# PRISM: An Experiment Framework for Straggler Analytics within Containerized Clusters

Dominic Lindsay
School of Computing & Communications
Lancaster University, UK
d.lindsay4@lancaster.ac.uk

Sukhpal Singh Gill
School of Computing & Communications
Lancaster University, UK
s.s.gill@lancaster.ac.uk

Peter Garraghan
School of Computing & Communications
Lancaster University, UK
p.garraghan@lanacster.ac.uk

## ABSTRACT

Containerized clusters of machines at scale are encountering substantive difficulties with stragglers – whereby a small subset of task execution negatively degrades system performance. Stragglers are an unsolved challenge due to a wide variety of root-causes and stochastic behavior. While there have been efforts to mitigate their effects, few works have attempted to empirically ascertain how system operational scenarios precisely influence straggler occurrence and severity. This challenge is further compounded with the difficulties of conducting experiments within real-world containerized clusters. System maintenance and experiment design is often an error-prone and time-consuming process, and a large portion of tools created for workload submission and straggler injection are bespoke to specific clusters, limiting experiment replicability. In this paper we propose PRISM, a framework that automates containerized cluster setup, experiment design, and experiment execution. Our framework is capable of deployment, configuration, execution and performance trace transformation of containerized application frameworks, enabling scripted execution of diverse workloads and cluster configurations. Our framework results in the time required for cluster setup and experiment design to be reduced from hours to minutes. We use PRISM to conduct an analysis of system operational conditions, and identify straggler manifestation occur is a affected by resource contention, input data size and scheduler architecture limitations.

## CCS CONCEPTS

• Cloud Computing • Scheduling • Distributed Architectures

## KEYWORDS

Containers, Stragglers, Clusters

**ACM Reference format:**

Dominic Lindsay, Sukhpal Singh Gill, and Peter Garraghan. 2019. PRISM: An Experiment Framework for Straggler Analytics within Containerized Cluster Environments.

## 1 INTRODUCTION

Large scale containerized clusters have driven development of Cloud technologies, required for execution of Big Data applications such as social media, e-commerce and data analytics. The velocity and volume of data generated require schedulers that can execute application workflows on highly distributed heterogeneous shared computing resources. Application isolation and resource abstraction are critical for shared cluster provisioning. Containers runtimes such as Linux Containers, Docker and OpenVZ have emerged as lightweight performant alternatives to virtual machines [1]. Due to the increased scale and inherent complexity of such containerized clusters in production, such systems are frequently exposed to emerging phenomena directly influencing system performance. One such phenomena is the Long Tail problem, whereby a small number of task stragglers negatively impact job completion time. It has been demonstrated that such stragglers are an unsolved challenge in production containerized clusters operated by Google [2], and Alibaba [3].

Whilst there have been considerable efforts to address the challenge of stragglers pertaining to their detection and mitigation [4-7], ascertaining the causes of straggler manifestation is challenging. This is because stragglers may occur from a wide variety of sources spanning resource contention, data skew, daemon processes, energy management, failure [2], or a combination of each. Stragglers are highly transient and stochastic in nature, making it difficult to replicate system conditions leading to their occurrence. Whilst we have begun to empirically study and understand straggler root causes [3][8], it is unknown to what degree system conditions directly influence their manifestation.

An effective means to address this problem is via conducting comprehensive experiments in real-world containerized clusters under various controlled operational scenarios in a laboratory setting. Conducting such experiments allows for empirical study of realistic system operation, as well as proposition of new approaches without interfering with production system behavior, as well as underpin parameterization of Cloud simulation frameworks [9].

Designing and performing experiments in real-world containerized clusters is a tedious, error-prone, and time-consuming process. We have identified three problems: **1)** setting up and integrating container environments such as Kubernetes[24] and Mesos with various data processing frameworks and scheduling architectures, **2)** manual design and implementation of workload submission and straggler injection tools that are bespoke in nature, often publicly unavailable, and only applicable to a specific cluster and experiment configuration, and **3)** researchers must collect and clean heterogeneous datasets extracted from various cluster components in order to conduct their analysis.

With these issues combined, a large portion of a researcher's time is dedicated to cluster maintenance and experiment design as opposed to exploring an experiment problem space. This results in challenges associated with experiment replicability, and limited comparison against state-of-the-art approaches (i.e. sufficient time to only compare a single scheduler framework as opposed to multiple). These issues are not solely limited to straggler research and encompass a large body of systems research for Cloud datacenters, Fog Computing, and IoT.

Similar to how Apache MapReduce simplified the complexities of deploying data processing on networked machines [10], we see a similar opportunity for experiment design in clusters. In this paper, we propose PRISM, a framework that enables automated cluster setup and experiment design for containerized clusters to study straggler manifestation. Our framework automates configuration of scheduling platforms, as well as translation, collection and execution of performance traces. Using our framework, it is possible to submit workload onto a cluster within various operational scenarios controlling cluster operation to ascertain the relationship between system conditions and straggler manifestation. Our contributions are two-fold:

− *Automated cluster experiment framework;* capable of interfacing with a wide variety of scheduler and workload types, and simplifying a large portion of cluster deployment, configuration, experiment execution, and data collection. Furthermore, the framework enables sharing of containers encapsulating cluster configurations and trace transformation.

− *Straggler analytics;* we demonstrate how the framework supports studying straggler manifestation under various controlled system conditions. Our preliminary findings show stragglers may manifests as a result of CPU contention and data size. Furthermore, we find a schedulers logical model of a cluster, can impact straggler manifestation.

The paper is structured as follows: Section 2 provides the research background; Section 3 discusses the related work; Section 4 presents the proposed PRISM framework; Section 5 presents a straggler analysis case study; Section 6 presents the conclusions.

## 2 BACKGROUND

**Containerized clusters**: Containers provide virtualized environments encapsulating applications and their configurations [1]. Similar to virtual machines (VMs), containers allow several application environments to share a single host machine. Implemented as a kernel feature, containers do not require hypervisor hardware emulation and instead achieve resource isolation via resource multiplexing of kernel resources [11]. Containers provide several advantages over hypervisor-based virtualization, including; smaller image size [12], rapid boot time, and greater resource efficiency [1] allowing for rapid scaling. Hence, providers such Google Cloud [2] and AliYun[3] are increasingly leveraging containers in their physical computing infrastructure to form *containerized clusters* for Cloud services. Scheduling platforms such as Apache Yarn use containers as their primary unit of execution and isolation. However, increased adoption and scale of containerized clusters, such systems frequently exposed to straggler manifestation.

**Stragglers**: A s*traggler* can be defined as abnormally slow task execution within a job [13]. It has been established that stragglers are particularly problematic towards ensuring predictable job execution within production systems due to volatile network conditions, resource dynamicity, and scheduling architecture [2]. It has been demonstrated that approximately 5% of tasks stragglers can negatively impact the performance of almost 50% of total jobs within containerized clusters [3]. If a straggler task prevents other dependent tasks from successfully completing, the job is unable to complete until straggler task completion, increasing job completion times, reducing system availability, and incurring addition resource overheads. Due to the wide variety of causes of stragglers, their transient behavior, and non-deterministic manifestation, it is considerably challenging to determine what system conditions influence their occurrence within production and laboratory conditions alike. Such work is key to design appropriate mechanisms to mitigate their effects.

**Cluster Experiment Frameworks**: For many researchers, studying straggler manifestation within containerized clusters is particularly challenging due to the complexities of cluster setup and experiment design. Even if a research group has access to a sufficiently large cluster, configuring and deployment of data processing framework requires considerable domain knowledge on management and monitoring cluster operation. This causes cluster setup and maintenance to be an error-prone and time-consuming process; an activity typically performed by a group of dedicated developers instead is reliant on a smaller research group. Designing experiments for clusters also encounters similar issues, whereby tools to control system operation, workload submission patterns, and straggler injection are bespoke to a specific cluster setup, and not generalizable to other containerized clusters. This is an issue given that such tools are frequently not made publicly available, reducing experiment replicability. This also imposes additional limitations on researchers, reducing the number of approaches that can be feasibly compared for evaluation. These issues have resulted in a large body of research relying on small-scale experiments [15] or simulated environments [14][16]. Although such approaches are appropriate for designing new straggler mitigation techniques, empirical analyses and experimentation are required in order to understand straggler manifestation within real-world systems, as well as capture non-deterministic system behavior.

## 3 RELATED WORK

Related work is categorized into two research domains: (1) straggler analysis, and (2) straggler evaluation frameworks.

**Straggler Analysis**: Eman et al. [17] identify a potential cause of stragglers stemming from data dependencies amongst parallel processes further complicated by differing task data priorities. The authors proposed a load balancing and partitioning technique to alleviate task slowdown and enhance job performance. Garraghan et al. [3] empirically analyzed straggler manifestation and their root-causes within two production Cloud datacenters using containers, discovering that approximately 5% of stragglers negatively impacted the performance of 50% of all jobs. Furthermore, they identified the most frequent cause of stragglers were due to resource contention (*CPU, disk, memory, and network*). Ouyang et al. [8] studied the impact of straggler manifestation from node failures contention, and observe high resource contention as an underlying cause of stragglers. Ganesh et al. [4] studied straggler manifestation within latency sensitive jobs, and demonstrated that job cloning as an effective means to minimize their impact. Farshid [6] identified that Mapper task duration increases as clusters scale, and designed an analytical model comprising application and hardware characteristics to capture this. Ganesh et al. [18] created various resource-aware techniques for straggler mitigation and identified the several causes of straggler manifestation from varying bandwidth, network congestion, workload imbalance and contention of resources.

Research findings define assumptions of straggler manifestation in containerized clusters, and are used to create straggler mitigation techniques that focus on different aspects of latency [4][19], network congestion [20], and energy [20]. However, no current work has attempted a comprehensively study how precise system operational conditions influence straggler manifestation.

**Straggler Frameworks**: There exist several straggler evaluation frameworks: Bux et al. [14] proposed DynamicCloudSim to simulate cluster execution by configuring different models for failure, resource contention, and straggler manifestation. Straggler behavior is configured with default values from prior work [7], and is used to simulate various existing mitigation strategies to improve cluster performance. Yanfei et al. [15] proposed a user-transparent task slot management framework *FlexSlot*, which identifies the stragglers and automatically resizes the number of virtual node slots to improve the speed of execution of tasks. The framework was evaluated within an 8-node Hadoop cluster, whereby they injected stragglers to alleviate job data skew. Tien-Dat et al. [16] proposed a framework for straggler detection and mitigation to enhance job execution time and system energy-efficiency. Using the Grid'5000 testbed consisting of 21-nodes, authors artificially injected stragglers into job application execution and evaluated the framework with straggler mitigation techniques.

Whilst these frameworks have evaluated various straggler mitigation strategies, most rely on simulation or small-scale clusters experiments for evaluation. Each framework is dependent on artificial straggler injection introduced by the developer and are not designed to explore natural system operation that may cause stragglers. Importantly, frameworks rely on manual design of experiment design from a domain expert to conduct experiments.

## 4 PRISM FRAMEWORK

PRISM framework enables automated deployment, execution, and performance collection of containerized cluster operation. Researchers capture stages of the experimental lifecycle as containers encapsulating components and configuration enabling deployment and sharing of bespoke scheduling systems as well as trace parsing and transformation. In doing so, configuration and algorithms can be deployed as modules. Furthermore, modules can be shared, reducing complexity associated with reproduction of experimental clusters and trace execution. PRISM also allows for injection of resource (*CPU, disk, memory and network*) utilization enabling cluster preloading. This allows researchers to submit identical workload patterns into a containerized cluster using different resource management frameworks (YARN, Kubernetes) under various levels of contention to study changes in cluster performance. Moreover, the system automatically extracts data parameters of interest spanning both software and hardware components into a data repository for ready analysis.

Several interfaces are defined for submission, execution and data collection. Figure 1 shows the system model, which describes the interaction of various software and hardware components of the cluster to study the performance while running the workloads. The framework is formed by three main components: *Experiment Runner*, *Cluster Manager*, and *Results Repository*.

**Experiment Runner**: Responsible for monitoring, executing, and controlling experiment conditions as well as collecting system parameters. The experiment runner is designed so that it can readily implement different scheduling platforms, workload patterns, and system operational scenarios. Different schedulers are integrated into the module via implementation of abstract interfaces responsible for mediating between the PRISM framework and the target scheduler. A variety of workload types and submission patterns are configurable by parsing and conversion of job traces, including specifying the number of jobs, application type, and data input. The module is also designed so that it can use real-world trace data to inform its submission patterns. The module is capable of controlling cluster operational scenarios, specifically resource contention (demonstrated to be a primary cause for failure [22] and straggler manifestation [2]). Achieved by co-deployment of utilization containers, designed to exert varying levels of load (10%, 20% .... 100%) on specified resources of a worker node.

**Cluster Manager**: Provides an abstraction to start, stop, and query a scheduler platform used for an experiment. The user is able to interact with a specified scheduler via a web interface for the scheduling control plane, used to administrate the scheduler platform. Because the scheduler control planes have different interfaces, ranging from *IPC* clients, to *REST* interfaces[23][24], our approach provides a cluster management interface. Users of the PRISM framework must for implement scheduler specific interface

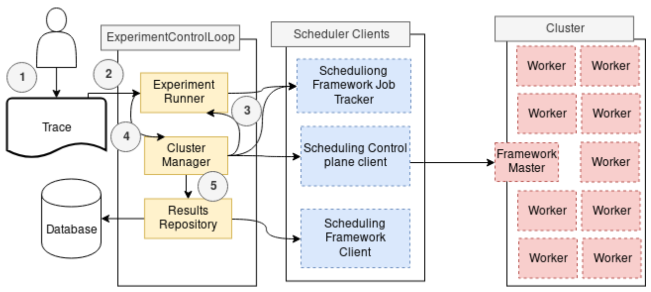


**Figure 1: PRISM Conceptual Modal**

(*IClusterInterface)* responsible for mediating between the scheduling control plane, job tracking and *ClusterManager* components.

**Results Repository**: Responsible for collecting results of jobs and parsing the traces from a scheduler framework specific format to a user defined format, before writing to persistent storage. Users implement the *IOutputWriter* interface responsible for encoding how job performance traces are parsed and transformed from their target scheduling framework trace format to a bespoke output format. Finally, traces can be pushed to a target database, or output as csv format for persistent storage.

### 4.1 Experiment Lifecycle

The PRISM framework abstracts the lifecycle of configuration, execution and collection of scheduling experiments. Identification of optimal scheduling configurations is achieved by comparison of results from test traces. The first stage of experimentation is often concerned with configuration of the cluster to enable or configure a feature of the scheduler. Comparing scheduler configurations of the same platform is relatively simple, and trace can be simply cast into job description. More involved is the process of converting traces of a different scheduling framework and application. Traces must be prepared and transformed into job descriptions that can be executed at the target scheduling framework. Furthermore, performance traces are application specific and as such require transformation into a common format for comparison. The workflow for PRISM (Figure 1) operates is follows:

1. Client initiates the experimental run by passing a path to a directory containing the PRISM framework configuration.
2. *ExperimentControlLoop* starts the experiment by calling the *ExperimentRunner* component to execute jobs traces found in the configuration manifest.
3. *ClusterManager* periodically interfaces with the scheduling control plane and *ExperimentRunner* to identify experiment progress and framework status.
4. Upon completion of experiment traces, the *ClusterManager* invokes the *ResultRepository.*
5. *ResultsRepository* collects and stores performance and telemetry metrics from the scheduling platform before transforming traces into a generic user defined format and exposing to persistent storage.

## 5 FRAMEWORK CASE STUDY

### 5.1 Experiment Setup

In order to demonstrate the applicability of our proposed framework, we have deployed PRISM into a containerized cluster to study straggler manifestation under various system conditions. We deployed the framework onto a medium-sized cluster containing 38 nodes (4 x i7-4770 cores, 8GB RAM, 256GB SSD). Our experiment setup deployed a PaaS infrastructure using Kubernetes1.15. Apache Yarn capacity resource scheduler [23] was automatically deployed into the Kubernetes [24] cluster as isolated namespaces. We used Hadoop 2.9.2 to create a suitable data processing network and persistent Hadoop Distributed File System (HDFS). Both HDFS and Yarn were configured with a single master node with 37 worker nodes, and managed by the Kubernetes StatefulSets. This entire process is automated by the PRISM framework by manipulating configuration manifests (Figure 4), hence it is relatively trivial to deploy an alternative scheduler framework, data processing framework, or application type by simply changing manifest files.

Our experiment case study provides a preliminary investigation into straggler manifestation under varying cluster operational conditions. As previous studies of production systems have indicated a relationship between cluster resource contention and straggler occurrence [2][3], thus we expose jobs to various controlled system conditions. Multiple experiment runs were performed, each configuring different application data input size (20GB, 40GB) and resource contention per node (0%, 20%, 50%, 80%) to provide sufficient coverage of high and low system usage. Each experiment run consists of submitting 100 jobs into the cluster via the Yarn scheduler, each job executing WordCount benchmark containing 150-320 tasks with and without speculative execution [19] (i.e. replicas automatically launched from detected slowdown). The design of PRISM allows all experiment run configurations and system conditions to be controlled via configuration files, parsed and executed by the framework. When combined together, our experiment design consists of 16 unique experiment runs, 4,800 unique job submitted, and totaling 9 days of cluster execution.

### 5.2 Analysis

**Job Execution**: Table 1 shows the statistical properties of job execution for each experiment run under various controlled operational conditions. It is observable that increased data input size and cluster resource contention levels results in increased average job completion times (JCT) from 175s to 3609s and 1013s – 6591s between 0% to 80% CPU contention, respectively. Whilst an increase in CPU contention and data input size results in a larger JCT is somewhat intuitive, an observation of interest is the substantial difference in JCT when speculative execution is disabled, reflected by a 3x-4x increase. We believed this is caused by variability in performance interference from tasks co-located on the same node, and caused by lack of speculative monitoring. Task execution is not deterministic, and straggler task latencies are allowed to accumulate, as reflected in JCT standard deviation.

**Stragglers**: We were able to observe straggler manifestation across two experiment runs, as shown in Figure 2. The reason for stragglers not detected in each experiment run is due to their highly transient nature as discussed in Section 2. Hence, it is not guarantee for stragglers occur every experiment run, nor replay deterministic system conditions to replicate their occurrence. This is important given that stragglers become increasingly frequent as system scale increases (i.e. stragglers rarely manifest at small-scale, seldom in medium-scale clusters, and frequently in 1000+ node clusters). Specifically, 43 - 220 tasks were detected as stragglers within experiment runs. We observe that stragglers appear to be contained with a small subset of jobs between 4-7%, echoing prior observations in production systems with similar probability [3][8].

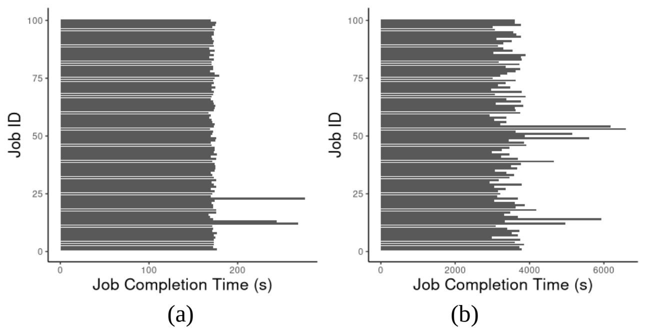
We observe that the deviation in task straggler severity increases at higher levels of CPU contention and data input, as shown in Figure 3 and Table 1. It is observable that at very high levels of contention and data input, task stragglers exhibit a large deviation between their execution in comparison. It is apparent that higher CPU and data input together results in a higher straggler occurrence. The reason for their occurrence is inconsistency between the schedulers logical state and the physical resource availability in the cluster. More specifically the schedulers view assumes exclusive access to resource at the worker machine, however production clusters rely on multitenancy to achieve higher throughput and utilization. This pattern is exemplified when observing an increase in both JCT and deviation per job and per task as shown in Table 1 and Figure 3. High contention levels result in greater deviation significantly reducing a subset of tasks execution latency.

We also observed stragglers occurring with 0% CPU contention and 20GB data input. The reason for such an occurrence was identified to be result of constrained scheduler execution units (slots in yarn [23]). When all execution units are occupied the scheduler can no longer start any new containers for maps/reduces. As such, the scheduler must wait application frameworks to release resources, before allocating resource to waiting jobs, impacting job latency. The point to emphasize again is that stragglers are non-deterministic, hence it is not a given that stragglers only occur at high contention.

**Platform Usage**: As discussed in Section 5.1, configuration variability was encapsulated within Docker images, and experiment design configurations was achieved by manipulation of a single line of configuration used initializing the PRISM framework. Scheduler traces are formatted using scheduler specific structures, whereas scheduler application framework clients submit jobs as manifests and/or via command line clients. A scheduler *ExperimentRunner::ItraceParser::MapReduceTracermodule* was implemented to parse yarn output traces into intermediary job description format. *ExerimentRunner::Iclient::WordcountRunner* which executes equivalents jobs at the target framework. Reproduction of performance traces is greatly simplified, as rather than estimating job parameter configuration on a case by case basis, we were able to develop a translation algorithm capable of creating new jobs, whilst maintaining characteristics from performance traces. Doing so reduced complexity and time associated with reproducing experiment results, furthermore we created several reusable modules which can be distributed alongside PRISM. As an example, traditional approaches we found to manually configure all associated experiment design components, taking on average

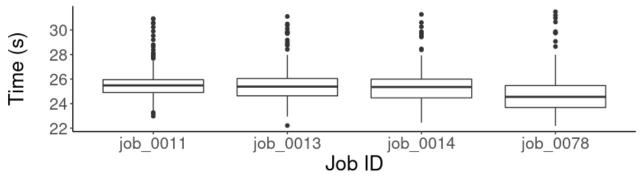
**Table 1 Job Execution Statistical Properties**

| Speculation | Data Input (GB) | CPU contention | JCT $\mu$ (s) | JCT $\sigma$ (s) |
|---|---|---|---|---|
| Enabled | 20 | 0 | 175.1 | 15.8 |
| | | 20 | 376.4 | 23 |
| | | 50 | 573.3 | 37.4 |
| | | 80 | 1608.8 | 236.8 |
| | 40 | 0 | 720.6 | 30.4 |
| | | 20 | 793.9 | 31.7 |
| | | 50 | 1196.6 | 49.4 |
| | | 80 | 3609.6 | 637.5 |
| Disabled | 20 | 0 | 1013.9 | 479.7 |
| | | 20 | 1188.107 | 568.5 |
| | | 50 | 1595.9 | 761.9 |
| | | 80 | 3119 | 1484 |
| | 40 | 0 | 2161.2 | 1030.1 |
| | | 20 | 2535.3 | 1206.4 |
| | | 50 | 3372.3 | 1601.9 |
| | | 80 | 6561.8 | 3109.3 |

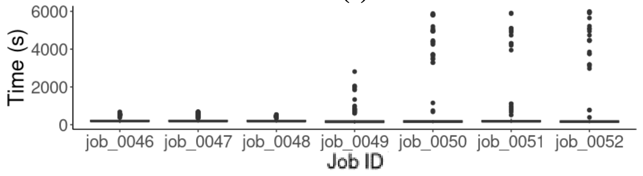


(a) (b)

**Figure 2: Straggler manifestation for job during (a) 0% CPU, 20GB, and (b) 80% CPU, 40GB.**



(a)



(b)

**Figure 3: Task execution distribution for job stragglers during (a) 0% CPU, 20GB, and (b) 80% CPU, 40GB.**

```
PRISM:  v0.1
            experiment_ID:straggler_20gb_0_util
experiment_input:
            trace_path: "test_trace.csv"
            trace_parser: yarn_json_parser
Result_spec:
            output_writer_module: csv
            Output_writer_out_args:
                        Path: straggler_20gb_0_util
Cluster_spec:
            framewokr: yarn
            Size: 38
            Master:
                        Image: yarn:master
            Workers:
                        Image: yarn:worker
            InterferanceInjector:
                        Image: resource_isolation:CPU
---Command Line--
"prismUser$ PRISM deploy straggler_20gb_0_util.yaml"
```

**Figure 4: Example PRISM experiment configuration file**

hours, and is exasperated by cluster misconfiguration leading to wasted experiment execution time. Contrasted with PRISM, configuration and algorithms for transformation of traces are encapsulated by containers and abstracted by interfaces, reducing time taken to reconfigure the cluster and tweak traces to minutes. Experiment traces and resource metrics must be collected and parsed into an intermediate storage format. *ResultsRepository* was responsible for collecting and integrating performance data. In its current state, PRISM only collects job performance statistics and node CPU utilization. Future development of PRISM will integrate telemetry and log data related to hardware operation, cooling system, power usage, and environmental data. This is relatively straightforward given (the intermediate interface transformation architecture as discussed in Section 4.

## 6 CONCLUSIONS

In this paper we have proposed the PRISM framework for automated containerized cluster setup, as well as experiment configuration and design to study straggler manifestation. We have discussed challenges associated with analyzing stragglers, as well conducting experiments in clusters. We leverage the framework to analyze straggler manifestation within real-world containerized clusters, and demonstrate we are able to simplify experiment design and controlling system conditions. Our analysis identifies that speculative execution impacts job completion time by as much as 300% - 400%, as well as reduce overall task latency variance. We find stragglers appear to be temporally related, and that their manifestation is influenced by resource contention within scheduler architectures. As such we have identified a need for dynamicity of slot based schedulers, capable of observing dominant workload characteristics and trends accounting for contention caused by machine resource constraints.

Future work includes extending the PRISM framework in order to capture a wider variety of scheduler architectures, workload types, and complex submission patterns. Furthermore, we aim to extend the framework to interface with telemetry services, as well as integration into Kubernetes. Moreover, we aim to make the platform publicly available to allow researchers to rapidly deploy containerized cluster environments and design experiments.

## ACKNOWLEDGMENTS

This work is supported by the Engineering and Physical Sciences Research Council (EPSRC) (EP/P031617/1).

## REFERENCES

[1] Q. Zhang, L. Liu, C. Pu, Q. Dou, L. Wu, W. Zhou, "A Comparative Study of Containers and Virtual Machines in Big Data Environment," IEEE International Conference on Cloud Computing (CLOUD), pp. 178–185, 2018.

[2] J. Dean, L A. Barroso,"The tail at scale", ACM Communications 56, pp. 74–80, 2013.

[3] P. Garraghan, X. Ouyang, R. Yang, D. McKee, J. Xu, "Straggler Root-Cause and Impact Analysis for Massive-scale Virtualized Cloud Datacenters," IEEE Transactions on Services Computing, 2016.

[4] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. "Effective Straggler Mitigation: Attack of the Clones." In NSDI, vol. 13, pp. 185-198. 2013.

[5] E. Coppa, I. Finocchi. "On data skewness, stragglers, and MapReduce progress indicators." ACM Symposium on Cloud Computing, pp. 139-152. 2015.

[6] F. Farshid. "Stochastic modeling and optimization of stragglers in MapReduce framework," Thesis, The Pennsylvania State University, 2015.

[7] M. Zaharia, et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing." USENIX NSDI, 2012.

[8] X. Ouyang, P. Garraghan, B. Primas, D. McKee, P. Townend, J. Xu, "Adaptive Speculation for Efficient Internetware Application Execution in Clouds, ACM Transactions on Internet Technology, 2018.

[9] R. N. Calherios, R. Ranjan, A. Beloglazov, C. De Rose, R. Buyya, "CloudSim: A Toolkit for Modeling and Simulation of Cloud Computing Environemtns and Evaluation of Resource Provisioning Algorithms", Software: Practise and Experience, pp. 23-50, 2011.

[10] J. Dean, S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," AC Communications pp. 107–113, 2013.

[11] M. A. Rodriguez and R. Buyya, "Container-based Cluster Orchestration Systems : A Taxonomy and Future Directions," pp. 1–19, 2016.

[12] C. Pahl. B. Lee, "Containers and Clusters for Edge Cloud Architectures – A Technology Review." International Conference on Future Internet of Things and Cloud, 2015.

[13] G. Nannicini, "Straggler Mitigation by Delayed Relaunch of Tasks." ACM SIGMETRICS Performance Evaluation Review,pp. 248-248, 2018.

[14] M. Bux, L. Ulf "Dynamiccloudsim: Simulating heterogeneity in computational clouds." Future Generation Computer Systems, pp. 85-99, 2015.

[15] Y. Guo, J. Rao, C. Jiang, X. Zhou. "Moving Hadoop into the cloud with flexible slot management and speculative execution." IEEE Transactions on Parallel & Distributed Systems, pp. 798-812, 2017.

[16] T. Phan, G. Pallez, S. Ibrahim, P. Raghavan. "A New Framework for Evaluating Straggler Detection Mechanisms in MapReduce," ACM Transactions on Modeling and Performance Evaluation of Computing Systems, 2019.

[17] E. B. Khunayn, S. Karunasekera, H. Xie, and K. Ramamohanarao. "Exploiting Data Dependency to Mitigate Stragglers in Distributed Spatial Simulation." ACM SIGSPATIAL, pp. 43-53, 2017.

[18] G. Ananthanarayanan, et al.. "Reining in the Outliers in Map-Reduce Clusters using Mantri." In OSDI, vol. 10, no. 1, p. 24. 2010.

[19] L. Lei, T. Wo, and C. Hu. "CREST: Towards fast speculation of straggler tasks in MapReduce." In IEEE International Conference on e-Business Engineering (ICEBE), pp. 311-316.2011.

[20] W. Da, G. Joshi, and G. Wornell. "Efficient Straggler Replication in Large-scale Parallel Computing." arXiv preprint arXiv:1503.03128 (2015).

[21] S.S. Gill, et al, "Holistic resource management for sustainable and reliable cloud computing: An innovative solution to global challenge." Journal of Systems and Software, Volume 155, 104-129, 2019.

[22] D. Tang, et al., Failure Analysis and Modeling of a VAXcluster System, International Symposium on Fault-Tolerance Computing, 1990.

[23] V. K. Vavilapalli, et al., "Apache hadoop yarn: Yet another resource negotiator," in ACM SoCC, 2013

[24] B. Burns, et al. "Borg, Omega, and Kubernetes", ACM Queue, pp. 70-93, 2016.