Big Data Platforms

Stragglers and Serverless Computing

Ashley Newman (806092), Ophira Blumner (330271933), Almog Friedlander (204735351)

28.2.2022

1. Abstract

Substantial cost-saving cloud computing services such as serverless systems are vulnerable to straggler tasks, which are slow processing nodes. Straggler tasks are essentially those tasks that run much slower compared with the other tasks in the job. Since a job is completed only when it has finished all the tasks, straggler tasks can have a significant, deleterious effect on the processing time of a job. These straggler tasks can significantly lengthen computation time. An increase in computation time refers to the time that it takes to get a response of a service across the network once a job is triggered. The straggler problem therefore gets worse with increased parallelism: the greater the number of servers, the higher the likelihood of having a straggler occur in any iteration. We propose two relatively straightforward, yet efficient solutions to mitigate straggler tasks in serverless systems. The first is an extension of speculative execution (with an added waiting time continuing launched tasks after the last processed byte) and the second is based on cloning tasks at the start of a job.

2. Motivation and background

2.1 *Describe the relevant background, putting emphasis on what are straggler tasks.*

Many computational problems a developer may encounter can be divided into multiple sub-problems. This practice is especially efficient when the original task is large, as the sub-problems can be solved at the same time, resulting in a faster completion time than solving them sequentially.

This approach of parallel computing relies on the ability to break down a large job into many smaller tasks, each processed by an independent entity such as a thread or a worker. As such, the original job is considered complete when the last of the small tasks is done - wherein lies one of the practice's weak points, as a single lagging task could severely hinder the completion of a job even if all the other tasks have finished. This lagging worker is called a straggler. [1]

2.2 ***Explain what is the root cause of their appearance.***

Slow processing nodes, straggler tasks, result from system noise in cloud-based systems. System noise arises from the limited availability of shared resources, network latency, and hardware failure. One of the most common causes of straggler tasks is data skewness. Data skewness is caused by skewed partitions, which occur when an unequally distributed amount of data is mapped to each partition. Then the larger skewed partitions lead to a longer computation time among tasks assigned for them, giving rise to stragglers. For example, a skewed partition could result from using join on a key that is not equally distributed across a cluster, resulting in some very large clusters so that the computing service is not able to process the data in parallel. Another example is if numerous files, unable to be split and therefore skewed by size, are then read into a dataset. [2]

Another prevalent cause of straggler tasks is skewed computation. Skewed computation occurs even when the data is evenly distributed among partitions, but the computation becomes skewed based on certain attributes of the data. One example is when a dataset points to a collection of file wrapper objects. In one of the partitions of this dataset, all corresponding file wrapper objects refer to larger files than in the other partitions, resulting in a straggler task. [2]

Stragglers can also potentially result when some of the tasks are hosted on a server that is suffering from slow disk read or write operations. For example, if a job executes a shuffle read or write operation, or if a job needs to save an intermediate dataset onto the disk, straggler tasks can result. [2]

One more cause of stragglers is a larger number of cores per executor. An example of when this might happen is if there is a simultaneous heavy demand from tasks on all the cores of an executor, leading to a struggle for common resources like memory. [2]

2.3 ***Provide examples and explain how different Big Data engines deal with straggler tasks.***

Two well-known Big Data engines, Apache Spark and Hadoop MapReduce, have their own processes to deal with straggler tasks. They both use a traditional technique known as speculative execution to mitigate stragglers.

In Hadoop, MapReduce works by segmenting jobs into stateless, idempotent map and reduce tasks, which run in parallel on multiple machines instead of sequentially, thus reducing execution time. This makes MapReduce hypersensitive to slow tasks, stragglers, even if there are only one or two since these stragglers will delay the processing of the entire job. As discussed previously, there are many causes of straggler tasks. The source could be a software misconfiguration, network congestion, a hardware failure or something else. Due to the plethora of causes of straggler tasks, Hadoop MapReduce's solution to stragglers does not aim to diagnose or even fix them. Rather, it detects when a task is running slower than expected and spawns a duplicate, back-up task. Specifically, MapReduce launches backup tasks for those tasks whose progress is slower, on average, compared with other tasks from the same job. If the original task finishes before the backup task, the backup task is deleted. If the backup task finishes before the original task, the original task is deleted. [3] In summary, the default scheduler in Hadoop is constantly monitoring and detecting stragglers all while running computational tasks. Once a straggler is found, the straggler task is relaunched at some other available node.

Apache Spark is also a parallel processing framework. When a job is submitted in Spark, it is broken down into many stages (stages are a logical grouping of tasks). Each stage creates many tasks that will execute on each node. Some stages may be dependent on previous stages to complete. This leads to a significant problem when a task from a previous stage is lagging, so the subsequent stages will also be delayed. The larger the cluster and number of stages, the greater the computational bottleneck, and the greater the job completion time. Straggler mitigation in Spark is based on speculation. A task can be considered for speculation mode if it is running one

and a half times slower than the median task. This cut-off is set according to the spark.speculation.multiplier, which defaults to 1.5. Before speculation is initiated at a given stage, 75% of tasks must finish. This limit is given by spark.speculation.quantile, which defaults to .75. Once these parameters are met, speculation mode will launch a new task in parallel and it will take the results from whichever task finishes first. [3]

It is necessary to note that while speculation is the approach employed by Hadoop MapReduce and Apache Spark and it does successfully mitigate stragglers, it still has several disadvantages. The first is that the identification of stragglers requires a constant monitoring of jobs, which can be costly if there are many workers in the system. There is also the possibility of a scenario where a task is marked as a straggler just before completion, consuming an exigent amount of overhead with additional invocation and setup. This situation becomes even more impactful for smaller jobs. [4]

3. Stragglers in serverless computing

3.1 *Explain what is serverless computing and the problem of straggler tasks in serverless Paradigm.*

Serverless computing is a distributed cloud computing paradigm in which the cloud provider manages the servers the computations are done on. In that sense the name “Serverless” is not accurate as servers are indeed used to execute the code, but the management and maintenance of the servers is done by the cloud provider and not by the developer.

This approach holds many advantages for the developers, as they are not concerned with configuration, capacity, fault tolerance, physical hardware etc. In addition, computations on serverless platforms are done in short execution sessions, and when the program is not running, no resources are being consumed and budget is saved.

One of the main benefits of cloud computing stems from the ability of parallelizing multiple workers, each dealing with a small task, which together comprise a job. This distributed approach enables shortening running times for the job as a whole, but one of the main challenges encountered in cloud computing is the large variance in completion time for each task due to a

combination of hardware failure, limited availability of shared resources, network latency, and other issues. The small portion of tasks that take much longer to complete than the median task time are called stragglers, which are responsible for slowing the finishing time of the job, as a job only finishes when the last task is done.

3.2 *Be specific and address various points on difference and similarity of the straggler tasks in serverless computing vs other cluster based Big Data engines.*

Straggler tasks are a potential roadblock for all big data processing engines, from Apache Spark and Map Reduce to serverless engines like Lithops. Unlike the high-performance computing Big Data engines Apache Spark and Hadoop, serverless computing faces more challenges with stragglers. Traditional approaches, like speculation and blacklisting machines likely to cause difficulties, try to solve the straggler problem by detecting stragglers in the first place before fixing them. Monitoring tasks can pose a challenge in serverless systems since worker management is done by the cloud provider and the user does not have explicit control over the workers.

4. Stragglers in jobs submitted by Lithops

4.1 *Explore and explain how Lithops submit its job and knows which tasks are finished and which tasks yet running.*

Lithops is a python framework that allows its user to scale his or her multiprocessor applications in the cloud using a serverless computing platform in the background. Serverless allows the execution of computational units called functions in the cloud. Behind the scenes, a host submit is initiated once Lithops is running on the host's computer, where it submits the serialized host's python function and serialized data and stores this information in a cloud object storage service, COS (e.g. AWS S3 or IBM COS). A job is submitted when Lithops invokes the cloud stateless function. On the server's end, the relevant function and data are taken from the object storage where they are invoked, serialized, and then placed back into the object storage at a pre-specified key. The existence of this pre-specified key signals the completion of

a job. Monitoring task progress is determined by the cloud service provider, (e.g. CloudWatch in AWS S3), so the user has no direct supervision over the workers. [5]

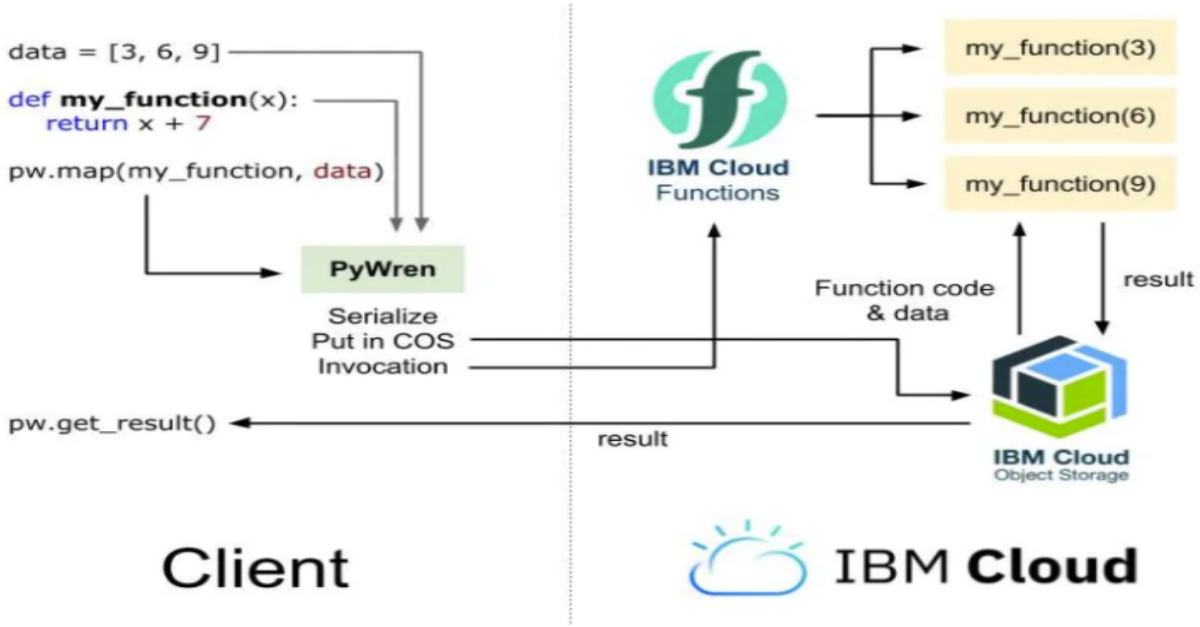

Figure 1: Invocation of a Function in Lithops, formerly known as PyWren (Sampe, 2018) [5]

4.2 Explain how Lithops knows when entire job is finished and how it waits for the running tasks to complete.

As stated previously, Lithops runs MapReduce tasks as serverless functions. Specifically, Lithops has a "serverful function scheduler" that allows map computations to execute as stateless functions through a simple API. For the map code to be run in parallel it is first serialized and is then stored in a cloud object storage service. Next, Lithops invokes a function that de-serializes the map code and executes it on the relevant data from the cloud object storage. After execution, the results are placed back into the cloud storage. The scheduler is therefore the way that Lithops knows when an entire job is finished since the scheduler actively checks the cloud object storage to detect that all partial results have been uploaded to before signaling the completion of the job. [2]

4.3 Provide two different approaches to extend Lithops with mechanism to prevent jobs to be affected by straggler tasks. Address different aspects of complexity of implementation, costs, performs. Explain positive and negative effects of the approaches you propose.

4.3.1 Approach 1 – Speculative-Resume (an extension of speculative execution)

The techniques to mitigate stragglers in serverless computing broadly belong to two categories: blacklisting and replication-based techniques. Blacklisting's aim is to detect machines that exhibit high variability and avoid using those faulty machines or to learn task-to-node pairings that lead to high variability and boycott those pairings. Nevertheless, despite using blacklisting, stragglers still regularly occur on non-blacklisted machines (due to I/O contention, network latency, and more). Replication-based approaches work by either relaunching a straggler task or by preemptively assigning each task to multiple nodes and then taking only the results of the task that finishes first, discarding the other. Our proposed approach is based on an extension of speculation. Speculative task execution continues to be the most widely deployed solution for straggler tasks for a reason. Computation redundancy, i.e., the approach used in speculative execution, works well for the stateless tasks of map reduce jobs and thus, provides a good starting point for our approach to extend Lithops' straggler task mitigation strategy.

Aktas, et. al, proved that with a replication-based strategy, employing a wait time into the strategy improves redundancy as long as there is a heavy tail on the time distribution of the tasks, which in fact, is the case in serverless systems as task execution times in parallel computing systems are known to exhibit a heavy tail because of the reasons mentioned in 2.2. Redundancy can lead to higher costs and should be considered with caution. For this reason, we propose optimizing a replication-based approach based on straggler mitigation and weighing its success against the cost of employing such an approach. We define cost as the total resource time spent while executing a job. An approach with a higher cost will lead to a greater crowding of system resources, consequently worsening job slowdowns or even driving the system to instability. Aktas, et al. showed with a queuing theoretical analysis how excessive replication of single-task job arrivals can drive a system to instability. [2]

Another reason to support employing a waiting time is because adding a waiting time increases the change of identifying stragglers in the first place. For instance, a task may appear to be lagging initially compared to the other tasks but could catch up to the median after an initial lag. We suggest a speculative-resume strategy to mitigate stragglers while keeping the cost low. The strategy is illustrated in Figure 2.

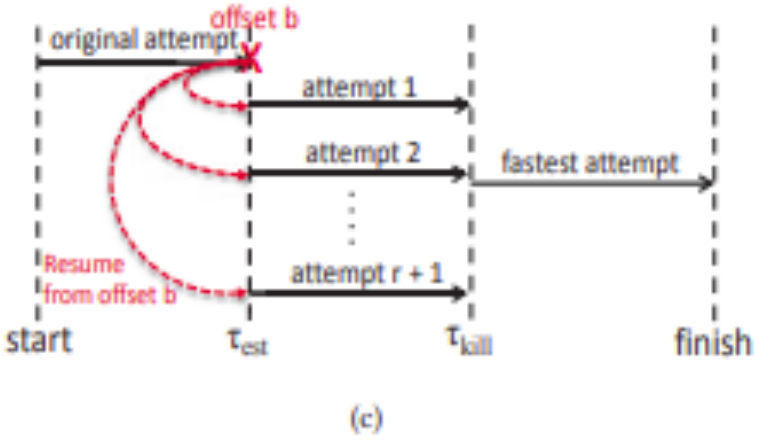

Figure 2: Illustration of Speculative Resume (Xu, et. al , 2017 [6])

Figure 2: Speculative-Resume Strategy (taken from Xu, et. al, 2017 [6]). T_{test} is effectively the wait-time before launching speculation. Offset b is the byte offset where the extra attempts start processing.

At the start of speculative-resume, a single attempt is launched for each task. The estimated completion time of each task is given by T_{test}. In our example, we set T_{test} to 60 seconds for simplicity. If T_{test} exceeds D (deadline of all tasks set by the user), the straggling task is killed and r+1 (set by the user) attempts are launched for the straggling task(s). For simplicity, we set D to 200 seconds, meaning that all tasks that are not completed at 59 seconds will be 'killed' and relaunched at 60 seconds. For simplicity of the model, we assume relaunch occurs instantaneously. In Speculative-Resume, the relaunched tasks do not reprocess the data that has already been processed by the original attempt. Instead, the relaunched tasks process the data starting from the byte after the last byte processed by the original straggler task. Avoiding the

loss of processed data in the killed straggler node leads to a lower overall cost. The progress scores of all the relaunched attempts are checked by T_{kill}, which we set to 120 seconds. Only the relaunched attempt with the smallest estimated remaining completion time is left running while the other r attempts are killed. Once the remaining task completes, the final results can be given.

We decided in favor of the Speculative-Resume strategy based on the research done by Xu, et. al (2017)[6]. In the paper, three replication-based strategies were tested by executing 100 MapReduce jobs and evaluating them based on the Map phases of sort and word count. The three strategies that were tested are cloning (which we will describe in approach 2), speculative execution (where after a certain time, the lagging task is marked and restarted r+1 times. The lagging task continues, and whichever task has the fastest progress is finished to completion), and Speculative-Resume. Based on the results, the clone strategy had the highest straggler mitigation score in Sort and the second highest straggler mitigation score in Wordcount, but it incurred the highest 'cost' of all three strategies since all attempts are launched at the start. Therefore, when taking the highest straggler mitigation score minus the cost, the Speculative-Resume strategy was the most successful across the board. Based solely on performance, Speculative-Resume received a score of 94% for straggler mitigation in Sort of MapReduce tasks (cloning had 97% and speculative execution had 96%) and 85% in Wordcount of MapReduce tasks (cloning had 81% and speculative execution had 79%). [6]

The key reasons for the significant improvement of stragglers in our strategy are due to launching multiple attempts (r+1) for the straggler task and having limited execution time for these r+1 tasks by having a threshold, T_{kill}, where only the fastest progressing task is kept alive. Implementation of the strategy is relatively straightforward. We recommend having defaults set for D, T_{test}, r, T_{kill}, and the user can alter these, if desired. This would be similar to Apache Spark's spark.speculation.multiplier and spark.speculation.quantile. One intricacy of implementation would be joining the processed data from the straggler task with the remaining processed data from the relaunched task, since the processed data are most likely processed on different nodes in a parallel computing framework and intermediary data is stored in cloud object storage. Another challenge, particularly in Lithops is that Lithops interacts with third party API's

which launch all the tasks. Therefore, Lithops would have to implement a strategy for straggler mitigation that could transfer to all these third parties.

One drawback from our Speculative-Resume approach is that there is no worst-case guarantee since both the original straggler task and the replicated ones could be delayed. This is something to consider for all replication-based approaches.

Up until this point we have ignored the impact of redundancy on the system. Redundant tasks exert extra load on the system, which is likely to aggravate the existing contention in the system resources. Given that resource contention is the primary cause of runtime variability [4], the added redundant tasks are likely to increase the variability in task execution times, possibly worsening the straggler problem. Although this is less of a concern in a serverless system, it is still a possibility.

4.3.2 Approach 2 – Dolly (an extension of cloning)

As mentioned in the previous section, the second approach for straggler mitigation we have chosen to discuss is task cloning. Whereas Speculative-Resume proposes an extension of the current standard of straggler mitigation (speculative execution), task cloning does away with speculating and waiting altogether. Instead of trying to predict stragglers in a reactive manner, we propose proactively creating multiple clones of every task a job is comprised of, and only using the result of the clone that finishes in the shortest amount of time.

Another approach to cloning is job-level cloning, where several clones are submitted for the entire job, and results are taken from the first job that finished. While this approach is easier to implement, we can see in Figure 3 that the probability for the job to straggle decreases at a slower rate when compared to task-level cloning.

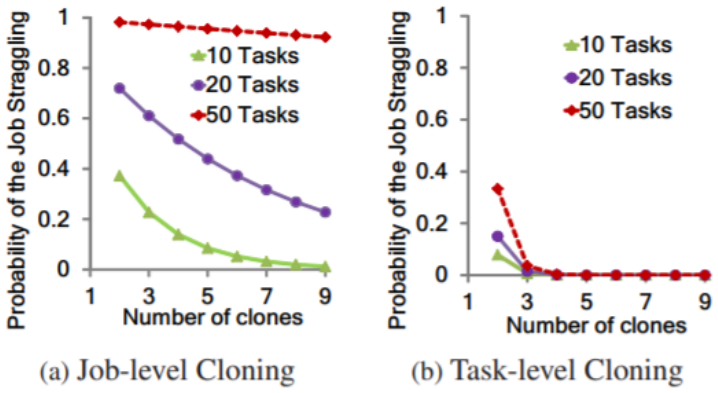

Figure 3: Probability of a job straggling for varying number of clones (taken from *Ananthanarayanan*, et. al, 2013 [1])

Figure 3: Probability of a job straggling using cloning strategy (taken from *Ananthanarayanan*, et. al, 2013 [1]). When comparing job-level cloning and task-level cloning we can see that the latter requires fewer clones in order to achieve the same probability of the job straggling.

There are two issues with task cloning that must be addressed. First, launching multiple clones of each task could use up an excessive amount of the limited resources available. However, we have seen that in the case of serverless systems, task execution times have a heavy-tail distribution with the smallest 90% of jobs consuming as little as 6% of the resources. The interactive jobs whose latency we seek to improve with this approach all fall in this category of small jobs. [1] This means that using a few extra resources will not significantly impair the performance of task cloning in the mitigation of stragglers but will aid in accelerating most of the jobs executed overall in the organization. [1]

The second challenge task-level cloning presents is the possible contention clones could create on intermediate data. When tasks in the first phase of a job finish executing, they pass along this data to tasks in the next phase and so on until the job is complete (e.g., map, reduce, join). Task cloning proposes launching multiples of tasks of different phases, which means that multiple downstream tasks will attempt to read the results of the earliest upstream clone. This can create contention at the upstream clone, an issue that can be dealt with in several ways.

Contention Avoidance Cloning (CAC) – In this approach each upstream clone has n designated downstream clones, one in each of the n clone groups, to which it will transfer the results. This means the other n-1 downstream clones in each clone group must wait for another upstream clone to finish. The inherent disadvantage of this approach stems from the fact that if an upstream clone straggles, its downstream path will also lag.

Contention Cloning (CC) – in this approach all downstream clones are able to read the data from all upstream clones, and they wait for the earliest one from each clone group from which they'll receive the data. This means all downstream clones could in theory start their computation at the same time, but in reality, some or all of them could be delayed due to contention on disk or network bandwidth constraints.

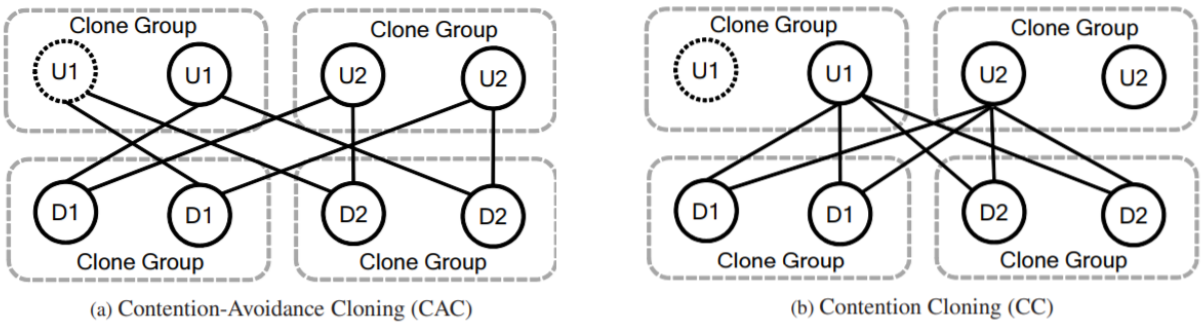

Figure 4: Intermediate data contention (taken from *Ananthanarayanan*, et. al, 2013 [1])

Figure 4: Comparison between CAC and CC for intermediate data contention (taken from *Ananthanarayanan*, et. al, 2013 [1]). The example job contains two upstream tasks (U1 and U2) and two downstream tasks (D1 and D2), with a cloning factor n = 2. One clone of U1 is a straggler - marked with a dotted circle. As explained, CAC will wait for the straggling clone while CC opts for the earliest clone.

When comparing the probability of a job straggling with CAC and CC with a cloning factor n = 3 there is an increase of over 10% for a job of 10 tasks and 30% for a job of 20 tasks, making CC the preferred approach. The gap between the straggling probability diminished as the cloning factor grows but this counteracts our aim to lower the cloning factor to minimize computation overhead.

Utilizing the CC strategy, as described above, causes contention for IO bandwidth as every downstream task reads a part of the output from each of the upstream tasks, at the same time. If no stragglers occur, there would be the same number of copies of the upstream output as the number of downstream clones attempting to read them. However, with stragglers, the number of downstream clones would be larger than the available upstream outputs and it will result in contention.

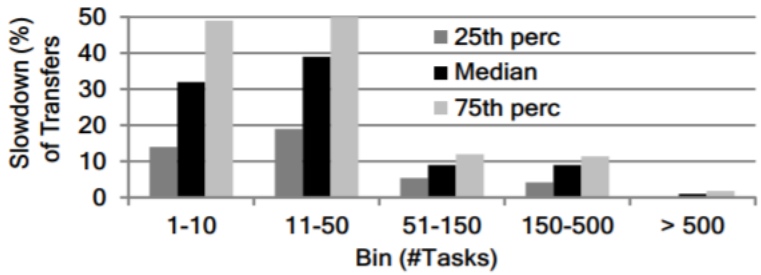

Figure 5: CC contention slowdown (taken from *Ananthanarayanan*, et. al, 2013 [1])

Figure 5: % of Slowdown of the transfer process of intermediate data between phases (all-to-all) due to contention by CC (taken from *Ananthanarayanan*, et. al, 2013 [1]). The slowdown caused by contention in CC with the first two bins representing smaller jobs, exhibiting 32% and 39% increase in median transfer time compared to a straggler-less execution.

Seeing both CC and CAC approaches have downsides, the solution presented in *Ananthanarayanan*, et. al, 2013 [1] is a hybrid, delay assignment which waits ω, a time delay, before assigning the early upstream clones in order to find an exclusive copy of the intermediate data as in CAC, but afterwards proceeds to assign the earliest upstream clone to be read as in CC. This approach allows normal variation in finishing time of upstream clones which are not considered straggling. Should the downstream clone wait ω time and not find an exclusive copy, it will read with contention from a finished upstream clone.

The value of ω is calculated each hour for each job bin, such that it minimizes the expected task duration with regards to the balance between avoiding contention and benefiting from the fact that multiple upstream clone output exists. Note that this ω waiting time is not speculative

as in the previous straggler mitigation strategy but it was chosen specifically to shorten task durations as dictated by the data. [1]

Two additional aspects of the Dolly strategy are Cloning Budget (β) and Admission Control. The cloning budget is a configurable fraction of the total capacity of the cluster that shall not be exceeded at any time. This ensures that utilization is not exceeded and the overall performance of the cluster is not threatened by the implementation of Dolly, while also using the available resources more efficiently and decreasing the number of idle workers. [1]

The cloning budget is best used to clone small jobs as defined earlier, as for this kind of jobs we saw the greatest effect using this approach. For each job that enters the admission control a cloning factor n is computed in order to achieve the acceptable probability of the job straggling, Ɛ. If at that time the resources needed to clone the tasks n times are available in the cloning budget, the cloning is performed. Otherwise, the cloning is denied, and the job is executed as usual. This admission control favors smaller jobs, as often the resource requirements for cloning larger jobs are not available in the cloning budget. [1] This is another aspect of the Dolly approach that emphasizes its use for smaller jobs, and less for larger ones.

5. **Prototype**

Pseudocode for Approach 1: Speculative-Resume (an extension of speculative execution)

Input: Values for T_{test}, T_{kill}, D (deadline of job completion), r (# of speculative relaunched tasks). We can also set these to default values. Initiated tasks.

Output: results

```
1  start time counter ← null      // initialize time counter once tasks are launched
2  if time counter == T_test:
3      for each running task, do:
4          RT ← 0                 // initialize remaining time
5          calculate an estimated remaining time RT
```

```
6               If RT > D (deadline time) then:
7                       kill original task
8                       launch r+1 speculative tasks from the last processed byte +=1
9               end if
10      end for
11  end if
12  if time counter == T_kill:
13      for each task ∈ speculative tasks do:
14              RT ← 0          // initialize remaining time
15              calculate an estimated remaining time RT
16      end for
17      Sort speculative tasks by RT in ASC
18      fastest attempt ←task whose RT is smallest
19      kill remaining attempts != fastest attempt
20      return fastest attempt results after completion
```

The pseudocode can generally be divided into four steps.

1) At time T_{test} all running tasks are traversed and their remaining time until completion is estimated. If any of their remaining time is > D (deadline of job completion set by the user), these tasks are marked as stragglers. Other tasks continue.
2) Straggler tasks are killed (but their preprocessed data is saved). r+1 tasks are launched for the straggler task starting from the byte after the last byte processed

3) At time T_{kill}, progress scores of all attempts are calculated and only the attempt with the smallest estimated remaining completion time is left running, while the other r attempts are killed.
4) Remaining task is run until completion.

Pseudocode for Approach 2: Dolly (an extension of cloning)

Input: Values for n (# of tasks), p (cluster-wide probability of a straggler), Ɛ (acceptable risk of a job straggling), C (cluster capacity), U, (cluster utilization), β (Budget in fraction), B_U (utilized budget in # slots).

The aim is to have the value of Ɛ < 5% and p is calculated every hour.

Output: results

```
1   c = ⌈log  1 − (1 − Ɛ) (1/n) - / log p          // calculates the desired number of clones per task
2   If (B_U + c · n) ≤ (C · β) and (U + c · n) ≤ τ:   // check if budget and utilization are not exceeded
3       Admission Control: Sufficient capacity to create c clones for each task
4       for each task t do:
5           Create c clones for t
6       BU ← BU + c · n
7   end if
8   for each running task, do:
9       if downstream:                  // check if the task needs to read intermediate outputs
10          wait  ω
11          for each upstream_clone ∈ finished  upstream clones, do:
12              if upstream_clone.readers == 0:      // Try to find an exclusive clone to read
13                  read upstream_clone.outputs
```

```
14                          upstream_clone.readers += 1 // Notify this clone is being read
15                  end if
16                  else:
17                          min_readers_clone = find upstream_clone with least readers
18                          read min_readers_clone.outputs
19                          min_readers_clone.readers += 1
20                  end else
21              end for
22      end if
23  end for
24  return fastest attempt results after completion
```

The pseudocode can generally be divided into four steps.

1) Calculating cloning factor c and checking if admission control terms are met; resources are available in the cloning budget (β) and cloning the job will not exceed utilization limit (τ).
2) Tasks are cloned and upstream tasks are executed.
3) Downstream tasks wait the time delay (ω) after the earliest upstream task had finished, and then they look for an exclusive upstream clone to read from. If not found, they read with contention. In both cases they update the other nodes from which upstream clone they are reading.
4) Process continues until completion.

6. Future Work

Future work would involve creating a software library in order to implement the proposed strategies in the cloud provider: those compatible with Lithops are AWS, IBM Cloud, Microsoft Azure, Google Cloud, Alibaba Cloud, Kubernetes, and OpenShift. [7] Once implemented, the user

could execute their code through serverless systems using Lithops as usual, since the straggler mitigation strategies would already be incorporated on the backend with the cloud provider. In this way, serverless systems like Lithops would be able to continue with their overarching purpose of reducing management on the user while still making straggler resilient algorithms behind the scenes.

7. Conclusions

In conclusion, we have seen that the run time of operations performed by serverless systems stands to be greatly improved using these novel approaches for straggler mitigation. Examining both the challenges and benefits associated with traditional methods such as Speculative Execution and Blacklisting, the two approaches discussed in this paper (task-cloning and speculative-resume) propose innovative solutions that build upon tried-and-true techniques and are shown to work even more effectively. As previously mentioned, these methods were developed with small jobs in mind, as they are most likely to be affected by stragglers, and account for the majority of the jobs run by serverless systems. Future approaches will no doubt utilize those mentioned here, perhaps in combination, or when dealing with different sub-categories of small jobs, in the effort to do away with tasks that just always seem to be left behind in the dust.

8. Bibliography

1. Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, Ion Stoica. 2013. Effective Straggler Mitigation: Attack of the Clones. University of California, Berkeley, KTH/Sweden. From https://www1.icsi.berkeley.edu/pubs/networking/ICSI_effectivestragglermitigation13.pdf

2. Xiaohan Huang, Chunlin Li, and Youlong Luo. 2019. Optimized Speculative Execution Strategy for Different Workload Levels in Heterogeneous Spark Cluster. In Proceedings of the 2019 4th International Conference on Big Data and Computing (ICBDC 2019). Association for Computing Machinery, New York, NY, USA, 6–10. DOI:https://doi.org/10.1145/3335484.3335493

3. Mehmet Fatih Aktaş and Emina Soljanin. 2019. Straggler Mitigation at Scale. IEEE/ACM Trans. Netw. 27, 6 (Dec. 2019), 2266–2279. DOI:https://doi-org.ezprimo1.idc.ac.il/10.1109/TNET.2019.2946464

4. Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the cloud: distributed computing for the 99%. In Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17). Association for Computing Machinery, New York, NY, USA, 445–451. DOI:https://doi-org.ezprimo1.idc.ac.il/10.1145/3127479.3128601

5. Sampe, J. (2018, April 24). Process large data sets at massive scale with PyWren over IBM cloud functions. IBM. Retrieved February 8, 2022, from https://www.ibm.com/cloud/blog/process-large-data-sets-massive-scale-pywren-ibm-cloud-functions

6. Maotong Xu, Sultan Alamro, Tian Lan, and Suresh Subramaniam. 2017. Optimizing Speculative Execution of Deadline-Sensitive Jobs in Cloud. SIGMETRICS Perform. Eval. Rev. 45, 1 (June 2017), 17–18. DOI:https://doi.org/10.1145/3143314.3078541

7. What is Lithops? (n.d.). Retrieved February 16, 2022, from https://lithops-cloud.github.io/docs/

8. Kumar, Umesh & Kumar, Jitendar. (2014). A Comprehensive Review of Straggler Handling Algorithms for MapReduce Framework. International Journal of Grid and Distributed Computing. 7. 139-148. 10.14257/ijgdc.2014.7.4.13.

9. Ajay Gupta CORE ·, Gupta, A., 20, N., & Like (3) Comment . (2020, November 20). Identify and resolve stragglers in your spark application - dzone big data. dzone.com. Retrieved February 4, 2022, from https://dzone.com/articles/identify-and-resolve-stragglers-in-your-spark appl#:~:text=Skewed%20partitioning%3A%20This%20is%20one,to%20each%20of%20the%20partition).&text=Once%20the%20computation%20gets%20skewed,could%20potentially%20result%20in%20stragglers

10. Danish Khan, Kshiteej Mahajan, Rahul Godha, & Yuvraj Patel. (2015, December 19). Empirical Study of Stragglers in Spark SQL and Spark Streaming. From https://pages.cs.wisc.edu/~dkhan/sparkstraglers.pdf

11. Speculative execution in Hadoop MapReduce. DataFlair. (2018, November 14). Retrieved February 7, 2022, from https://data-flair.training/blogs/speculative-execution-in-hadoop-mapreduce/#:~:text=In%20Hadoop%2C%20MapReduce%20breaks%20jobs,thus%20reduces%20overall%20execution%20time.&text=Hadoop%20doesn't%20try%20to,called%20speculative%20execution%20in%20Hadoop.

12. Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the cloud: distributed computing for the 99%. In Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17). Association for Computing Machinery, New York, NY, USA, 445–451. DOI:https://doi-org.ezprimo1.idc.ac.il/10.1145/3127479.3128601

13. Sampe, J. (2018, April 24). Process large data sets at massive scale with PyWren over IBM cloud functions. IBM. Retrieved February 8, 2022, from https://www.ibm.com/cloud/blog/process-large-data-sets-massive-scale-pywren-ibm-cloud-functions

14. Can Karakus, Yifan Sun, Suhas Diggavi, and Wotao Yin. 2019. Redundancy techniques for straggler mitigation in distributed optimization and learning. J. Mach. Learn. Res. 20, 1 (January 2019), 2619–2665.

15. P. Garraghan, X. Ouyang, R. Yang, D. McKee and J. Xu, Straggler Root-Cause and Impact Analysis for Massive-scale Virtualized Cloud Datacenters. In *IEEE Transactions on Services Computing*, vol. 12, no. 1, pp. 91-104, 1 Jan.-Feb. 2019, doi: 10.1109/TSC.2016.2611578.