PRISM: An Experiment Platform for Straggler Analysis for Containerized Clusters

Dominic Lindsay

School of Computing and Communications  
Lancaster University, UK

d.lindsay4@lancaster.ac.uk

Sukhpal Singh Gill

School of Computing and Communications  
Lancaster University, UK

s.s.gill1@lancaster.ac.ukPeter Garraghan

School of Computing and Communications  
Lancaster University, UK

p.garraghan@lancaster.ac.uk

**ABSTRACT**

Various cloud providers such as Facebook, Google, Amazon and Microsoft are facing challenges related to latency distribution for user-interactive workloads. Such high latency episodes termed the ‘long tail’ increases the execution time of jobs, as well as reduced system availability. As per existing research work, resource contention is the main reason for stragglers, occurring when different jobs are waiting for shared resources. Different applications executing on different nodes may also content for shared global resources. The other reasons for stragglers can be: CPU over-utilization, data abstraction, ineffective resource or job scheduling, data skew and faults. This paper proposes a platform to build an automated testbed for straggler analysis in distributed systems analyses called PRISM to identify the main reasons of high latency and their effects on the system performance. Furthermore, the performance is evaluated to know the relationship between controlled parameters and latency. Moreover, proposed testbed will be used for future evaluations.

**CCS Concepts**

• **Computing methodologies➝ Massively parallel and high-performance simulations.**

**Keywords**

Cloud Computing; Distributed Systems; Straggler.

# INTRODUCTION

Various technology paradigms including e-commerce, social media, and the Internet of Things (IoT) generate large amounts of data transmitted via different types of communication mediums. The velocity and volume of data generation is continuously expanding, requiring substantial computing capacity to process data efficiently. Due to the increased complexity of modern large-scale systems, certain emerging phenomena - which directly affects system performance - can occur. This is known as the Long Tail Problem, or the scenario where a small number of task stragglers, negatively affect the time of the job completion. Task stragglers can occur within any highly parallelized system, which processes jobs consisting of multiple tasks.

Google’s MapReduce framework [1], introduced originally to process huge amounts of in-house data efficiently is one such framework susceptible to stragglers. MapReduce allows for scalability of the system to huge clusters of cheap commodity servers. MapReduce framework splits the large size job into various smaller-sized tasks to improve the completion time of a job by executing the tasks in parallel, utilizing resources from several physical/virtual nodes of the cluster [2]. Once all the tasks composing the job are finished, the job is said to be complete. The parallel execution of tasks increases the speed of execution and handles the failures automatically without human intervention using IBM’s autonomic model [3]. Another example of a similar, big-data processing framework is the Apache Spark [3]. Whilst MapReduce is optimized for linear processing of huge datasets, Apache Spark focuses on a broader approach, attempting to allow faster data processing, linearly or non-linearly, utilizing additional functionality such as Machine Learning or graph processing. Yet another big-data processing framework is Dryad [4]. Although not much different from the previously mentioned frameworks, Dryad allows for higher number of stages than simply “map” and “reduce”.

The occurrence of stragglers is caused by an unnecessary delay caused by one or more the tasks within a job. This delay can occur due to several reasons, including failure [13]. There are two types of failures that can occur during the execution of jobs: task failures and node failures. The task failures occur when a specific task within a job fails, due to diverse sources of software and hardware faults [5]. The node failures occur when one of the resources of a specific node, which executes the job’s task, fails. This can be caused by a myriad of possible OS or hardware level faults. As an example of straggler mitigation techniques, MapReduce attempts to mitigate task failures by relaunching the task once it fails [6]. In terms of a node failure, MapReduce would re-execute all the tasks that were originally scheduled to be executed on that node.

In terms of node failures, when a performance of a node degrades, either due to an OS or hardware fault or the node completely fails, a specific task’s (straggler) execution time can be unnaturally increased, causing any other tasks that depend on it to wait for its completion [7]. At the job level, for the job to be considered complete, the tasks comprising this job must finish. If a straggling task prevents other dependent tasks from successfully completing, the job will not complete until the straggler task completes. Furthermore, additional resources can be consumed due to stragglers, causing further effects on the rest of the system.

In this research paper, we propose a PRISM (XXXXXXXXXXXX), a platform for conducting automated experiments of straggler manifestation within containerized cluster. Our platform allows for XXXXXXXXXXXXX XXXXXXXXXXXXX (what are advantages/how does it work at very very high level?). Using our platform we submit various workload onto a cluster within various operational scenarios controlling cluster resource contention and data skew in order to ascertain the relationship between system conditions and straggler manifestation. The rest of the article is structured as follows: Section 2 presents the related work of straggler management techniques. Section 3 presents the system framework. Section 4 presents the performance evaluation and experimental results. Finally, Section 5 concludes the article and discusses the open challenges and future research directions.

# MOTIVATION

Configuration of scheduling platforms is complex and sub-optimal configuration can lead to performance degradation.

Empirical analyses of scheduling performance has previously enabled better understanding of how scheduling framework designs perform against workloads characteristics, leading to better scheduler designs.

However, due to complexity of scheduling platforms fair performance comparison between configurations and architectures is challenging. Furthermore, interfaces for execution, monitoring and trace collection requires knowledge of different tools across systems. The development of which is often partitioned between distributed working groups of developers, increasingly nuanced knowledge required for reporting and understand problems users may encounter.

Hence, the researcher community makes use of simulated environments [14]. Although such systems are capable of providing proof of concept results, empirical analyses are required in order to understand transient and complex issues which manifest as a result of underlying system infrastructure [13].

A platform capable of abstracting and automating trace parsing, job submission, monitoring and trace collection is needed. Doing so would ensure published framework configurations, can be applied, enabling repetition of experiments, and fair comparison between frameworks without intervention of the original author. We see several advantages to this approach, first the configuration of both scheduling platform and jobs can be communicated to third parties. Second, large numbers of experiments can be executed sequentially with minimal operator interference. Finally, such a framework will aid research independently published scheduler characteristics.

# RELATED WORK

Garraghan et al. [1] empirically analyzed straggler manifestation and root-causes within production Cloud datacenters, where they discovered that approximately 5% of stragglers negatively impacted the perform of 50% of all jobs. Furthermore, they identified that the most frequent cause of stragglers was due to resource contention (CPU, memory, and disk). Ouyang et al. [2] proposed a method to reduce Late-Timing Failure (LTF) and analyze the root-cause of stragglers in cloud datacenters (CDC) such as server failures or task concurrency and resource contention. Further, this study identified the high temporal resource contention as a main root-cause of stragglers. Further, experimental results demonstrate that proposed method maintains the performance of the computing systems while tolerating the system failures effectively. Eman et al. [9] proposed a parallel model for straggler mitigation in distributed spatial simulation called Priority Asynchronous Parallel (PAP) model to exploit data dependencies of parallel processes to be computed and synchronized based on data priority to the other workers. Further, load balancing and partitioning method is proposed to balance the workloads among different nodes and help to improve the performance speedup by a large extent.

Ganesh et al. [3] explored the straggler mitigation techniques and identified the impact of reasons of stragglers in latency sensitive jobs. Further, authors designed workloads with small number of jobs and performed cloning of small jobs. It has been identified that the cloning of small jobs uses less resources but improves the reliability of computing services. In this research work, a system named Dolly is proposed to generate multiple clones of jobs and execute jobs within their specified budget. Experimental results demonstrate that Dolly sped up jobs by 46% by using only 5% extra resources. Farshid [5] analyzed that map phase of MapReduce (MR) framework takes longer with the increase in number of servers, which further affects negatively the execution time of MapReduce job. Moreover, authors designed an analytical model to identify the impact of stragglers on efficiency of computing system using map phase in terms of application, system and hardware parameters. Experimental results show that model reduces the execution time during execution of MapReduce applications. Ganesh et al. [8] studied and explored the straggler management in resource aware techniques and identified the main causes of stragglers such as varying bandwidth, network congestion, workload imbalance and contention of resources (network, memory and processor).

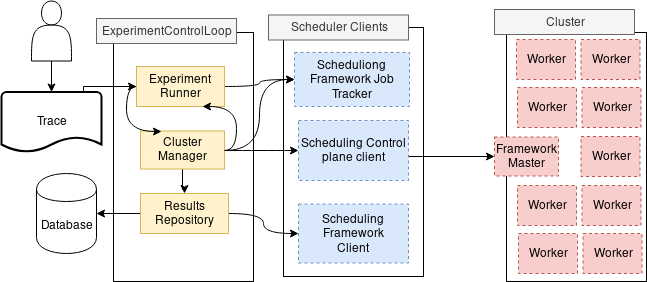
Da et al. [10] analysed the trade-off between latency and cost to find out the best replication technique for straggler management based on following parameters: 1) when to perform replication for straggling tasks, 2) number of replicas to be launched and 3) is it necessary to destroy the original copy or not. Further, a Straggler Management Approach (SMA) is proposed to calculate the value of latency-based empirical distribution of execution time of task. The experimental results demonstrate that proposed approach gives better results in terms of cost and latency. Lei et al. [11] proposed a straggler management technique called CREST (Combination Re-Execution Scheduling Technology) for fast speculation of straggler tasks in MapReduce framework, which further reduces the response time of MapReduce jobs. The re-execution of set of tasks on set of computing nodes in CREST to improve the speed of task execution. Sukhpal et al. [12] have analyzed the performance the effect of various parameters (SLA violation rate, energy consumption, CPU utilization, Memory utilization, Reliability, Latency, Network bandwidth, Fault detection rate, and Intrusion detection rate) on probability of stragglers but they have not identified the correlation between conditions and probability of stragglers. Nawab et al. [13] proposed dynamic container-based resource scheduling framework, that shifts coupled associations of job profiles to dynamically available resource containers. Also, it relieves static container allocations and presumes them as a fresh piece of resource allocation for new job profile. The experimental evaluation shows that the proposed dynamic framework reduces wastage of resource allocations and increase ecosystem performance than default job profile in spark ecosystem.

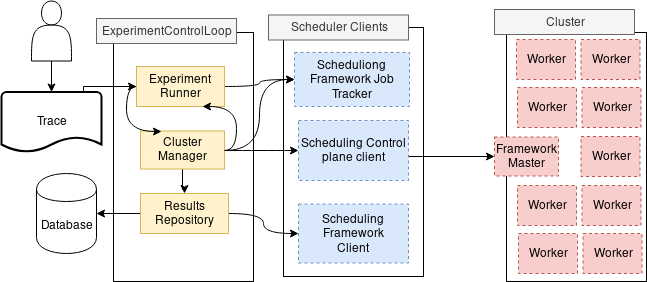
What is the uniqueness of this work/what about experiment platform related work? This is all straggler which is marginally related.

XXXX

1. PRISM FRAMEWORK

The PRISM framework is a platform that enables automated deployment, execution, and analysis of containerized cluster operation. This allows a researcher to readily define various system traces, job types and submission patterns, as well as precisely control cluster experiment conditions. This allows, for example, to submit identical workload patterns into a containerized cluster using different resource management frameworks (YARN, Kubernetes) under various levels of CPU and disk 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.



  
Figure 1: PRISM Conceptual Modal

The platform is formed by three main components: Experiment Runner, Cluster Manager, and Results Repository.

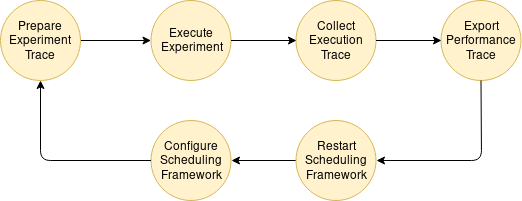
**Experiment Runner**: Responsible for monitoring, executing, and controlling experiment conditions as well as collecting system parameters. The experiment runner is designed so that it can readily implement different scheduling platforms, workload patterns, and system operational scenarios. Different schedulers are integrated into the module by implementation of bespoke interfaces responsible for mediating between the PRISM framework and the scheduling platform. A variety of workload types and submission patterns are configurable via configuration files, and job trace including specifying number of jobs, application type, and data input. The module is also designed so that it can use real-world trace data in order to inform its submission patterns. Moreover, the module is capable of controlling the cluster operational scenarios, specifically resource contention (which has been demonstrated to be a primary cause for failure [REF] and straggler manifestation [REF]. This is performed by automatically creating a daemon process within each cluster node which executes (dummy) execution imposing additional resource utilization. The module is capable of controlling the level of contention (e.g. 10%, 20% … 100%) via co-deployment of container with yarn worker nodes.

**Cluster Manager**: provides abstraction to start, stop, and query a scheduler platform used in for an experiment. The user is able to interact with a specified scheduler via a web interface for scheduling control plane clients used to administrate the scheduler platform. Because scheduler control planes have different interfaces, ranging from *IPC* clients, to *REST* interfaces, our approach provides cluster management interface. Users of the PRISM framework are responsible for implementing scheduler specific interface (*IClusterInterface)* responsible for mediating between the scheduling control plane, job tracking components and the *ClusterManager* component.

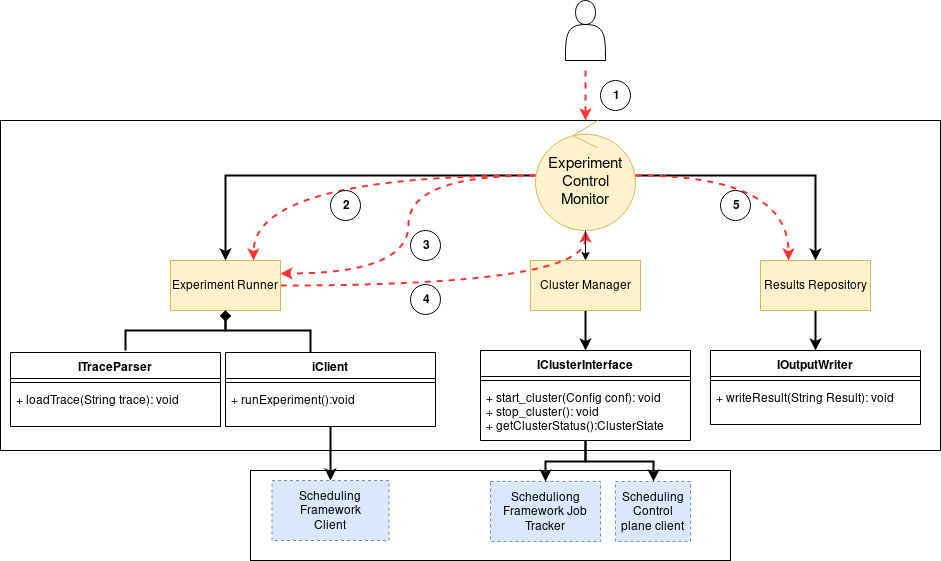
**Results Repository**: is responsible for collecting results of jobs and parsing the traces in scheduler framework specific format to a user defined format, before writing to persistent storage. Users implement the *IoutputWriter* interface responsible encoding how job performance traces are parsed 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.

* 1. Life cycle of Experimental Process

The PRISM framework aims to abstract the lifecycle of configuration, execution and collection of scheduling experiments. Figure X identifies key stages of scheduling framework evaluation to be abstracted.

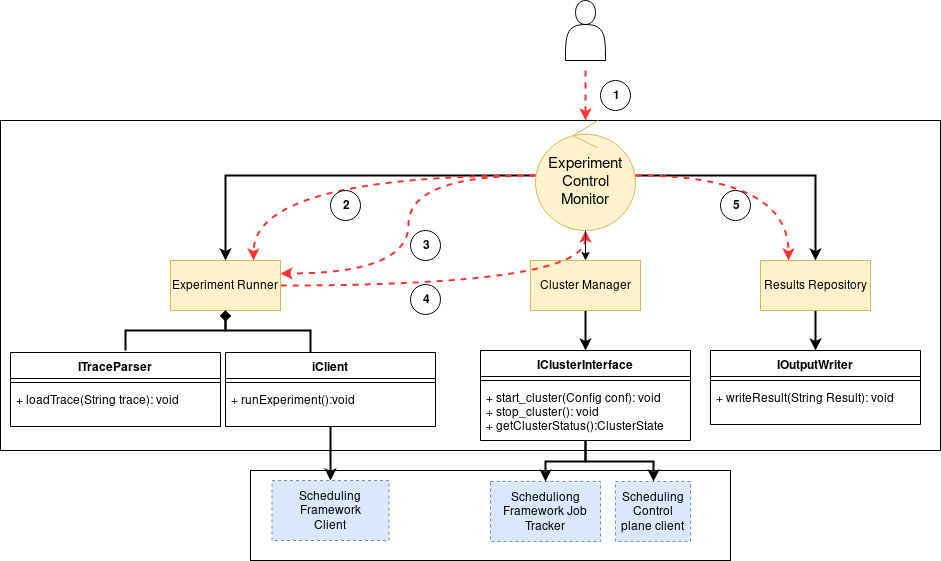
Identification of optimal scheduling configurations is achieved by comparison of results from test traces. The first stage of experimentation is often concerned with configuration of the cluster to enable or configure a feature of the scheduler. Comparing scheduler configurations of on the same platform 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.

When the trace is prepared and transformed into job descriptions, the job can be executed at the target scheduling framework. Collecting performance trace from the execution framework dependent and must be transformed to a common format so that the results can be analyses. Storing results persistently may also impact the final performance trace.

X

The workflow for his PRISM operates is as follows:

1. Client first initiates the experimental run by passing a path to a directory containing input traces.
2. The experimental control monitor, a control loop calls the *ExperimentalMonitor component to execute the jobs traces found in the provided directory.*
3. Periodically the *ExperimentalControlLopp* will poll the *ExperimentalRunner* to monitor the progress of experiment executions.
4. Completed experimental runs are processed by the ResultsRepository. The *ExperimentalControlMonitor* is responsible for identifying completed traces.

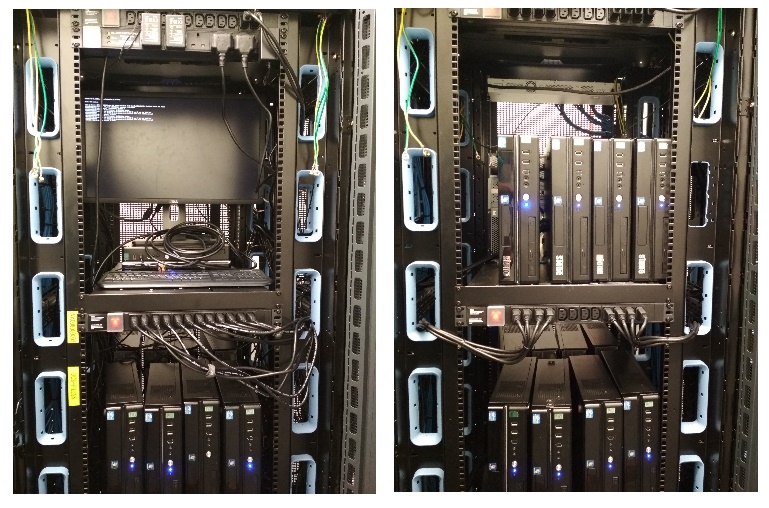
Figure X2: PRISM Framework

1. Experiment performance history can be exported by interfacing with the scheduling framework *MetricsServer*.

# PLATFORM CASE STUDY

We deployed the PRISM framework onto a medium-sized cluster experiment hosted at, consisting of 38 nodes (i7-4770 3.4Ghz (4 cores), RAM - 8 GB DDR3, Storage: 256GB SSD) as shown in Figure 2.

We managed our cluster as a PaaS infrastructure by deploying Kubernetes 1.15. Schedulers environements were deployed into Kubernetes clusters as isolated namespaces, enabling deployment of several concurrent platforms. We used Hadoop 2.9.2, persistance was managed by HDFS distributed filesystem, containing inputs and outputs of our wordcount jobs. We also deployed Apache Yarn as our case study scheduler. Both HDFS and Yarn were configured with a single master node for HDFS and Yarn, as well as 37 Worker nodes, managed as Kubernetes StatefulSets.



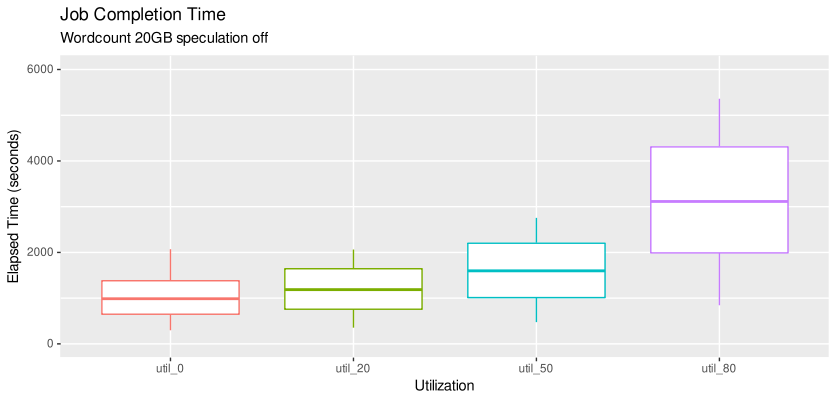
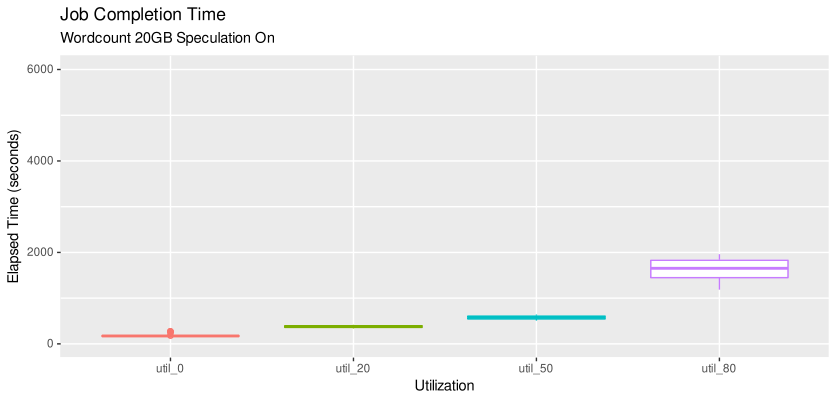
 

Figure 2. The Platform/Testbed/Cluster.

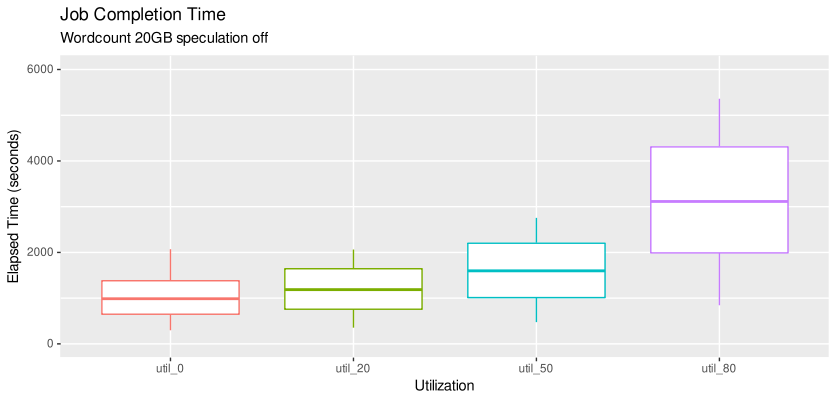
## Straggler Case Study

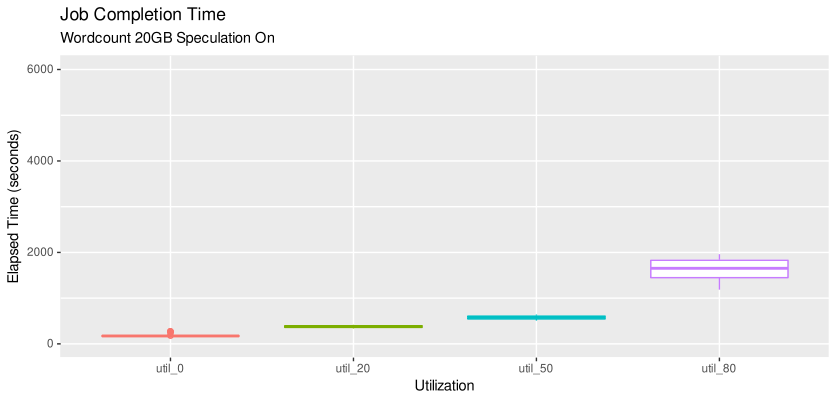
In order to demonstrate a potential use of PRISM, we have conducted experiments to study straggler manifestation under various system operational conditions. Previous work of production systems [R] have demonstrated that there exists some form of relationship between cluster resource contention and straggler manifestations. Therefore, we configured PRISM to execute experiment runs using Hadoop wordcount within yarn 2.9.2 scheduling framework, and controller the data input and cluster contention. Data input was configured at 20GB and 40Gb each, whilst CPU contention per node was configured at 0%, 20%, 50% and 80%. Moreover, we executed jobs with and without speculation [11] (i.e. automated container replica launched) in order to ascertain its impact upon the system. When combined together, this gives us a total of 16 unique experiment runs, and 4,800 unique job submitted (equivalent to XX days of cluster execution).

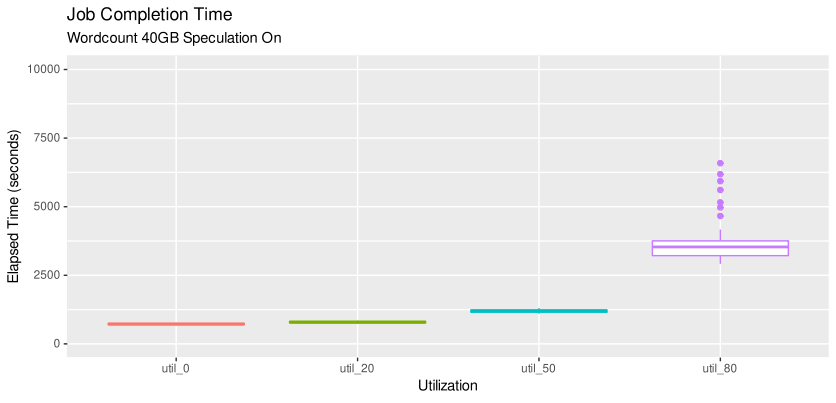
**5.1 Job Completion Time Analyses**

We first identify straggler jobs, we defined a straggler as any jobs who’s Job Completion Times (JCT) is greater than 150% of its mean completion time.

When speculative containers are disabled, there are no stragglers jobs, however in the worst case our results show jobs achieve 1100 seconds longer mean completion time. Furthermore worst case inter-quartile range (IQR) is ~4000 seconds longer.



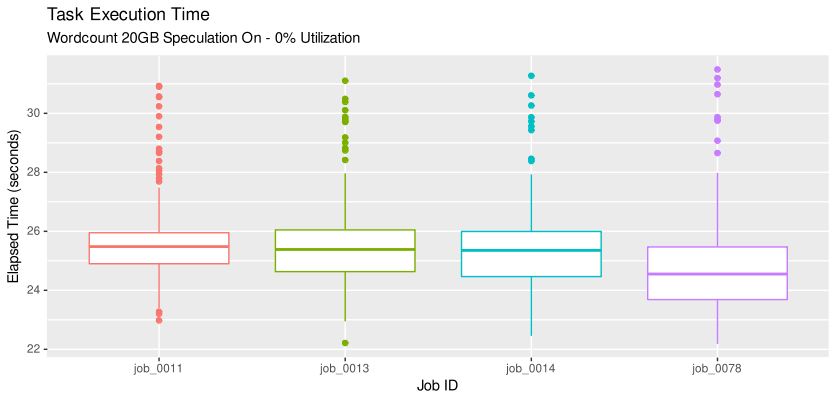
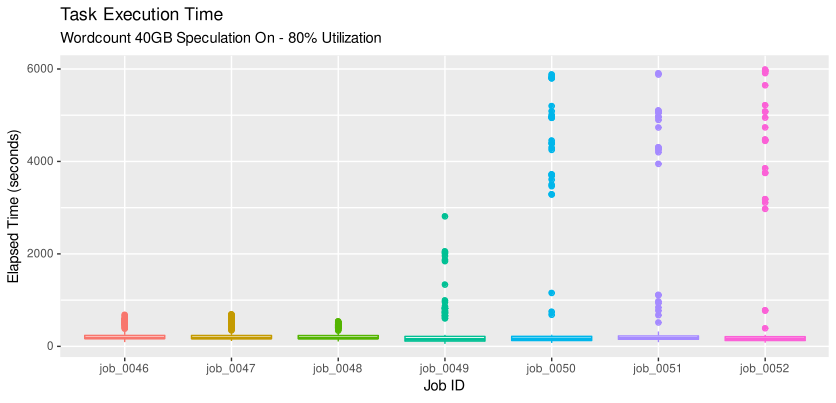
When speculation is enabled, JCT variance is reduced and JCT mean performance is improved. Four outliers when the cluster was at 0% utilization. In the worst case the outlier job had ~100 seconds greater than mean JCT.

Variance and mean completion time performance is degraded when speculation is disabled, reflecting the results found when the input file is 20GB. 7 Stragglers were identified, the worst case straggler performed 2974 seconds worse than the mean job completion time.

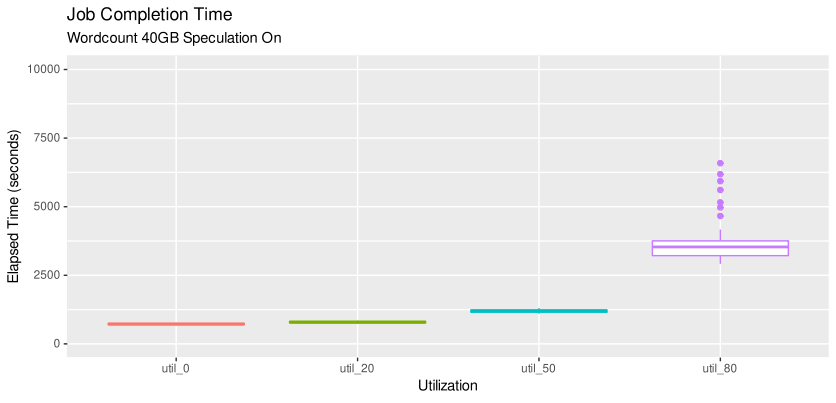
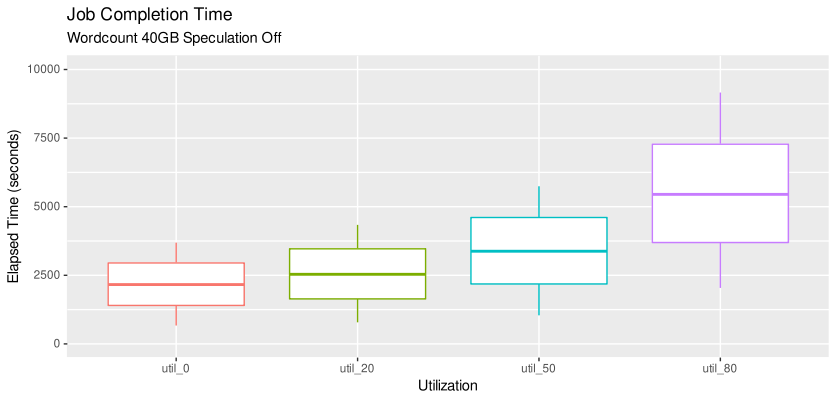
**5.3 Straggler Task Completion Time**

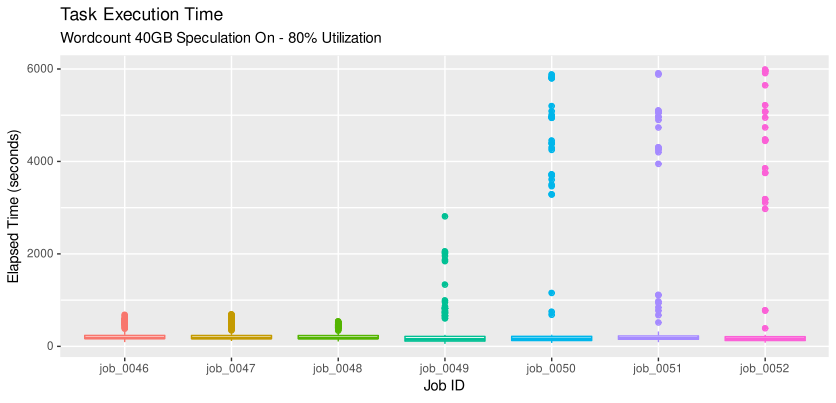
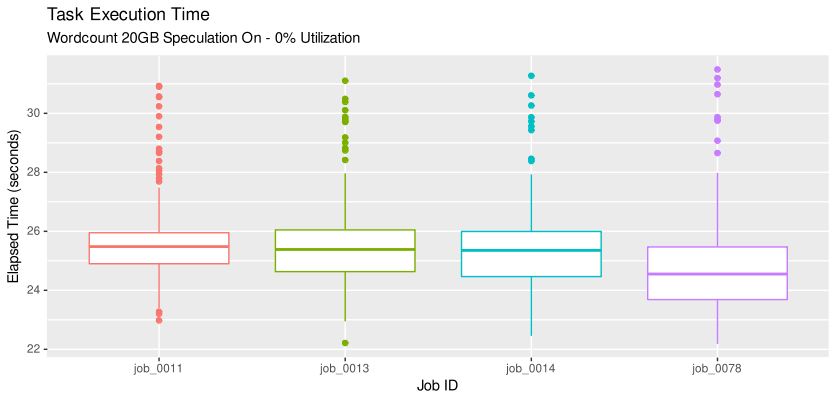
Task Completion Time (T.C.T.) measures latency of a task from when it has started until it completes. T.C.T. tail-latency is the main factor influencing straggler performance in mapreduce clusters[4], in this section we evaluate straggler task impacts on *Job Completion Time*.

Figure X describes 20gb straggler wordcount jobs, manifesting 199 straggler tasks. On average there were 50 straggler tasks, a minimum of 8 and a maximum of 16.

When the cluster was loaded at 80% and the data input size was increased to 40GB, we observed *126 straggler task* across 7 *Straggler Jobs* (*mean=18)*.

Stragglers manifested at both high (80%) and low (0%) cluster utilization, when executing wordcount with different input sizes (20GB, 40GB). Although Wordcount is a cpu-bound application, these results suggest processor interference has had little impact on straggler behavior.





By observing the sequenced *job\_id’s* of straggler jobs we can infer almost all straggler jobs happened in sequence. We suggest our observed straggler manifested are cause by constrained scheduler execution units (slots in *apache yarn*[ref:]). 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.

Our findings show that high processor load correlates with decreased task performance, but not tail latency. Furthermore, our results show data input size may not impact on straggler manifestation. We find scheduler architecture has the biggest impact. Furthermore, grouping of straggler behavior suggests temporal manifestation. Suggesting scheduler policy adaptation could be an appropriate counter measure.

1. DISSCUSSIONS AND TRENDS

This research work shows the development and deployment of a novel platform for building an automated testbed in clustered computing systems for straggler detection. We developed a case study for producing and analyses straggler behavior. PRISM allowed us to create several reconfigurable experiments to diagnose the straggler tasks. We were able to manifest straggler behavior by configuring data input sizes, and preloading

# CONCLUSIONS AND FUTURE WORK

In this research paper, we have proposed a platform to build an automated testbed and analyzed the straggler occurrence in distributed systems and measured the effects of straggler on system performance. We have run the wordcount application using real cluster and diagnose the occurrence of straggler with different value of data size and CPU limit to identify its trend. We have identified a need for dynamacity of slot based schedulers, capable of observing trends and adjusting scheduling policies.

Several limitations must be addressed in our PRISM framework. At this time our PRISM framework provide a control loop mediating between a administrator configuration and scheduling framework. At this time our interfaces are simple providing only enough control for execution of core features. In the future we will extend the framework to interface with telemetry services, as well as integration into Kubernetes[ref] platform as an ‘Operator’.

# ACKNOWLEDGMENTS

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

# REFERENCES

1. Garraghan, Peter, Xue Ouyang, Renyu Yang, David McKee, and Jie Xu. "Straggler root-cause and impact analysis for massive-scale virtualized cloud datacenters." IEEE Transactions on Services Computing (2016).
2. Ouyang, Xue, Peter Garraghan, Renyu Yang, Paul Townend, and Jie Xu. "Reducing late-timing failure at scale: Straggler root-cause analysis in cloud datacenters." In Fast Abstracts in the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. DSN, 2016.
3. Ananthanarayanan, Ganesh, Ali Ghodsi, Scott Shenker, and Ion Stoica. "Effective Straggler Mitigation: Attack of the Clones." In NSDI, vol. 13, pp. 185-198. 2013.
4. Coppa, Emilio, and Irene Finocchi. "On data skewness, stragglers, and MapReduce progress indicators." In Proceedings of the Sixth ACM Symposium on Cloud Computing, pp. 139-152. ACM, 2015.
5. Farhat, Farshid. "Stochastic modeling and optimization of stragglers in mapreduce framework." (2015). Thesis, The Pennsylvania State University.
6. Zaharia, Matei, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing." In Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, pp. 2-2. USENIX Association, 2012.
7. Nannicini, Giacomo. "Straggler Mitigation by Delayed Relaunch of Tasks." ACM SIGMETRICS Performance Evaluation Review 45, no. 2 (2018): 248-248.
8. Ananthanarayanan, Ganesh, Srikanth Kandula, Albert G. Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. "Reining in the Outliers in Map-Reduce Clusters using Mantri." In Osdi, vol. 10, no. 1, p. 24. 2010.
9. Bin Khunayn, Eman, Shanika Karunasekera, Hairuo Xie, and Kotagiri Ramamohanarao. "Exploiting Data Dependency to Mitigate Stragglers in Distributed Spatial Simulation." In Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, p. 43. ACM, 2017.
10. Wang, Da, Gauri Joshi, and Gregory Wornell. "Efficient Straggler Replication in Large-scale Parallel Computing." arXiv preprint arXiv:1503.03128 (2015).
11. Lei, Lei, Tianyu Wo, and Chunming Hu. "CREST: Towards fast speculation of straggler tasks in MapReduce." In e-Business Engineering (ICEBE), 2011 IEEE 8th International Conference on, pp. 311-316. IEEE, 2011.
12. Gill, Sukhpal Singh, Peter Garraghan, Vlado Stankovski, Giuliano Casale, Ruppa K. Thulasiram, Soumya K. Ghosh, Kotagiri Ramamohanarao, and Rajkumar Buyya. "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.
13. Qureshi, Nawab Muhammad Faseeh, Isma Farah Siddiqui, Asad Abbas, Ali Kashif Bashir, Keehyun Choi, Jaehyoun Kim, and Dong Ryeol Shin. "Dynamic Container-based Resource Management Framework of Spark Ecosystem." In 2019 21st International Conference on Advanced Communication Technology (ICACT), pp. 522-526. IEEE, 2019.
14. Calheiros, Rodrigo N., Rajiv Ranjan, Anton Beloglazov, César AF De Rose, and Rajkumar Buyya. "CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms." Software: Practice and experience 41, no. 1 (2011): 23-50.
15. V. K. Vavilapalli et al., “Apache Hadoop YARN,” Proc. 4th Annu. Symp. Cloud Comput. - SOCC ’13, pp. 1–16, 2013.