Master of Science in Computer Science

University of California, San Diego, 2017

Professor George M. Porter, Chair

The evolution of big data processing and analysis has led to data-parallel frameworks such as Hadoop, MapReduce, Spark, and Hive, which are capable of analyzing large streams of data such as server logs, web transactions, and user reviews. Videos are one of the biggest sources of data and dominate the Internet traffic. Video processing on a large scale is critical and challenging as videos possess spatial and temporal features, which are not taken into account by the existing data-parallel frameworks. There are a broad range of users who want to apply sophisticated video processing pipelines such as transcoding, feature extraction, classification, scene cut detection and digital compositing to video content

Parallel video processing poses several significant research challenges to the existing data processing frameworks. Current systems are capable of processing videos but with higher resource startup times, a small degree of parallelism, low average resource utilization, coarse-grained billing, and higher latency. This research proposes a low latency software run-time for processing a single video efficiently by orchestrating cloud-based microservices. The system leverages lightweight microservices provided by Amazon Web Services Lambda framework.

Chapter 1

Introduction

The rate of development of tools for data processing and analysis enables new applications over large data sets. Large data sets include medical records, web server logs and user reviews. Data-parallel frameworks such as MapReduce [19], Hadoop [17], Spark [3] and Storm [4] are used to process these large data sets efficiently.

The video, one of the biggest sources of data on the Internet is complicated to analyze. Today's digital cinematography, game industry, advanced robotics, television, and many other fields take advantage of data-intensive video analysis and processing. In 2013, video traffic was 60% of all Internet traffic, and according to Cisco's forecast, the percentage will grow to 75% in 2020. A majority of the video traffic includes video content delivery over the network for users streaming videos. In addition to streaming video content, users are demanding more complex video processing pipelines. Examples include video editing, video annotation, object recognition and video classification.

1.1 Background

Video refers to recording, manipulating, and displaying moving images, especially in a format that can be presented on the screen. With the advent of social media and television, users demand more computationally intensive complex processing to be done on videos.

Video processing is data intense, and several systems have been developed to speed up the ability to process videos by exploiting parallelism. A majority of systems support processing the videos in parallel to serve a massive number of users. Video processing systems operate two levels of parallelism to analyze videos: inter-video parallelism and intra-video parallelism. Current video streaming systems such as YouTube and Netflix deploy their systems in large clusters and rely on inter-video parallelism to improve the responsiveness of the system and support the broad range of users. As a result, these systems use coarse-grained parallelism - e.g., one thread per video. And hence they do not optimize the efficiency of processing a single video. In other words, intra-video parallelism has gained lesser attention.

Parallel video processing is critical, and existing data-parallel frameworks do not perform well on videos for several reasons. First, unlike other sources of data, video possess spatial and temporal correlations among the nearby frames which make it difficult for fine-grained parallelism. Second, users demand near real-time (interactive) processing with videos which is hard to achieve in spite of the available parallelism. Third, video processing jobs are computationally intensive and take a lot of CPU. Currently, it takes hours to process videos in high definition standards.

1.2 Motivation

This section sets up the motivation of this research based on the current situation of video processing systems.

Latency is critical for users. Users increasingly seem to apply complex processing to videos. Today, systems performing complex tasks such as video editing and compositing often take hours even for a short movie.

Optimizing intra-video processing is highly critical for users demanding near real-time video processing. Efficient processing of a video requires a system supporting intra-video parallelism which is challenging because splitting a video across threads loses the temporal correlations among the nearby frames.

The infrastructure that supports the execution of video processing jobs should be cost-effective. Most of the current video processing systems are developed using dedicated infrastructure such as large clusters, that incur significant cost and time in deployment and maintenance. With the advent of cloud computing, offering a pay-as-you-use model, video processing systems can be deployed using the resources provisioned on-demand such as virtual machines. Most of the cloud resources are billed on an hourly basis. Though cloud computing is cost-effective, the billing is not fine-grained.

Current video processing systems either run their applications on a dedicated cluster or deploy the application using virtual machines running in the cloud. Considering the resources, the efficient use of available computational resources is a key [40], and this creates the demand for systems that can optimize the utilization of these computational resources in response to the request. This problem becomes critical when the incoming rate of jobs is sporadic. For instance, there will be an unanticipated burst of client requests due to a seasonal event. There are moments when there is very low demand, and the resources are idle, thus underutilizing the resources. Hence, the current systems [40] do not take into account the seasonal variation of requests for processing and are not cost-effective.

Users require interactive processing of videos. Interactive processing of a video demands the application to exploit a large amount of parallelism for a very short period. In other words, this processing is bursty in nature and requires a large number of short-lived resources for a short period to process the video. Moreover, from an economic perspective, the variation in demands do not justify a massive investment in infrastructure, to provide computing power for peak situations.

1.3 Problem statement

The problem is to develop a software run-time that processes a single video efficiently by orchestrating cloud-based microservices running in parallel.

1.4 Contributions

This research makes two key contributions:

- A software run-time that orchestrates parallel computations using microservices running in the cloud. The run-time exploits the recent availability of services like

Amazon Web Services (AWS) Lambda. Unlike a traditional virtual machine running in the cloud, which takes minutes to start and billed on an hourly basis (AWS EC2), Lambda starts in milliseconds and bills usage at sub-second granularity.

- A system that interacts with the user in getting the input video and using the software run-time in an efficient way to stream the processed video to the user.

1.5 Layout of Thesis

The reminder of the thesis is organized as follows: Chapter 2: *Literature Review* focuses on reviewing the literature of parallel video processing using different data-parallel frameworks and the cloud. Chapter 3: *Parallel video processing* breaks down the problem into sub-problems and explains the background, problem, and goals in detail. Chapter 4: *Orchestration Framework for Microservices* talks about using AWS Lambda, its use cases and the orchestration framework for invoking AWS Lambda. Chapter 5: *Analysis of Grayscale pipeline* talks about the system design of using AWS Lambda for video processing, explaining the architecture, design, and implementation in detail. Chapter 6: *Conclusion* discusses the key contributions of this research, lessons learnt and the future work.

Chapter 2

Literature Review

Parallel video processing has a substantial literature. We review the literature in video processing on two fronts: data-parallel frameworks that support video processing applications and the computing resources that support the video processing systems.

2.1 Related work

There has been significant work done in the area of data-parallel frameworks that processes data using dedicated infrastructure. The work done in the last decade focuses on leveraging frameworks such as MapReduce [19], Apache Hadoop [17], Apache Spark [3], Apache Storm [4], StormCV [25], Dryad [37] and HTCondor [43] for efficient data analysis. They are suited for tasks with coarse-grained parallelism such as analyzing web server logs and medical records, where data is separable and distributed across threads. In these tasks, each thread processes a logically independent subset of data.

The computing resource that is used to run the video processing applications ranges from centralized scale-up compute servers, distributed scale-out architectures, cloud compute instances such as AWS Elastic Compute Cloud (EC2), Docker containers, and microservices.

This chapter discusses in detail on how each of the above frameworks along with the computing resource solves the problem of parallel video processing while optimizing on some of the research goals mentioned in Section 1.2.

2.2 Research Goals

Following is a comprehensive list of the goals of this research and our literature reviewed is compared against these goals.

- Improve the speed of processing a single video by optimizing on the intra-video parallelism.
- Orchestrate cloud-based microservices, each working on a fraction of the video to handle the bursty workload.
- Minimize the cost of the infrastructure supporting the execution of video processing system through fine-grained granularity in billing.
- The running time of processing a video should not be a function of the video length.
- A simple interface for the user to input the video and the operator, and get the processed video streamed.
- Ability to start and stop the computing resources faster than the conventional virtual machines.

2.3 Terminology

This section defines few important terms to describe the stages of a typical video processing job.

- ***The split step***: This is the **first** stage of any video processing job, where a single video file is decoded and separated into a set of frames (images), in PNG or JPEG format. In other words, for a single input, this stage produces M outputs, where M is the number of frames (images).
- ***The process step***: This is the **second** stage, where the operator, for example, grayscale is applied on each frame to produce a grayscaled frame. In other words, for a single input, this step creates a single output. Video processing jobs may include one or more process steps.

- ***The merge step***: This is the **last** stage, where a set of frames are combined and encoded into a single video in the compressed format. This stage is an M-1 stage, where M inputs are combined to produce a single output.

2.4 Cluster based computing

A cluster refers to a group of servers (nodes) and other resources that give a single system image with availability, fault-tolerance, and load-balancing for parallel processing. The distribution of tasks in a cluster for parallel processing has been adopted by many data-parallel frameworks to process the data efficiently.

2.4.1 Map-Reduce

The Map-Reduce paradigm [32] is a framework for processing large datasets using a dedicated cluster of computing nodes. MapReduce designates a node as the master, which coordinates the execution of jobs. Several computing nodes are identified as data nodes or worker nodes, which execute the tasks given by the master node. A MapReduce job consists of a Map stage, where the master node splits the data and distributes the splits across a set of worker nodes that process the information. The Reduce phase follows the Map phase in collecting the partial outputs from the worker nodes and combining them to produce the output. Apache Hadoop is a popular Map-Reduce implementation which consists of a Job Tracker, where the client submits a job and a Task Tracker which executes the map and reduce tasks.

Video processing has been solved using MapReduce paradigm, and several approaches have been proposed. Rafael et al. [40] proposed a distributed video processing architecture that implements a video processing application using Hadoop. Though Hadoop is efficient in processing large volumes of data, there are numerous challenges involved in the deployment. First, this requires dedicated infrastructure, which requires a substantial upfront investment and hence is not cost-effective. Second, Hadoop has been successful in batch-processing jobs but are not suitable for interactive processing. Weishan et al. [45] implemented the micro-batching model which approximates interactive processing but do not justify the investment required.

Distributed Video Transcoder [29] describes a video analysis framework that uses Hadoop to process videos efficiently. This framework assumes that video filter formats have a hierarchical structure like MPEG-2 and H.264. This hierarchical structure in the video helps the splitter to decode arbitrary input chunks. At a high level, the first stage of video processing has two jobs:

- A video sequence header MapReduce Job to look for metadata that is present in the init (first) chunk of the video file.
- A video decoder MapReduce job that uses the metadata obtained from (1) to decode a particular chunk and write into the Hadoop Distributed File System (HDFS).

In both of the above studies, the native Hadoop interfaces for splitting a data set do not fit videos because the splitter does not take into account the temporal correlations of the video. The split step as described in Section 2.3 has to be customized for dividing a compressed video file into images. The split step and the process step can be implemented as map stages, and the merge step can be carried out as the reduce stage.

In summary, though Map-Reduce aims at solving video processing through the high degree of parallelism, it requires upfront investment in setting up and maintaining the necessary infrastructure. Also, a MapReduce job has a higher startup delay in setting up the resources.

2.4.2 Apache Spark

Apache Spark [3] aims at processing volumes of data in near real-time through micro-batching. Spark can process data at a much faster rate than Hadoop and MapReduce because of the in-memory computing. Usually, Spark is deployed in a cluster environment to support batch-processing and near real-time streaming.

Spark employs in-memory computing to avoid disk reads and writes by keeping the data in memory. Spark follows the master-slave architecture. As shown in Figure 2.1, it has a single controller (driver) that communicates with many workers (executors). The driver converts the job into multiple tasks and assigns the tasks to the executors. Executors are processes running in the worker nodes executing the assigned task.

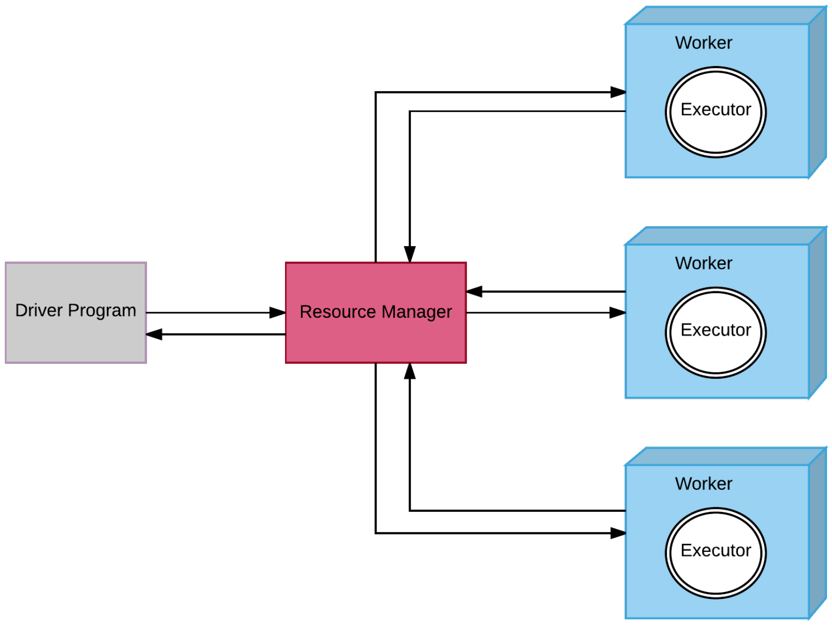

Figure 2.1: Spark Architecture

Spark creates the required executors for each job submitted, and all the executors share the memory available in the cluster. As described in Figure 2.1, the executor is a Java Virtual Machine (JVM) process with configurable amount of memory. Each executor has a pool of task execution slots, which is the unit of parallelism exploited by the tasks. The number of task slots is a function of the number of worker nodes, memory per node and number of executors per node sharing the memory.

So in short, the task slots of an executor use shared memory architecture to read and write data. Effectively, the throughput of the system depends on the number of task slots available, the memory associated with each slot and interactions with the driver.

Comparing to the research goals laid out in Section 2.2, Spark supports job with the high degree of parallelism and low startup latency due to the availability of light-weight task slots. Spark provides fault-tolerance and supports near-real-time processing due to micro-batching. Yet, there are few challenges associated with the deployment of video processing systems using Spark. First, deploying the system in "always on" clusters is not cost-effective and do not justify the variation in demand. Second, the number of task execution slots [35] available in a cluster is limited for a burst workload.

Third, scaling the cluster up and down is expensive and not instantaneous [35]. Fourth, the latency involved in the creation of task execution slots cannot efficiently support bursty workload like video processing.

2.4.3 Thunder

Thunder [26] is an ecosystem of tools that focuses on data-parallel, independent analysis of image data. It supports large-scale video processing by leveraging the capabilities of Spark. It provides uniform API irrespective of the job being run locally or distributed.

A job submitted to Thunder starts by getting the image or time series data from the user. Unlike Hadoop or Spark, the user has to execute the split step as discussed in Section 2.3 before submitting the job to Thunder. In essence, Thunder exploits parallelism with a rich set of operators for analyzing image data. The split and the merge step are not currently performed by Thunder.

2.4.4 StormCV

StormCV [25] is an open-source data-parallel framework that enables the use of Apache Storm [4] for image and video processing by leveraging Computer Vision (CV) operations. StormCV primarily supports OpenCV (Open Computer Vision) services and is flexible for the addition of new libraries. This platform enables the design and development of distributed video processing pipelines deployed on Apache Storm clusters.

The architecture of Storm [4] is similar to Spark in the context of job execution. A job submitted to the Storm creates a topology with a particular configuration indicating the amount of parallelism needed. A topology is executed by a set of the worker process. A worker process has its memory consisting of a set of executors. Each executor, in turn, has one or more tasks which are the threads indicating the unit of parallelism. The tasks created here are similar to the task execution slots created by Spark as described in Section 2.4.2.

Comparing against the research goals in Section 2.2, Storm [25] performs similarly to Spark because of the similarity in the architecture. StormCV supports additional video processing operators providing a richer toolkit for the users, yet it has few drawbacks. The latency involved in the creation of worker process, executors and tasks are higher, increasing the end-to-end latency for interactive video processing. Also, the inter-task communication is high when the amount of parallelism increases thus increasing the processing latency. Besides latency, Storm clusters [25] incur a significant cost for deployment and maintenance.

2.5 Cloud computing

Cloud computing is a way to dynamically provision and use scalable and virtual resources on-demand over the Internet. Cloud providers such as Amazon Web Services (AWS) [2], Microsoft Azure [22] and Google Cloud Platform (GCP) [16] offer a broad range of infrastructure as a service with a different granularity of billing.

The computing resources of cloud can be broadly categorized under three buckets: virtual machines, containers, and microservices. Most of the cloud providers including Amazon Web Services [2], Google Cloud Platform[16], and Microsoft Azure [22] provide all the three types of compute resources. Virtual machines and containers are discussed in the following sections, and microservices will be introduced in the next chapter.

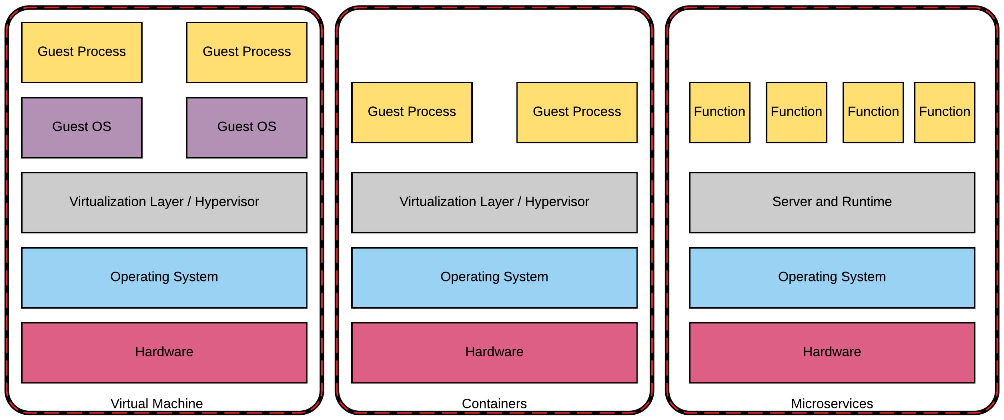

Figure 2.2: The Three Generations of virtualization

2.5.1 Virtual Machines

The virtual machine provides an abstraction of the hardware and Instruction Set Architecture (ISA), disk, CPU, network and peripherals. The virtual machine residing on a physical server runs a separate instance of the operating system. A physical server can host multiple virtual machines. A hypervisor creates and runs the different virtual machines in a physical server.

Cloud providers such as Amazon offer Elastic Compute Cloud (EC2) [1] tools which are virtual machines running in the cloud. Spinning up a new virtual machine is fast and takes approximately 45-60 seconds [38]. EC2 offers dynamic provisioning and auto-scaling of virtual machines based on the variation in demand. For example, as of 2017, m4.4xlarge is a type of virtual machine provided by EC2, which comes with 64GB of RAM costing $0.8 per hour.

Virtual machines solve most of the problems associated with the infrastructure needed by inter-video parallelism. They are cost-effective, can be dynamically provisioned, scaled up and down with granular billing. It poses some challenges in using virtual machines for parallel video processing. First, spinning up virtual machines each working on chunks of the video is expensive regarding cost and startup delay. Second, the billing is done on an hourly basis which does not justify the seasonal variation in demand.

Rafael et al. [40] emphasizes on Split & Merge architecture for high-performance video processing. This architecture is a generalization of the MapReduce paradigm, that rationalizes the use of resources by exploring on-demand computing. This system is deployed on Amazon Web Services (AWS) Public Cloud using Elastic Compute Cloud (EC2), Simple Storage Service (S3) and Relational Database Service (RDS). Few EC2 instances are designated as master instances that spawn and monitor multiple EC2 worker instances that execute the split and merge step. The intermediate results and final output are written to S3 and RDS. This approach is not cost-effective and has high startup times due to the reasons discussed above.

2.5.2 Containers

Traditionally, operating system allows one one user space instance to run applications. Operating system level virtualization makes it possible for the kernel to run multiple isolated virtual instances. These isolated virtual instances are called containers. Various OS-level virutalization systems have been developed over years.

A container consists of an entire runtime environment: application, dependencies, libraries and configuration files, bundles into one package. As shown in Figure 2.2, by containerizing the application, differences in the operating system distributions are abstracted away. Containers improve on many of the aspects of virtualization by improved performance isolation, resource control, and efficiency.

One major problem with virtual machines was the adaptability of the system to variance in load. Usually, with websites that support millions of users like Amazon, Facebook, and Netflix, there will be variation in traffic (requests per hour) that may affect the system performance and quality of service. With containers, it is easy to run components of the system with isolation so that the critical elements of the system are not affected.

Wrapping the web services with containers, the application is least affected by any of the changes that occur in the host environment that support all the containers. Containers are more efficient than virtual machines for large scale systems. Containers can be cloned and started relatively fast compared to virtual machines. Minimal **start-up time** (approximately < 1 second) compared to virtual machines. Reduces the start-up time of application which is directly proportional to the quality of service impacting the user.

Chapter 3

Parallel video processing

This chapter explains the structure of a video format and emphasizes the challenges in developing a parallel video processing system thus setting the expected peak performance of an ideal video processing system.

3.1 Structure of video

3.1.1 Video formats

A video consists of video frames displayed at a specified frame rate (frames per second). The frame format determines the size of images regarding pixels. Most movies use 24 frames/sec.

3.1.2 Frame Type

There are three types of frames in a video: I-frame, B-frame, and P-frame [39]. ‘I’-frame refers to the Intra-coded frame, which is a reference frame representing a full image and is independent of the other frames. I-frames are encoded without motion compensation which makes it a reference for future predicted frames. ‘B’-frame refers to the bi-directional predictive frame. This frame contains the difference information from the adjacent I- or P- frame. ‘P’-Frame refers to the Predictive frame, which is encoded using motion compensated prediction from either I- or P- frame.

A group of Pictures (GOP) [39] relates to the collection of frames between two successive 'I'-frames including the 'I'-frames. The presence of GOPs in the video enables random access to the video, such as fast forward and reverse playback.

3.1.3 Keyframe

Videos are available in the compressed or raw format. A compressed video applies video compression techniques to exploit the correlation among the adjacent frames. Sections of the image are not repeated in the encoded bit stream if the portion of the images does not change between the adjacent frames. But this makes random seek to the video stream impossible because decoding in the middle of a stream does not work when the images depend on one another.

Video encoders add **"Key Frame"** or **"Stream Access Point"** (SAP) in the beginning, and periodically in the middle to enable random seek in the video. The property of key frame is that it resets the stream and becomes a barrier. The video decoder can start decoding from any key frame irrespective of the availability of the previous frames of the bit stream. When the video is encoded, usually it carries a key frame for every 10 seconds so that the minimum seek interval while streaming the video will be 10 seconds, which is reasonable for videos of all length.

The chunks of a compressed video that are separated by a key frame are independent. This property is useful in encoding the video in parallel. Each thread can operate on its piece of chunk without communicating the frames of a video. When it comes to processing other than encoding and decoding, such as applying the black-white filter to the video, it is easy to emit the frames of a video and let each thread apply the filter to a small set of frames.

3.2 What is parallel video processing?

Parallel video processing refers to the intra-video parallelism or frame level parallelism, where multiple threads process independent chunks of a video. Each thread works on a chunk of the video. Frame level parallelism is widely adopted to enable near-real-time processing because it can scale with the length of a video. Most systems do frame level parallelism where each worker processes thousands of frames. The goal of this research is to enable fine-grained parallelism where each thread operates on a small set of frames.

3.3 Solution Overview

Many operators can be applied to video content such as classification, object recognition, grayscaling, compositing and color grading. Grayscaling is one of the simplest and fundamental problems in video processing. The problem of grayscaling a video can be broken down into three sub-problems. Problem-1 (Stage-1): Split the compressed video from mp4 format into a set or frames. Problem-2 (Stage-2): Apply the grayscale operator to each of the individual frames of the video to produce a grayscale frame. Problem-3 (Stage-3): Merge the frames back into a video by using a suitable video encoder. An analogy of this problem will be with a typical Map Reduce job. Stage-1 resembles the splitter task (one-to-many function) where the input data is split into independent chunks based on a particular format. Stage-2 resembles the mapper task, a one-to-one function where a function is applied to an input to produce an output. Stage-3 resembles the combiner and the reducer task (sometimes a many-to-one function), where multiple partial outputs are merged into a single output.

3.4 Infrastructure

As discussed in Chapter 2, the required infrastructure to support the execution of parallel video processing jobs falls into two broad categories: cluster-based computing and cloud computing. Data-processing frameworks such as Hadoop and Spark today rely on the high-performance computational platform offered by cloud providers such as Amazon EC2 and Azure virtual machines. Cloud providers offer heavyweight virtual machines for running the application servers.

3.5 Components of an ideal system

A parallel video processing system should have the following components. The first component is an orchestration framework that manages the spawning of threads and cloud resources for processing the video. The framework should handle a high degree of parallelism with minimal overhead to minimize the overall running time of the system. In addition to this, this framework should handle fault tolerance, replication, data management and network interactions. Finally, a high-level programmatic interface that the end user can use to process the video.

A set of efficient parallel processing algorithms that operate on videos and frames. A typical video processing job will have three sub-tasks as discussed in Section 3.3. An input splitter that converts a video into a set of images by using a decoder and a video processing tool such as FFmpeg. An operator that processes each image and produces an output image. A combiner or merger that encodes the set of images back into a video and makes it available for consumption by the streaming services and the end user.

3.6 Expected peak performance

This section talks about the expected peak performance of an ideal video processing system that has the components discussed in Section 3.5. The performance metrics are not limited to the following sections. However, this research focuses on the following criteria to evaluate the system. Other performance measures include time spent in network communication while accessing the cloud resource, and performance achieved when using the cloud resource optimally as instructed by the cloud providers, etc. This research is evaluated based on the standard features that will be offered by any cloud provider or dedicated infrastructure to understand the base system performance better.

3.6.1 Degree of parallelism

The level of parallelism is mainly limited by some resources that can execute in parallel. With containers such as Docker, Saarinen et al. [41] says that a maximum of

500 containers can be created on a host. Cloud-based microservices are available when the application has a bursty workload, where the system needs to spawn hundreds of resources for a shorter period and tear-down quickly. With AWS Lambda, microservices framework from AWS, the number of concurrent executions without being queued (not compromising the running time) is the limit that controls the degree of parallelism. Any user having an account with AWS gets a default limit of 100 concurrent invocations and can be extended to 1,200 by request. Other cloud providers such as Microsoft Azure and Google Cloud Platform have similar defaults and extended concurrent limits.

A high degree of parallelism, together with an appropriate chunk size can achieve high performance. When the level of parallelism is set to the concurrent limit exposed by the underlying infrastructure, the chunk length can be readily determined by the following equation:

$$chunk\_length = \frac{video\_length}{concurrent\_limits}$$

3.6.2 Running time complexity

The running time of processing a video is measured as the end-to-end latency involved in executing all the stages of the processing, and the interactions with the orchestration framework and the cloud service. When the concurrent limit of the available resource is increased, the chunk length is reduced, doing less work per thread, which reduces the time to process the video. So the running time of the stage is a function of the chunk length processed by the resource.

Ideally, when there is no overhead due to communication and concurrency, the running time of a stage should be the time taken to process a chunk. But in reality, interactions with the orchestration framework, cloud resources startup, storing intermediate data, networking and throttling increase the running time of a stage by a variable amount. Still, the asymptotic complexity that is desired to be a function of the chunk length.

Let N represent the number of stages involved in processing a video, t_i represent the time spent in executing a stage, C represent the chunk length, T represent the total time taken to complete a job and α be the time spent in the interactions in the system.

$$t_i = O(C)$$

$$T = \sum_{n=1}^{N} t_n + \alpha$$

3.6.3 Space complexity

The space complexity includes the space occupied to store the input video, output video and any intermediate data produced. A video processing problem in most cases is reduced to the image processing problem by splitting the video into images using existing tools such as FFmpeg. During the execution, there are sets of images produced which are written to the intermediate storage. The number of images can be computed as the product of the frames-per-second and the length of the video in seconds. For example, Sintel [24], an animation movie runs for 888 seconds, encoded at 24 frames-per-second, has a total of 21,312 images. Let f be the frames-per-second of a video and L be the length of the video in seconds. The asymptotic space complexity is a function of their product.

$$s = O(f * L)$$

3.6.4 Cost Estimation

One significant advantage of leveraging the infrastructure offered by cloud providers is the pay-per-use model through different granularity of billing. As discussed in Section 3.3, it is possible to estimate the cost incurred for running a video processing job by getting the metering for all the services consumed. Cloud -based microservices such as AWS Lambda offer granular billing for consuming microservices. For example, if a job requires 1000 invocations of a function in the cloud, the cost of computing will be the product of the number of invocations and cost per invocation.

Chapter 4

Orchestration framework for Microservices

This chapter emphasizes the importance of using microservices [21] for video processing and introduces the software run-time that processes a single video efficiently using microservices. The software run-time uses AWS Lambda [6] framework for achieving the desired performance. The features of AWS Lambda discussed in this chapter are taken from the documentation [7] provided by Amazon as of 2017.

4.1 What are microservices?

For years, applications are designed as a unified component and deployed in a monolithic architecture style. The monolithic design has several consequences. First, as new requirements for software come up, changes made to the software has to be deployed everywhere causing intermittent availability issues. Second, due to lack of modularity in deployment, productivity slows down, and the deployment becomes complex. Third, scaling individual software components becomes a bottleneck, because all the elements co-exist in the same logical resource.

Microservices [21] are an architecture as well as a mechanism. It is an architectural pattern which breaks a complex application into a set of small and independent processes communicating with each other using language and platform agnostic APIs. In other words, it changes the strategy from having a single legacy unified application to a functional paradigm that operates as a set of functions over the web.

Microservices architecture breaks the unified software into a set of software components that can deploy independently into different services. Services can communicate by invoking each other through standard protocols such as HTTP. Microservices solve all the three problems discussed above with a negligible overhead in orchestration. The major challenge in migrating towards a microservices-based architecture is the need for a stable orchestration framework, that is a system, capable of managing microservices with reliability, fault-tolerance, and responsiveness.

With microservices, the computing resources are launched and destroyed on-demand, thus achieving nearly 100% utilization of the resources. Microservices reduces the task of server management by spinning up computing power on-demand.

4.2 Microservices for video processing

The primary use case for microservices is to improve the large-scale software deployment and server management. Though microservices are not intended for solving parallel computing problems, the architecture, usage, and cost of microservices make it efficient for solving an embarrassingly parallel problem such as video processing. A typical video processing workflow as discussed in Section 3.3 includes a series of complex interleaving operations such as splitting, transcoding, processing, merging and streaming. These processes follow the microservices architectural style which enables each of the operation to evolve independently and scale out in multiple dimensions. Since these operations are data-parallel and isolated from each other, it is easy to launch a large number of microservices for each operation.

4.3 AWS Lambda - Run code in the cloud

AWS Lambda [6] is an event-driven, serverless computing platform provided by Amazon as a part of the Amazon Web Services. It runs applications without having the need to provision or manages servers. The user uploads the code to be executed and configures the trigger, and the AWS Lambda infrastructure takes care of execution and scalability. Though AWS Lambda is not intended for general-purpose parallel computations, the features of lambda enable it to be a plausible solution for solving an embarrassingly parallel problem like video processing.

AWS Lambda [7] is a cloud platform with almost zero administration required. The main benefits of AWS Lambda framework are sub-second (100ms) billing [6], serverless computing, and continuous scaling. Some of the common use cases of Lambda are real-time file processing, real-time stream processing, ETL (Extract, Transform and Load) style applications and web application back-end.

Lambda lets the user create a simple function by uploading a zip of the application binaries. For example, a simple lambda function would be to generate a thumbnail for the photo uploaded by a user in an AWS S3 [9] bucket. The lambda function has the functionality to create a thumbnail and produce the output to another AWS S3 bucket for display. Every object added to the S3 bucket will trigger the lambda function.

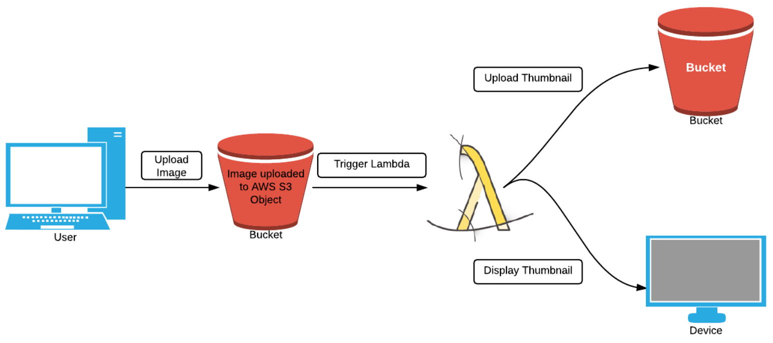

Figure 4.1: Thumbnail creation pipeline using AWS Lambda

4.3.1 Concepts

Following are some of the fundamental concepts associated with the AWS Lambda framework.

- ***Lambda function***: The most important component of the Lambda framework is the **lambda function**. It is a piece of code written in a programming language supported by the AWS Lambda framework such as Node.js, Python, Java, etc.

The function defines an entry point called the *lambda_handler* irrespective of the programming language, and it accepts two parameters. The first parameter is the **event**, which captures a set of key-value pairs. The second parameter is the **context**, which specifies the execution environment such as programming language, memory configuration, timeout in seconds, permission and role.

- ***ARN***: Amazon Resource Name refers to the code and the metadata that is stored durably in AWS. The Lambda infrastructure creates an instance of ARN on an invocation of the lambda function.
- ***Invocation*** : Invocation refers to the lambda function in action. Lambda invocation gets instantiated when the request for invocation comes from AWS Management Console or AWS Command Line Interface.

4.3.2 Programming Paradigm

This section talks about the lambda programming model as indicated by AWS documentation. Execution starts from the *lambda_handler*, that has access to the **event** object containing the JSON data structure. Additional copies of the function are created to scale and evolve with changes.

Each lambda invocation comes with 512MB (subjected to change) of temporary disk space. This storage is not durable, hence any persistent data should be stored in AWS S3, DynamoDB or Redis or any other reliable AWS storage service.

The code written in the lambda function can make use of the functionality supported by the programming language chosen, or the Linux environment that supports running the lambda function, or the official SDK provided by AWS to interact with other AWS services.

4.3.3 Run-time Environment

Having seen the programming paradigm of AWS Lambda, this section talks in detail about the characteristics of the lambda run-time environment.

- The *context* object contains the timeout that stops the execution of lambda function when time elapses. Usually, most of the lambda functions use 60 seconds as a timeout. However for video processing, encode, decode, and other image operations take significant time to execute.

- The *memory* and the *timeout* requirements are configured during the function creation. All invocations of the same function have the same memory and timeout configuration. Memory varies anywhere from 128MB [1] till 1.5GB [2]. The maximum timeout allowed is 5 minutes (300 seconds). A lambda invocation can create up to 256 threads and up to 1,024 file descriptors.

- The *invocation role* gives the lambda function required permission to execute. The execution role provides the lambda function the access to other AWS services.

- *AWS CloudWatch* stores the fine-grained statistics for each lambda function such as the latency, context information, errors, etc.

- The CPU power, network and disk bandwidth are chosen based on the memory configured to the lambda function. The user does not have control over the configuration except memory and timeout. *Billing* depends on the amount of memory set (not the actual memory used) and the time in 100 milliseconds granularity the function took to execute.

- By default, each user gets a *concurrent limit* of 100 lambda invocations per AWS region across different functions. The limits can be extended to 1,200 per AWS region upon request. Any requests that are invoked after the limits are exhausted are throttled.

4.3.4 Cost

As of 2017, a single AWS Lambda invocation costs $0.00001667 for every GigaByte-second [8]. Comparing to the closest AWS EC2 instance (c3.large) [5] which runs with same memory configuration, but higher memory and disk space, the cost of

[1]**MB**: Mega-byte 1MB = 1024 * 1024 bytes
[2]**GB**: Giga-byte 1GB = 1024 MB

using lambda is significantly lesser. The difference in cost attributes to the fine-grained billing of 100-ms versus hourly billing. Hence, lambdas are extremely cost-effective and powerful for solving general-purpose parallel computations.

4.3.5 Startup Latency

AWS Lambda executes in a container that provides protection and isolation to the executing function. Lambda creates containers and reuses them whenever possible. Every time a lambda is invoked, there is a small delay in container startup, language initialization, and code initialization. Cold start [10] refers to the increased lambda invocation time which occurs when the lambda function is invoked for the first time or after an extended period which does not benefit from container reuse.

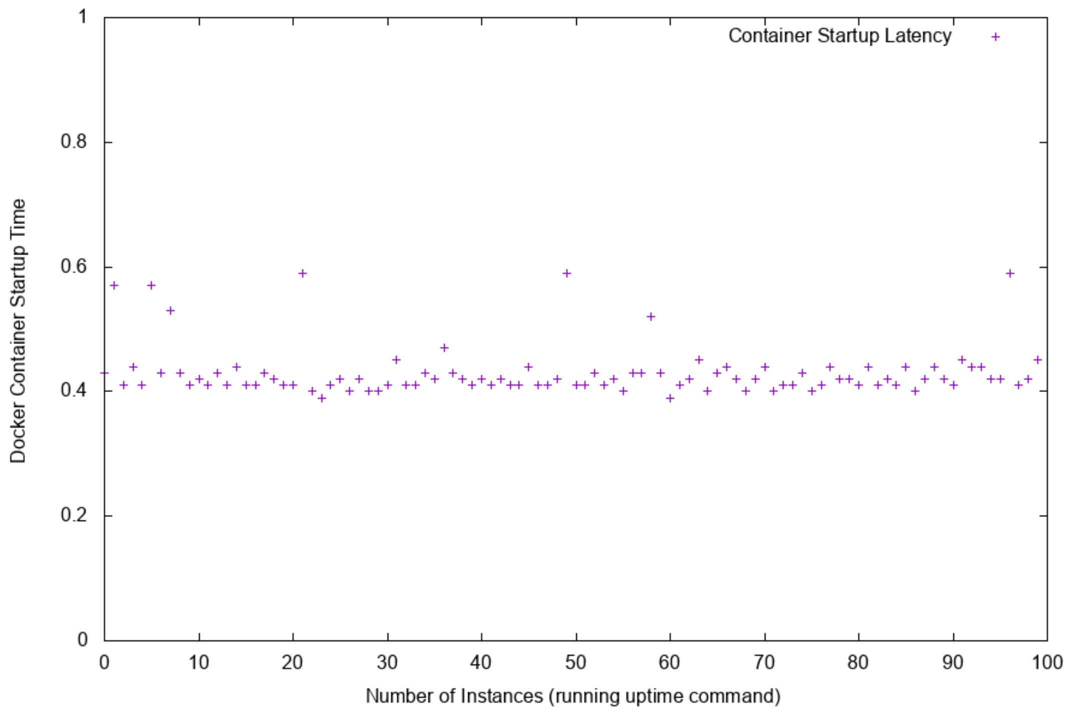

Figure 4.2: Startup latency of containers

From Amazon [11], if the code of the function is not changed and not too much time has gone by, Lambda can reuse the previous container for the current invocation. By reusing, it offers a significant performance by skipping the language initialization,

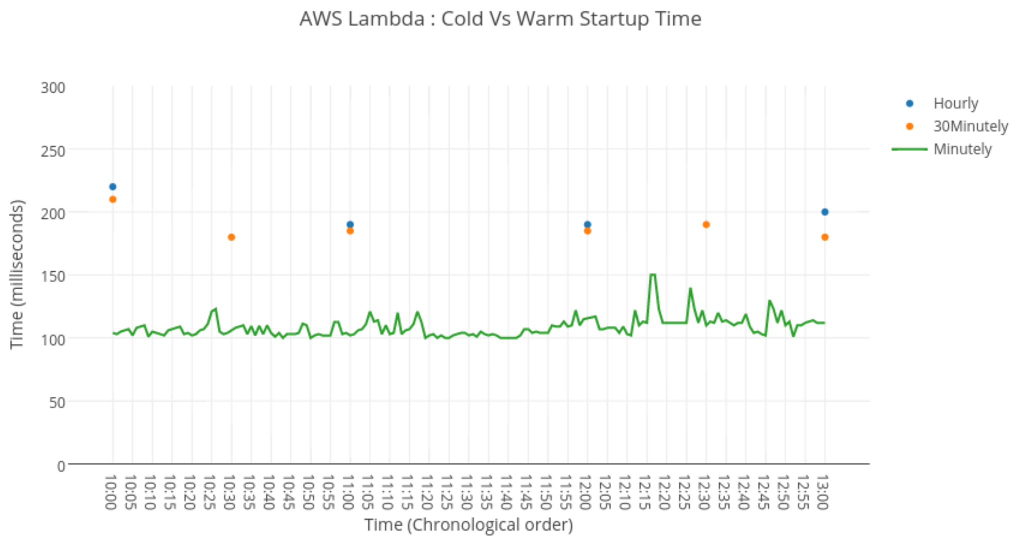

Figure 4.3: Cold start and warm start latency of AWS Lambda. Minutely refers to the lambda function invoked once every minute. 30Minutely refers to the lambda function invoked once in 30 minutes. Hourly refers to the lambda function invoked once in an hour

code initialization and the files in the temporary storage will still be available. This is referred to as warm start.

4.4 An orchestration framework for lambdas

With a large number of isolated and virtualized compute resources such as containers and microservices, managing the resources in a coordinated manner becomes a major challenge, creating the necessity for a high-level controller that could drive and monitor the application. With standard Hadoop and Spark clusters, YARN [30] controls job scheduling, fault-tolerance, synchronization and coordination among the different components of a job. Like YARN, there exists operating systems such as CoreOS [12], RancherOS [23] and Ubuntu Core [28] that provides support for containers and their orchestration. Systems such as Docker Swarm [13], Kubernetes [18] and Mesosphere [20] are designed as orchestration system for application containers.

What are the features of an orchestration framework? First, the ability to launch hundreds of little workers that wraps up the application and controls execution. Second, the capacity to launch, kill, restart and monitor the resources (containers or microservices) across different jobs submitted by the user. Third, support resource consumption across multiple hosts, availability, fault-tolerance, scalability in different dimensions, network communication, storage and integration with other services.

Most of the frameworks discussed above deal with application containers like Docker, LXC (Linux Containers) and containers running in the cloud and not targeted towards infrastructure like lambda functions. Since these frameworks are tightly coupled to the underlying computing resource, they lack portability in managing the lambda functions. Hence, this creates the need for a new orchestration framework that can schedule, launch, deploy and manage lambdas.

mu is a new orchestration framework written for managing lambdas [14]. The framework is tailored to run with AWS Lambda but can be extended to other microservices like Azure functions, and Google Cloud functions. It also supports the lambda function integration with storage service such as AWS S3. The fundamental concept of this framework is to spawn and manage lambdas, each processing a chunk of the data thus exploiting high degree of parallelism. The ability to launch lambdas instantaneously handles applications that are embarrassingly parallel such as video processing.

4.5 System Architecture

The mu framework is organized into several components. Figure 4.4 explains the architecture of the mu framework.

4.5.1 Controller

The controller is a long-lived coordinating server that creates a pool of lambda workers. The coordinator instructs the workers through standard Remote Procedure Call (RPC) request-response interface by sending HTTP requests over TLS connection. It also maintains the state of worker threads from creation till termination and displays the status of each worker.

There are three basic features of the controller. First, controller avoids deadlock between lambda workers during communication with dependencies by scheduling the

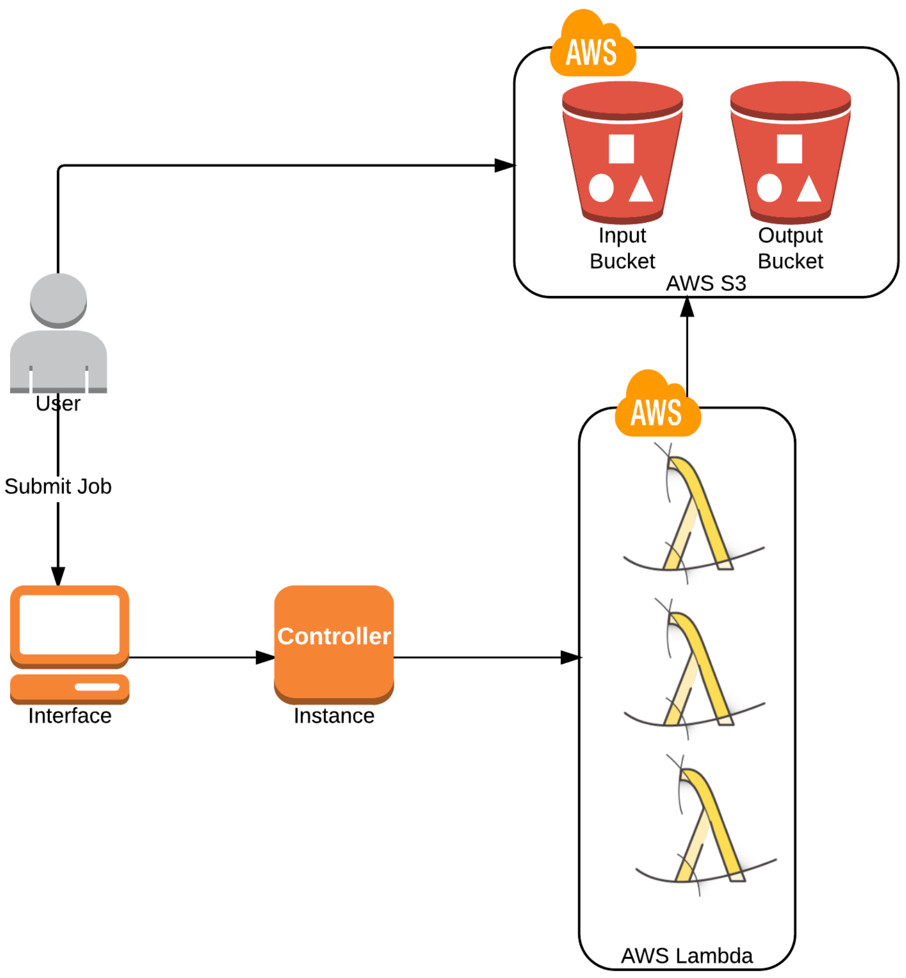

Figure 4.4: System Architecture of the mu framework

workers in order. Second, the controller instantaneously starts a large number of worker lambdas with the same lambda function. Third, the workers communicate with each other using a rendezvous server.

4.5.2 Service Endpoint

There are different ways to invoke a lambda function: AWS Software Development Toolkit (SDK), AWS Management Console and HTTP requests to the service endpoint. Each lambda function has an HTTP endpoint which can be invoked through standard HTTP requests (GET, HEAD, and POST). Every POST request includes a

JSON data structure containing the input to the lambda function. Each request creates a lambda invocation with the memory and timeout configuration specified during the lambda function creation.

4.5.3 Lambda Worker Pool

Workers are short-lived lambda functions running in the cloud as opposed to the controller. When the worker is invoked, it establishes a TLS connection to the controller. The controller sends messages containing the commands to the worker via a simple RPC interface, and the worker replies back with the return value and output via the same RPC interface.

The controller creates a pool of lambda workers which start from the prelaunch state. Once the lambda worker can begin execution, it moves to the active state. When the worker finishes and returns with (0) return value, it moves to the done state, else it moves to the error state. Figure 4.4 depicts the flow of the state machine of the lambda worker. Following are some of the characteristics of the mu framework.

- The lambda workers may spawn out of order.
- The lambda invocation created for the first time is called as cold lambda invocation. Cold lambda invocations incur a small latency in the order of milliseconds because of the run-time initialization. Recently invoked lambda can be cached and hence called as warm. Usually, warm lambda invocations are instantaneous to be fired up.
- Workers are behind a Network Address Translator (NAT). Workers use NAT-traversal techniques to communicate with each other.
- Controller instructs the worker with commands to be run and receives the response to the command through standard RPC interface.

4.5.4 Communication

The controller communicates with the worker through standard HTTP request/response synchronous communication over TLS. The controller sends the commands to be run in-

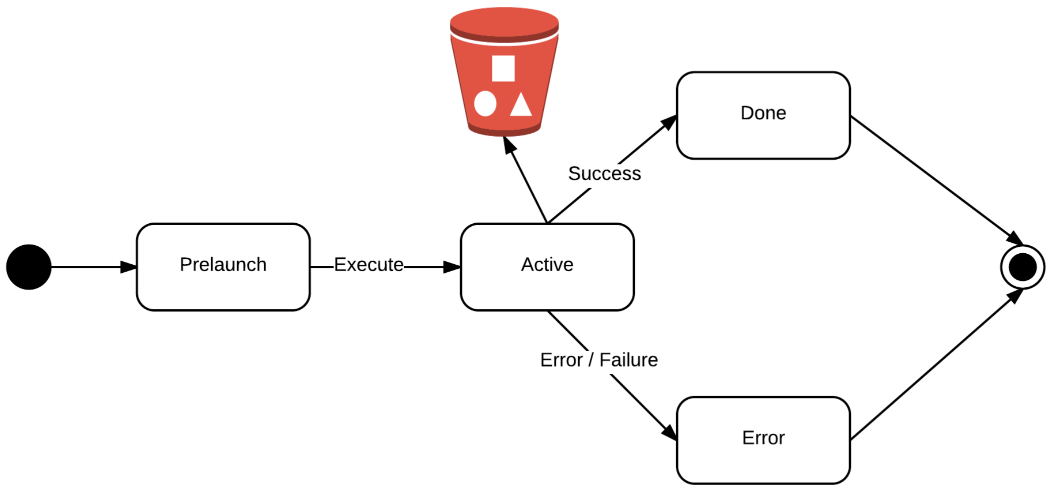

Figure 4.5: High-level State Machine of a controller in the mu framework

side the lambda as an HTTP message and the function that executes inside the lambda invocation receives the message and runs the command. The response contains the output of the command and the return value.

4.5.5 Throttling

AWS Lambda infrastructure imposes a limit on the number of concurrent lambda invocations on a region basis. Though Amazon does not reveal the implementation details of the lambda infrastructure, OpenLambda [36] has done analysis on the infrastructure that could be used by Amazon to support lambdas. Each lambda invocation spins a container that has the binaries of the function with the required run-time configuration set in it. The concurrency limit comes from the number of containers that can be scheduled, deployed and running simultaneously.

By default, each user gets 100 concurrent lambda invocations. Due to the demands of this research, the concurrency limits are increased to 1,200 lambda invocations per region. In addition to this, Lambda service runs in multiple regions, which increases the number of concurrent lambda invocations to the product of the concurrency limits and the number of regions supported. Any lambda invocation that crosses the limit gets an error indicating the throttling.

4.5.6 Data Partitioning

The interesting component of the mu framework is to deal with data partitioning across the several thousand lambdas. Each lambda worker is identified by a unique identifier which starts from 0 till N-1 (N: number of concurrent lambdas required). The controller computes the start and end of a chunk that has to be processed by the corresponding lambda invocation and instructs the lambda to download the chunk from AWS S3. In video processing, there are two kinds of input to be partitioned. First, a single video file in the mp4 format that has to be read by multiple lambda invocations to emit the images out of the video chunk. In this case, the worker computes the byte offset based on the duration of chunk to be processed. Second, AWS S3 holds thousands of images, and each lambda invocation needs a unique subset of images to be processed. In this case, the controller specifies the start and end image name or identifier to the lambda invocation.

4.5.7 Deployment

The controller is a long-lived server running on a dedicated machine which launches the pool of lambda workers and controls the execution till completion.

4.6 Data flow

The previous sections talk about the different components of the mu framework and their interactions. This section talks about the data flow in the system starting from the user submitting a video till the processed video getting streamed.

- User specifies the location of the video stored in AWS S3 bucket; the operator to be applied to the video (e.g., gray-scale); the amount of parallelism (number of lambdas).
- The mu framework invokes the appropriate controller, determines the stages for processing; finds the chunk size to be processed by each lambda.
- The controller creates a pool of workers.

- The lambda worker is given the chunk identifier, the command to run and any other meta-data needed by the controller.
- The controller communicates with the corresponding lambda worker through standard HTTP request/response messages.
- The controller monitors the state of the lambda workers and transitions them to the next state based on the output of the command run in the lambda.
- The controller exits when all the lambda workers exit by success or failure.
- The output video is produced in an AWS S3 bucket which can be streamed to the user.

4.7 Conclusion

This chapter discusses the need for microservices for large-scale video processing, the advantages, and disadvantages of using it. It is clear that a new orchestration framework is required in spite of the availability of a lot of frameworks. The mu framework is introduced which acts as an orchestration layer between the user and the AWS Lambda infrastructure. The data flow shows the efficiency of mu in managing thousands of lambdas running in the cloud.

4.8 Acknowledgements

The mu framework discussed in Chapter 4 is implemented by the co-authors Riad S. Wahby and Prof. Keith Winstein of the submitted publication. Material from Chapter 4 in part is currently being prepared for submission for the publication.

Chapter 5

Analysis of Grayscale pipeline

This chapter discusses how this research helps gray-scale a video efficiently regarding cost, time, space and user experience.

5.1 Grayscaling a video

Grayscaling is the most basic filter that can be applied to a video. A video in the compressed format can be grayscaled by grayscaling every image of the video. Figure 5.1 depicts a grayscaled image. This chapter explains the performance, bottlenecks, limitations and improvements of the system in grayscaling a single video.

The job of grayscaling a video has three tasks as discussed in Section 3.3. First, split the compressed video in mp4 format into a set of images in PNG format using FFmpeg [15]. Second, run the operator (gray-scale) on each of the image produced. Third, combine the gray-scaled images back into a video and stream the video back to the user.

5.2 Scheduling the stages of a grayscaling job

A video processing job can be typically broken down into three stages as described in Section 3.3 and the three stages can be executed in sequential or parallel fashion. Breadth-first scheduling refers to the execution of the three stages sequentially.

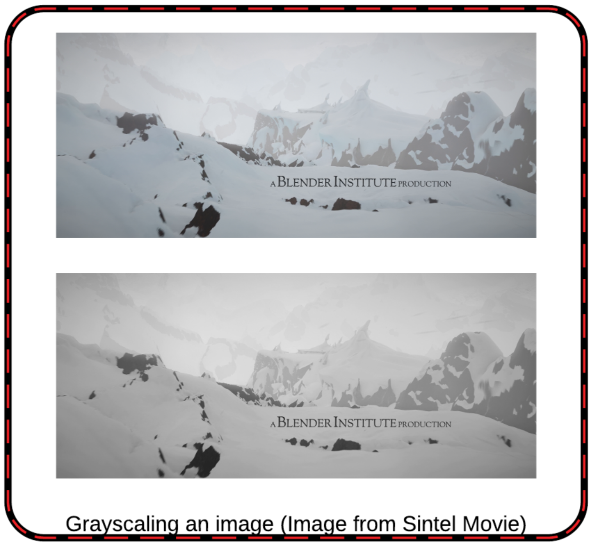
A BLENDER INSTITUTE PRODUCTION
A BLENDER INSTITUTE PRODUCTION
Grayscaling an image (Image from Sintel Movie)

Figure 5.1: Example of Gray-scaling an image (Image taken from Sintel)

The output of one stage is passed as the input to the next level. Parallelism is exploited inside a stage, in other words, this method implements intra-stage parallelism.

Each stage in the grayscaling has a controller implemented using the mu framework. The controller takes in the S3 bucket name, S3 keys to access, number of lambdas and size of the chunk. The controller implements a state machine with transitions and stores the per-state logic. Also, the controller assigns a unique identifier to the lambda worker running in the cloud. This unique identifier is used for assigning the chunks to the lambda worker. Lambda worker and lambda invocation both refer to the current instance of the lambda function and can be employed interchangeably.

Table 5.1: Different videos used for the grayscale experiments

Video Name	Duration in seconds	Pixel Format	Size	Frames/second
Whiteboard	185	420p	5MB	30
Lord	273	420p	50MB	24
Sintel	888	1k	149.3MB	24

5.3 Splitter

5.3.1 Configuration

The splitter stage is driven by a controller implemented in the mu framework. In this stage, the compressed video is given as input, and a set of images are produced as output. If the number of lambdas specified exceeds the concurrent limit exposed by AWS Lambda, the number of lambdas is lowered to the maximum concurrent limit available. The controller for splitter computes the chunk duration based on the duration of the video in seconds and the number of lambdas available. The chunk duration is computed as the fraction of video duration in seconds and the number of lambdas available. For example, from Table 5.2, the Sintel video runs for 888 seconds and running the splitter with 888 lambdas leaves each lambda to operate on a 1-second chunk.

Table 5.2: Work distribution for different videos used in Splitter stage

Video	Duration (sec)	Number of lambdas	Chunk Duration (sec)
Whiteboard	185	185	1
Lord	273	273	1
Sintel	888	888	1

5.3.2 State Machine

The controller creates a pool of lambda workers in the prelaunch state. Once the lambda worker is created, the worker is moved into the active state. The active state can be further broken down into a state machine as shown in Figure 5.3. The machine starts with **ConfigState**, where it sets the run-time environment configuration required for the lambda. If the configuration is set right, it transitions to the **SplitterState**, where the start point and duration of the chunk are determined based on the lambda worker number and video duration in seconds.

SplitterState runs the FFmpeg tool with the required input to emit the images out of the video chunk. Since FFmpeg is used with HTTP option, it issues byte range request to the video stored in S3 and downloads only the chunk that is required by the lambda thus saving storage and network bandwidth. Now, FFmpeg chops the chunk into

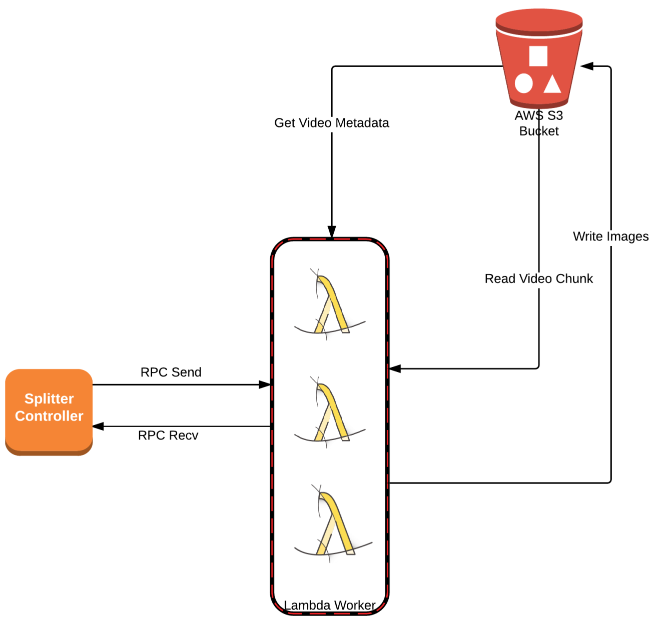

Figure 5.2: Splitter: Dataflow of the splitter stage

a set of images and writes to the local file system. The file system here is the temporary file system, that is ephemeral and local to the lambda invocation. For instance, if the chunk duration is 2 seconds and the video is encoded at 30 frames per second, there will be 60 images written to the local file system.

Once the FFmpeg command returns with success, the SplitterState transitions to the **UploadLoopState**. This state invokes the **UploadState** iteratively till all the images are uploaded to S3. UploadState is given the image to upload by the UploadLoopState, and it uses S3 API to upload the image. Once all the images are uploaded, the Upload-LoopState transitions to the FinishState, where the lambda worker is moved to done or error state based on the success of the workflow.

5.3.3 Micro-benchmarks

The following table documents the performance metrics collected from Lambda and S3. The controller runs on a dedicated machine with four processors of Intel(R) Xeon(R) CPU E5-2690 v2 3.00GHz. RTT refers to the round trip time measured in the network. RTT {A, B} relates to the round trip time between the endpoints A and B.

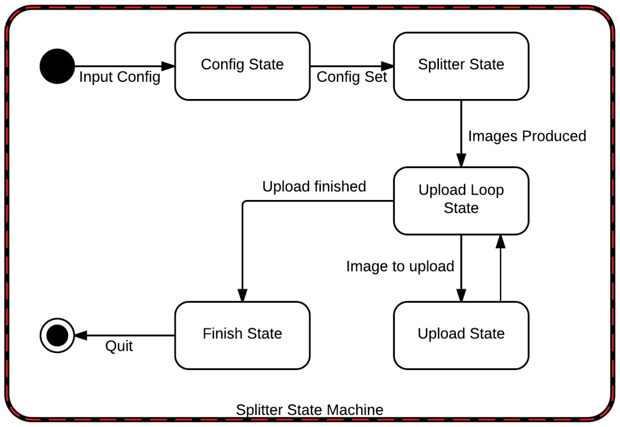

Figure 5.3: Splitter: State Machine of the controller

Table 5.3: Microbenchmark results from tests performed on AWS Lambda and S3

Metric	Time (milliseconds)
RTT {Controller, Lambda}	68ms
RTT {Lambda, S3}	60ms
TLS Connection Establishment {Controller, Lambda}	2 RTT {Controller, Lambda}
TLS Message Exchange {Controller, Lambda}	1 RTT {Controller, Lambda}
Lambda - File system write	0.03 seconds / 1MB of data

5.3.4 Bottlenecks

Bottlenecks in the splitter arise due to the cold start of lambdas, concurrency limits exposed by AWS Lambda and S3, network bandwidth, storage bandwidth and the bottleneck in the controller. As described in Table 5.2, X refers to the duration of a video in seconds, Y refers to the number of lambdas processing the video and Z refers to the chunk size, i.e., the fraction of video processed by a lambda.

Cold start [10] occurs when the container running the lambda function is not reused from previous invocations of the same function. From Figure 5.6, running the splitter for the first time has an increased latency in the fire-up state. The cold start adds approximately 100 milliseconds to each lambda to create the container, initialize the run-time and perform program initialization.

Network bottleneck refers to the overhead incurred due to the communication

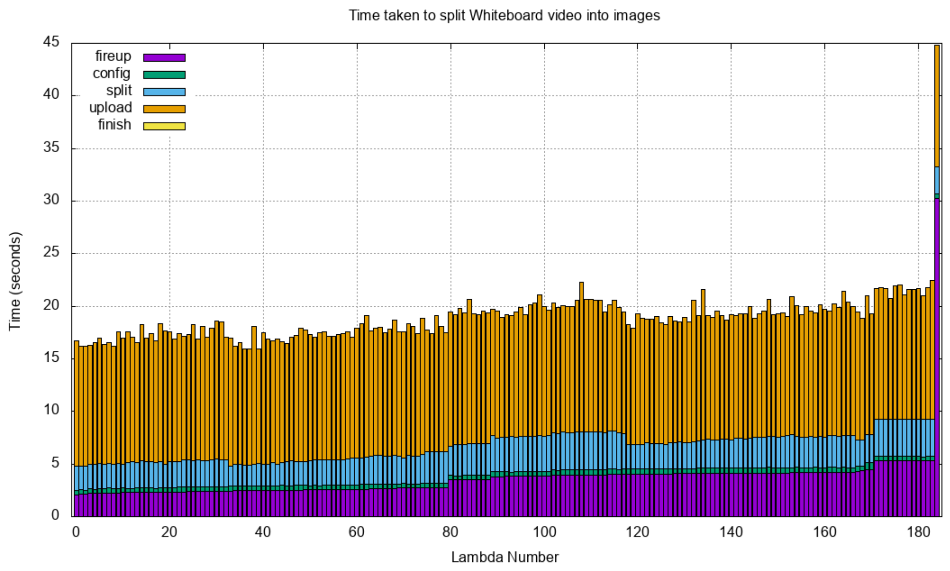

Figure 5.4: Splitter: Time taken to split a video (whiteboard) into images by each lambda

between the controller and the lambda worker. Once, the lambda worker is spawned, it establishes a connection with the controller which takes two round-trip-times according to TLS benchmark results [27]. Once the connection is established, the controller sends the command to be run inside the lambda as an HTTP message over the established connection. When the lambda finishes executing the command, it sends back the output of the command to the controller. Hence, for every command to be run inside the lambda, it exchanges two messages. The time to transmit the message is a function of the RTT between the controller and the lambda.

Storage bottlenecks can be classified into two: local file system writes and S3 uploads. From Figure 5.4, the split state includes the time to write data to local file system, and the upload state represents the time to upload the images to S3. Local file system writes significantly faster compared to the S3 uploads. It is evident that the S3 transfers are slow due to two reasons: the latency of transferring data from lambda to S3 and the number of concurrent read operations (800 operations per S3 bucket) and write operations (300 operations per S3 bucket) supported by S3. For example, from Figure 5.6 and Table 5.2, each lambda emits 24 frames. This results in 24 local file system write and 24 S3 uploads.

The *Controller* stores the per-state information, transition logic, and the com-

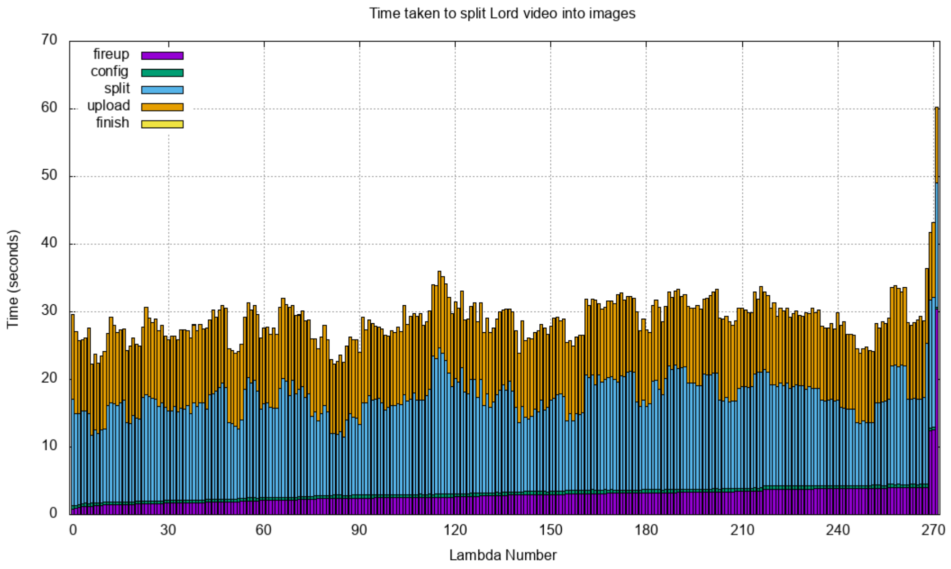

Figure 5.5: Splitter: Time taken to split a video (lord) into images by each lambda

mands to be run in the lambda. From Figure 5.4, splitter stage processes 1 second of video per lambda, hence the lambda executes 1 (configuration setup) + 1 (FFmpeg decode/encode) + M (s3 uploads) commands, where M is the frames per second of the video. For each command, the controller sends a message over the network. This turns out to be 2 + M messages per lambda. The experiment in Figure 5.4 runs 185 lambdas, each handling a second of video resulting in N * (2 + M) total messages communicated.

Stragglers are disproportionately long-running lambda workers that are either delayed in startup significantly or are being stuck in execution and takes a longer time to complete. Usually, stragglers in splitter task take five times longer than the median completion time to finish. This increases the average completion time of the job. Figure 5.5 shows a straggler that increases the time to job completion. There are different reasons for stragglers in the splitter task. **First**, due to the throttling by AWS Lambda, the requests sent by the controller are not yet served and are queued. **Second**, the lambdas in execution are queued by S3 when the number of outstanding read/write requests in flight crosses the concurrent limits by S3. From the experiments performed with three different videos, there is one straggler which took significant time to be started.

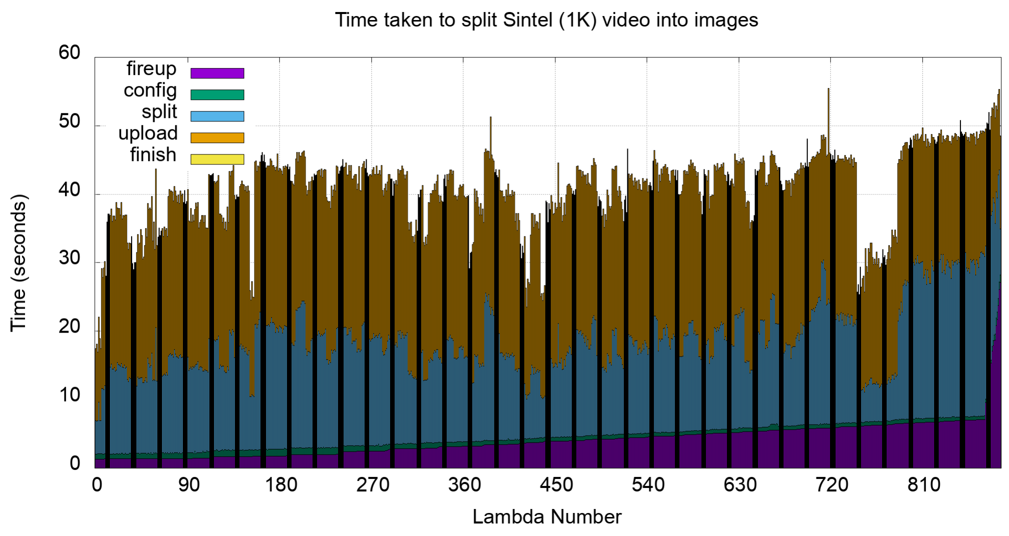

Figure 5.6: Splitter: Time taken to split a video (sintel) into images by each lambda

5.4 GrayScale

5.4.1 Configuration

In this stage, the images of the video are given as input, and a set of grayscaled images are produced as output using the mu framework. This stage is driven by a controller implemented in the mu framework. The grayscale controller computes the chunk size (number of images per lambda) based on the number of images to be processed and the number of lambdas available. If the number of lambdas specified exceeds the concurrent limits, the number of lambdas is lowered to the maximum limit available to the user.

Table 5.4: Work distribution for different videos in GrayScale stage

Video Name	Number of images	Number of lambdas	Images/Lambda
Whiteboard	5,550	786	7
Lord	6,528	816	8
Sintel	21,312	888	24

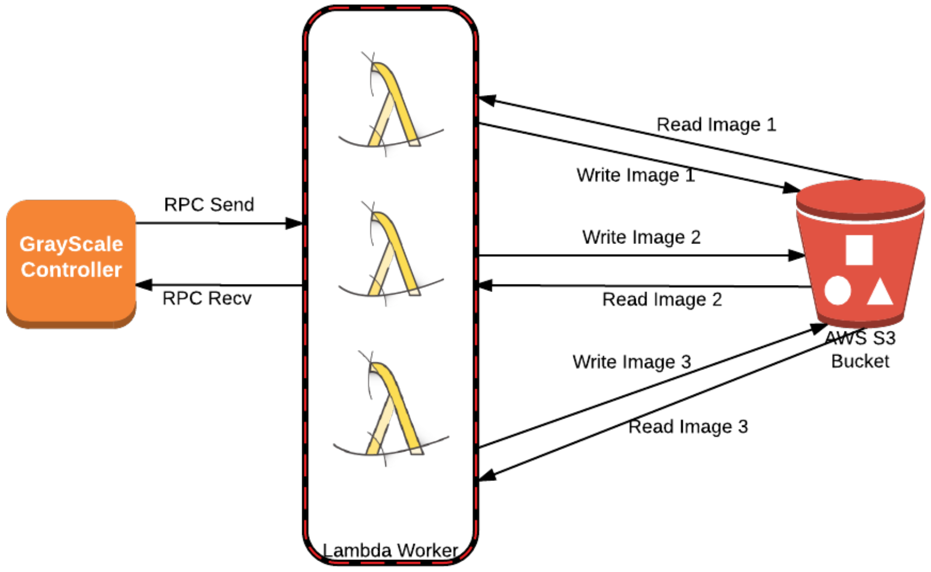

Figure 5.7: GrayScale: Dataflow of the grayscale stage

5.4.2 State Machine

The controller creates a pool of lambda workers in the prelaunch state. The controller implements a state machine with transitions that are executed by each lambda worker. The machine starts with **ConfigState**, where it sets the run-time environment configuration required for the lambda. Chunk size is determined based on the lambda worker number (actor number) and total images available. If the configuration is set right, it transitions to the **GrayScaleLoopState**.

GrayScaleLoopState invokes **GrayScaleState** till all the images of the chunk are grayscaled. For example, with the specified configuration in Table 5.4, each lambda worker gray scales ten images. GrayScaleLoopState invokes GrayScaleState 10 times to grayscale the chunk. GrayScaleState retrieves the image from the S3 bucket and stores in the local file system. Now, it runs the FFmpeg command to grayscale the image and writes the grayscaled image to the local file system. The GrayScaleState returns once the grayscaled image is uploaded to the output S3 bucket. Once all the images are grayscaled, GrayScaleLoopState transitions to the FinishState where it returns from the lambda worker.

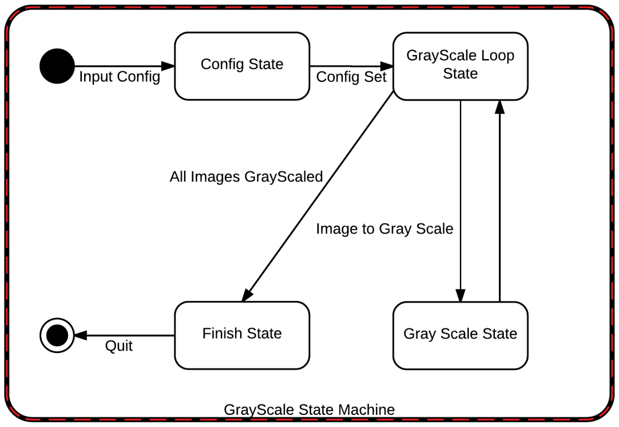

Figure 5.8: GrayScale: State Machine of the grayscale controller

5.4.3 Micro-benchmarks

The GrayScaleState (including GrayScaleLoopState) retrieves the image; executes the FFmpeg command to grayscale; writes the grayscaled image to the local file system; uploads the grayscaled image to S3. This state can be broken into three subtasks:

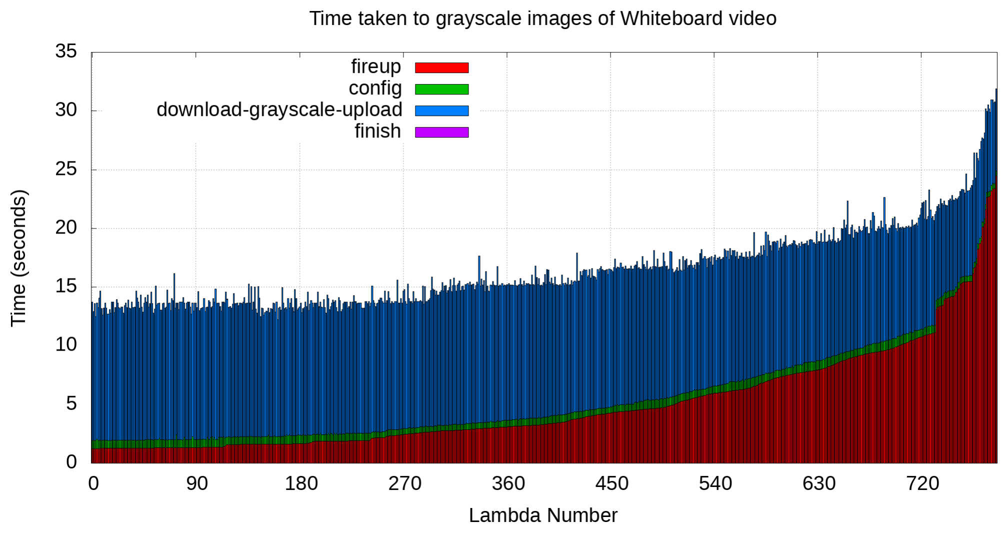

Figure 5.9: GrayScale: Time taken to grayscale the images of Whiteboard video

- Retrieve the image to be grayscaled from S3. The time to complete the request depends on the available network bandwidth between the lambda worker and the

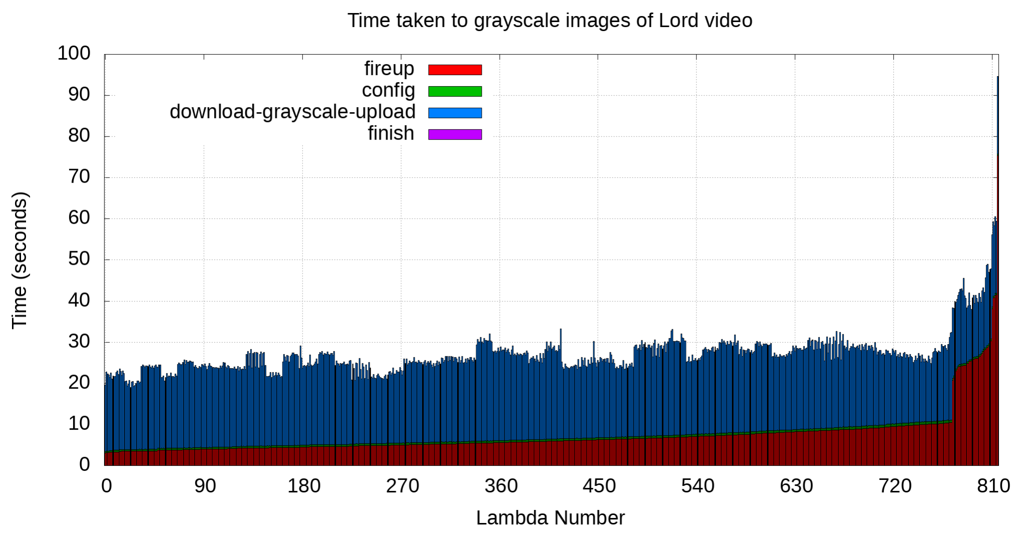

Figure 5.10: GrayScale: Time taken to grayscale the images of Lord video

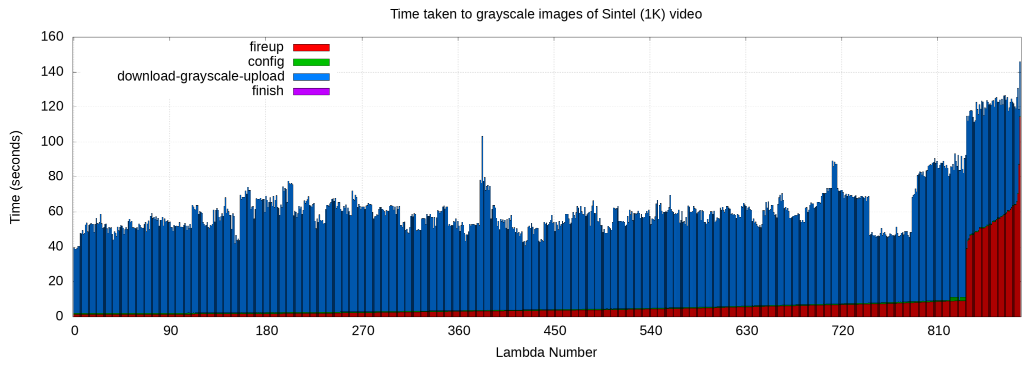

Figure 5.11: GrayScale: Time taken to grayscale the images of Sintel video

S3 service. This is usually a high-performance bandwidth link allowing fast transfers. This, in turn, has two operations: issue a GET request for the object and store the object in the local file system. The performance of the request is dependent on the network that connects lambda and S3. The benchmark tests run on lambda shows that it takes 0.03 seconds to write 1MB data into the disk. The average size of the image produced is about 1MB which results in N * 0.03 seconds for writing the images into the file system.

- FFmpeg runs to grayscale the image downloaded. Once the grayscaled image is produced, it is written back to the local file system.

- Upload the grayscaled image back to S3. The two operations here are local file system write and network transfer. These estimates are similar to the results discussed on retrieving images from S3.

To summarize the running time of a lambda worker includes startup, code initialization, download data from S3, write to local file system, grayscale the image through S3, write data to file system, upload data to S3. The major bottlenecks are network and concurrency limits.

5.4.4 Bottlenecks

Let N be the number of images processed by a single lambda worker. As discussed in Section 5.3.4, the bottlenecks of the splitter apply to the grayscale stage. The network bottleneck is high at this stage between the controller and the lambda. The controller sends three commands: retrieve image from S3, run FFmpeg, upload image to S3 to the lambda worker for each image to be grayscaled. The experiments performed shows that if there are N images to be processed per lambda, the controller exchanges 3N requests/responses with the lambda worker. The lambda executes N uploads and N downloads with S3. So the network includes 5N HTTP messages transported per lambda.

The controller has a higher overhead in maintaining the state of the lambda as well as handle more TLS connections due to the high degree of parallelism. This adds bottleneck in the controller in two fronts: the number of TLS connections to open and maintain and the amount of memory needed to store the per-worker state.

5.5 Combiner

The task of the combiner is to combine the grayscaled images back into a video and stream the video to the user. In this stage, the grayscaled images are given as input, and a single video is produced as output and streamed to the user. The combiner operation can be done serially or in parallel. In the serial merge, all the grayscaled images are downloaded, and the images are encoded into a video in mp4 format. This step does not use lambda and runs on a dedicated machine.

5.5.1 Bottlenecks

The serial merge is a bottleneck in the entire video processing job. However, due to the timing constraints of the research, the experiments for this research is focused on implementing the serial merge.

5.6 Streamer

There are different streaming protocols to stream the processed video to the user. The open standard to stream video is Dynamic Adaptive Streaming over HTTP (MPEG-DASH) [42] protocol. The DASH specification provides support for MP4 and MPEG-2 file formats. The video is partitioned into one or more segments and delivered to the client running in the browser using HTTP. A media presentation description (MPD) [42] contains the list of segments, and the information pertained to the segment.

5.7 Improvements

If the S3 service is replaced or removed, what will be the expected peak performance? From the discussion on bottlenecks, it is obvious that the reads and writes performed with S3 dominates the running time and becomes the bottleneck. For instance, with the splitter, several hundred lambdas reading a single object increases contention among the lambdas. If S3 is replaced with a better storage service or removed from the execution and rather relied on any in-memory store, the network and storage bottlenecks can be reduced. The running time of the stage will be just a function of the decode and encode operations and not on the reads and writes.

How to make lambdas fault-tolerant? A lambda may fail for several reasons: throttling by AWS Lambda, throttling by the other AWS resources such as S3, failure inside the lambda. Lambda provides retry mechanism if any of the commands run inside lambda fails. Alternatively, the mu framework can detect failures by checking the output of each command and re-run the lambda worker.

How to mitigate stragglers? Several straggler mitigation strategies [31] have been discussed which applies to micro-services as well. *Delay assignment* of the task to workers can reduce the contention among the resources accessed. For example, when the number of outstanding requests reaches 90% of the concurrency limits exposed, the request rate can be slowed down by queueing the requests internally.

Chapter 6

Conclusion

The work described included a broad range of topics starting from data-parallel frameworks, video processing, microservices, orchestration framework for managing AWS Lambda, performance analysis of video processing. In particular, this work focused more on the AWS Lambda among the different microservices framework available. Microservices orchestration seems to be a new area of research because of the lack of a reliable yet customized framework.

The grayscale pipeline discussed gives better insights into the data flow and interactions that occur in the system. Understanding the bottlenecks in the system helps design scalable and fault-tolerant video processing system using lambdas. The analysis reveals that lambdas are cost-effective compared to using virtual machines.

The system implemented achieves the requirements set for the project. However, there are numerous components in the system that could be improved. First, replacing the controller with a distributed controller will remove some of the bottlenecks associated with the controller. Second, a straggler mitigation policy can reduce the average completion time of the job thus improving the reliability of the system. Third, implementing retry mechanism for lambda can improve the correctness of the system without compromising performance. Last, running the controller in a serverless architecture such as a container in the cloud will make the entire system serverless.

Though multiple aspects of the system are addressed, this work just touches the surface of the ocean of problems to be solved with parallel video processing. Developing scalable and reliable microservices orchestration frameworks would be a more challenging and exciting problem to solve to meet the current needs of the users. Microservices have been out on the market just for the last few years. It will be exciting to see how the microservices architecture help transforms long-running high-performance servers to ephemeral supercomputing service running in the cloud.

Bibliography

[1] Amazon Elastic Compute Cloud. https://aws.amazon.com/documentation/ec2/.

[2] Amazon Web Services. https://aws.amazon.com.

[3] Apache Spark. http://spark.apache.org.

[4] Apache Storm. http://storm.apache.org.

[5] AWS EC2 Pricing. https://aws.amazon.com/ec2/pricing/on-demand/.

[6] AWS Lambda. https://aws.amazon.com/lambda/.

[7] AWS Lambda - Run code in the cloud. https://aws.amazon.com/blogs/aws/run-code-cloud.

[8] AWS Lambda Pricing. https://aws.amazon.com/lambda/pricing/.

[9] AWS S3. https://aws.amazon.com/s3/.

[10] Cold start optimization. https://blog.newrelic.com/2017/01/11/aws-lambda-cold-start-optimization.

[11] Container reuse in AWS Lambda. https://aws.amazon.com/blogs/compute/container-reuse-in-lambda/.

[12] CoreOS. https://coreos.com/.

[13] Docker Swarm. https://www.docker.com/products/docker-swarm.

[14] Excamera - mu framework. https://github.com/excamera/mu/.

[15] Ffmpeg. https://ffmpeg.org.

[16] Google Cloud Platform. https://cloud.google.com.

[17] Hadoop. http://hadoop.apache.org.

[18] Kubernetes. https://kubernetes.io/.

[19] Mapreduce architecture. https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html.

[20] Mesosphere. https://mesosphere.com.

[21] Microservices. https://techbeacon.com/containers-microservices-how-modernize-legacy-applications.

[22] Microsoft Azure. https://azure.microsoft.com.

[23] RancherOS. http://rancher.com/rancher-os/.

[24] Sintel. https://durian.blender.org/.

[25] StormCV. https://github.com/sensorstorm/StormCV.

[26] Thunder. http://thunder-project.org/.

[27] Transport layer security benchmarking. https://hpbn.co/transport-layer-security-tls/.

[28] Ubuntu Core. http://developer/ubuntu.com/en/snappy/.

[29] Using Hadoop MapReduce for Distributed Video Transcoding on a Large Scale. https://content.pivotal.io/blog/using-hadoop-mapreduce-for-distributed-video-transcoding.

[30] YARN. https://hortonworks.com/apache/yarn/.

[31] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Effective Straggler Mitigation: Attack of the Clones. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 185–198, Lombard, IL, 2013. USENIX.

[32] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[33] 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)*, Boston, MA, 2017. USENIX Association.

[34] Raymond Greenlaw, H. James Hoover, and Walter L. Ruzzo. Limits to Parallel Computation. https://homes.cs.washington.edu/ ruzzo/papers/limits.pdf.

[35] Alexey Grishchenko. Spark Architecture. https://0x0fff.com/spark-architecture.

[36] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Serverless Computation with OpenLambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, Denver, CO, 2016. USENIX Association.

[37] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. *SIGOPS Oper. Syst. Rev.*, 41(3):59–72, March 2007.

[38] Ming Mao and Marty Humphrey. A Performance Study on the VM Startup Time in the Cloud. In *IEEE CLOUD*, pages 423–430. IEEE, 2012.

[39] Thomas B. Moeslund. *Introduction to Video and Image Processing: Building Real Systems and Applications (Undergraduate Topics in Computer Science)*. Springer, 2012.

[40] Rafael Pereira, Marcello Azambuja, Karin Breitman, and Markus Endler. An architecture for distributed high performance video processing in the cloud. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 482–489. IEEE, 2010.

[41] Timo Saarinen. Container-based video processing. Master's thesis, School of Science, Aalto University, 2015-06-10.

[42] Iraj Sodagar. The MPEG-DASH Standard for Multimedia Streaming over the Internet. *IEEE MultiMedia*, 18(4):62–67, October 2011.

[43] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the Condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.

[44] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, San Jose, CA, 2012. USENIX.

[45] Weishan Zhang, Pengcheng Duan, and Qinghua Lu. Towards a load-aware scheduling framework for realtime video cloud. In *Identification, Information, and Knowledge in the Internet of Things (IIKI), 2015 International Conference on*, pages 1–6. IEEE, 2015.