Analyzing Alibaba's Co-located Datacenter Workloads

Yue Cheng, Ali Anwar†, Xuejing Duan*
*George Mason University, †IBM Research–Almaden, *George Washington University*

***Abstract*—Warehouse-scale cloud datacenters co-locate workloads with different and often complementary characteristics for improved resource utilization. To better understand the challenges in managing such intricate, heterogeneous workloads while providing quality-assured resource orchestration and user experience, we analyze Alibaba's co-located workload trace, the first publicly available dataset with precise information about the category of each job. Two types of workload—long-running, user-facing, containerized production jobs, and transient, highly dynamic, non-containerized, and non-production batch jobs—are running on a shared cluster of 1313 machines. Through workload characterization, we find evidences that imply that one workload scheduler makes seemingly independent scheduling decisions regardless of the co-existence of the other. This upsurges an imminent need for a more integrated, global coordinating system that transparently connect multiple resource schedulers together and cohesively coordinates the multiple heterogeneous workloads for greater efficiency. Our multifaceted analysis reveals insights that we believe are useful for system designers and IT practitioners working on cluster management systems.**

I. Introduction

While modern datacenter management systems play a central role in delivering quality-assured cloud computing services, warehouse-scale datacenter infrastructure often comes with a tremendously huge cost of low resource utilization. Google's production cluster trace analysis reports that the overall utilization are between 20–40% [10] most of the time. Another study [30] observes that a fraction of Amazon servers hosting EC2 virtual machines (VMs) show an average CPU utilization of 7% over one week.

To improve resource utilization and thereby reduce costs, leading cloud infrastructure operators such as Google and Alibaba *co-locate* transient batch jobs with long-running, latency-sensitive, user-facing jobs [18], [37] on the same cluster. Workload co-location resembles hypervisor-based server consolidation [16] but at massive datacenter scale. At its core, the driving force is what is called a Datacenter Operating System [39], managing job scheduling, resource allocation, and so on. As one example, Google's Borg [37] adopts the workload co-location technique by leveraging resource isolation provided by Linux containers [7].

Workload co-location has become a common practice [18], [37] and there have been studies focusing on enabling more efficient co-location based cluster management [24], [25], [29], [33], [36]. To better facilitate the understanding of interactions among the co-located workloads and their real-world operational demands Alibaba recently released a cluster usage and co-located workload dataset [2], which was collected from Alibaba's production cluster in a 24-hour period.

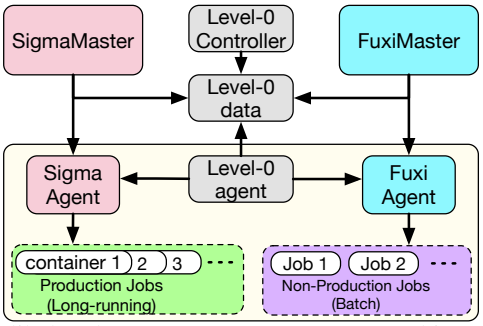


Fig. 1: Alibaba cluster management system architecture. Each server machine has an instance of Sigma agent and Fuxi agent which coordinate with the Level-0 management component to enable Sigma and Fuxi schedulers to work together.

We perform a characterization case study targeting Alibaba's co-located long-running and batch job workloads across several dimensions. We analyze the resource request and reservation patterns, resource usage, workload dynamicity, straggler issues, interaction and interference of co-located workloads, among other aspects. While confirming old issues still *surprisingly* persist (e.g., the straggler issues for batch workloads) that have long been observed in other work [11], we make several unique insightful findings. Some of them may be specific to the Alibaba infrastructure, but we believe the generality is critical and applicable to designers, administrators, and users of co-located resource management systems.

II. Background and Related Work

a) Cluster Trace Studies.: In 2011, Google open-sourced the first publicly available cluster trace data [4] spanning several clusters. Reiss et al. [35] study the heterogeneity and dynamicity properties of the Google workloads. Other works [26], [31] focus their studies on different aspects of the Google trace. Alibaba, the largest cloud service provider in China, released their cluster trace [2] in late 2017. Different from the Google trace, the Alibaba trace contains information about the two co-located container and batch job workloads, facilitating better understanding of their interactions and interferences. Lu et al. [32] perform characterization of the Alibaba trace to reveal basic workload statistics. Our study is focused on providing a microscopic view about how each co-located workload behaves and interact with each other.

b) Cluster Management Systems.: A series of state-of-the-art cluster management systems (CMSs) support co-located long-running and batch services. Monolithic CMSs

such as Borg [37][1] (and its open-sourced implementation Kubernetes [6]) and Quasar [25] use a centralized resource scheduler for performing resource allocation and management. Two-level CMSs such as Mesos [28] adopts a hierarchical structure where a Level-0 resource manager is used to jointly coordinate multiple Level-1 application-specific schedulers. Parallel CMSs achieve low-latency scheduling by using lock-free optimistic concurrency control techniques as in shared-state Omega [36] and Apollo [17]. Sparrow [34] is a de-centralized CMS that employs distributed resource managers to concurrently serve scheduling requests, thus achieving millisecond-level latency.

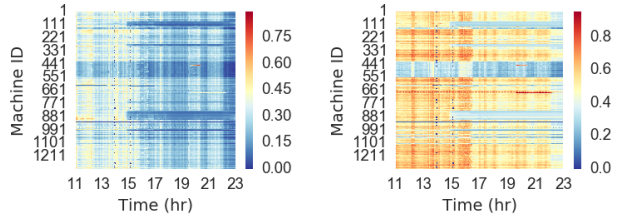
c) Alibaba's CMS and Co-located Workloads.: Alibaba's CMS resembles the architecture of a two-level CMS shown in Figure 1, which consists of three component: (1) a global Level-0 controller. (2) a Sigma [8] scheduler that manages the long-running workload, and (3) a Fuxi [40] scheduler that manages the batch workloads. Alibaba's online services are hosted by Alibaba's Pouch [9] container engine, managed by an orchestration system called Sigma [1]. Sigma is responsible for deployment decision making for **online service containers [9] for the production jobs**. These online services are user-facing, and typically require low latency and high performance. Sigma has been used for large-scale container deployment purposes by Alibaba for several years. It has also been used during the Alibaba Double 11 Global Shopping Festival. We analyze the long-running workloads in §IV.

Fuxi, on the other hand, manages **non-containerized non-production batch jobs**. Fuxi is used for vast amounts of data processing and complex large-scale computing type applications. Fuxi employs data-driven multi-level pipelined parallel computing framework, which is compatible with MapReduce [23], Map-Reduce-Merge [38], and other batch programming modes. We study the batch workloads in §V. The Level-0 mechanism coordinates and manages both types of workloads (Sigma and Fuxi) for coordinated global operations. Particularly the Level-0 controller performs four important functionalities: (1) manages colocation clusters, (2) performs resource matching between each scheduling tenant, (3) strategies for everyday use and use during large-scale promotions, and (4) performs exception detection and processing. Different types of workloads were running on separate clusters before 2015, since when Alibaba has been making effort to co-locate them on shared clusters.

d) Alibaba Cluster Trace.: The Alibaba cluster trace captures detailed statistics for the co-located workloads of long-running and batch jobs over a course of 24 hours. The trace consists of three parts: (1) statistics of the studied homogeneous cluster of 1313 machines, including each machine's hardware configuration[2], and the runtime {CPU, Memory, Disk} resource usage for a duration of 12 hours (the 2^{nd} half of the 24-hour period); (2) long-running containerized job workloads, including a trace of all container deployment requests and actions, and a resource usage trace for 12 hours; (3) co-located batch job[3] workloads, including a trace of all batch job requests and actions, and a trace of per-instance resource usage over 24 hours. Unlike the Google trace [35] that lacks precise information about exact purpose of individual jobs, the Alibaba trace well compensates this by tracing the two different workloads separately, thus offering researchers visibility of real-world operational demands of co-located workloads.

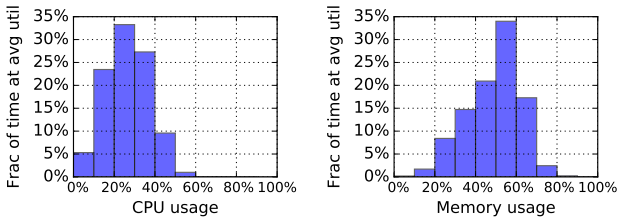
[1]Technically, Borg partitions the whole cluster into cells and assigns a replicated scheduler for each cell that provides a certain set of services, e.g., batch services.

[2]Over 99.6% machines have the same hardware composition (64-core CPU, normalized memory capacity 0.69 (normalized by the largest memory capacity), and normalized disk capacity 1, except a few with slightly different memory capacity).



(a) CPU usage. (b) Memory usage.

Fig. 2: Cluster resource usage (hour 11–23).



(a) Average CPU usage. (b) Average memory usage.

Fig. 3: Histogram of average resource usage.

III. Overall Cluster Usage

We start with the overall cluster usage to understand the aggregated workload characteristics. Figure 2 shows the per-machine resource usage temporal pattern across the 12-hour duration. Cluster CPU utilization (Figure 2(a)) is at medium level (40%–50%) for the first 4 hours but decreases during the rest time, while memory utilization (Figure 2(b)) is above 50% in majority of the time. This can also be reflected from Figure 3, which shows the average usage distribution. The cluster spends over 80% of its time running between 10%–30% CPU usage (Figure 3(a)). Average memory usage (Figure 3(b)) is relatively higher and in over 55% of the time the machines have a memory usage above 50%. Google trace analysis [35] reports that the CPU and memory usage were capped at 60% and 50%, respectively. In contrast, at Alibaba, memory tends to be of higher demand with an average of over half of the overall capacity consumed for over half of the tracing duration.

[3]Each job forms a directed acyclic graph (DAG), consisting of one or more tasks; each task has multiple instances; all instances within a task execute the same binary following a single program multiple data (SPMD) model commonly seen in MapReduce-style batch processing frameworks; instance is the smallest scheduling unit of a batch job scheduler.

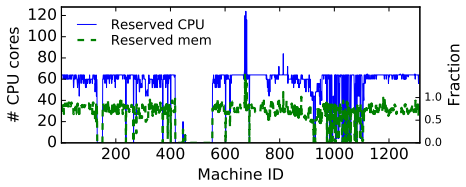


Fig. 4: Distribution of reserved resources at container creation time across the cluster (note reserved memory capacity (normalized) are shown on secondary Y-axis).

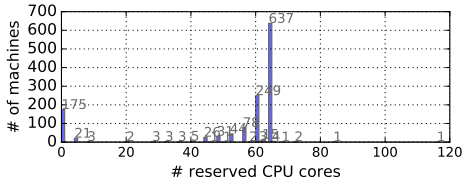


Fig. 5: Histogram of reserved CPU cores.

IV. Long-Running Job Workloads

A. Resource Reservation

We first calculate the per-machine resource amount requested and reserved by containerized long-running jobs from the container request trace (`container_event.csv`). Figure 4 show that, except for a few servers with overbooked CPUs (spikes in machine 671–679, and semi-spikes in machine 797–829), the machine-level CPU allocation are consistently capped at 64–the maximum number of CPU cores of one machine. Figure 5 and Figure 6 plot the same trend: for both CPU and memory allocation, overbooking is rare but does exist; about half of the machines (637) get all 64 cores reserved for containerized long-running applications; 60–80% of memory on 74% machines are reserved for containers. Deep troughs in Figure 4 are due to zero container deployment on the machines. In fact, container management systems such as Sigma [8] need to consider a lot of other constraints when performing scheduling for long-running containers, including affinity and anti-affinity constraints (e.g., co-locating applications that belong to the same services for reducing network cost, or co-locating applications with complementary runtime behaviors), application priorities, whether or not the co-located applications tolerate resource overbooking of the same host machine [27]. A side effect of such multi-constraint multi-objective optimization is that the number of containers is unevenly distributed across the cluster as shown in Figure 7.

B. Resource Usage

The container workload trace (`container_usage.csv`) samples the resource usage of each container every 5 minutes. We aggregate all the container-level resource usage statistics by the machine ID based on $container \rightarrow machine\_ID$ mapping recorded in the `container_event.csv` file and plot the resource usage heat map shown in Figure 8. The dominating pattern is the horizontal stripes across the 12-hour tracing period[4]. Each stripe corresponds to one machine holding multiple containers, clearly reflecting the long-running nature of containerized applications. In fact, container re-scheduling and migration is not fully integrated into Sigma yet, according to our communication with Alibaba, thus lacking flexibility to some extent. Another observation we make is that, even though Sigma tries to balance out the amount of reserved resources (as one constraint of the scheduling heuristic), the actual CPU and memory usage by container workloads are imbalanced across all machines.

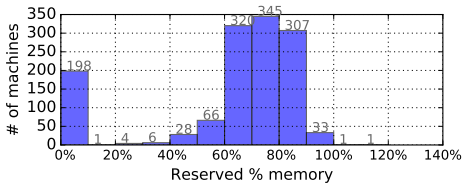


Fig. 6: Histogram of reserved memory.

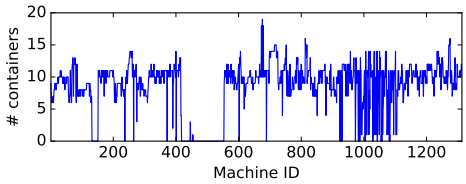


Fig. 7: Distribution of containers across the cluster.

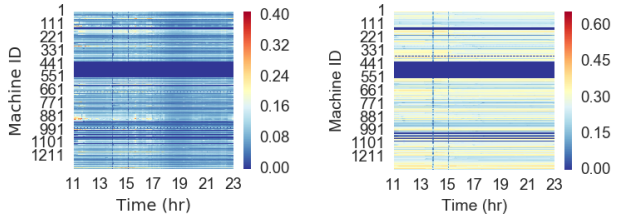
C. Resource Over-Provisioning and Usage Dynamicity

To better understand the observed resource usage imbalance, we compare the reserved CPU and memory capacity with the actual usage, as shown in Figure 9. We make the following observations: (1) CPU resources are over-provisioned by all containers (Figure 9(a)), while memory resources are over-committed by a large majority of containers (Figure 9(b)). (2) The CPU and memory request patterns are clearly visible—CPU requests have 4 distinguishable patterns while memory requests have 6. (3) $\sim 60\%$ of containers are inactive in terms of CPU usage, having less than 1% average CPU utilization with the maximum percentile capped at 3%; average resource usage correlates to temporal stability—the more resource consumed on average, the higher the temporal variation tends to be—this is especially true for CPU; memory usage dynamicity is not as high as that of CPU. (4) Most containers have a higher average memory usage above 2%, which is as expected since containers need a minimum amount of memory to keep online services functional. This is, in fact, consistent with the well-studied behaviors of web-scale distributed storage workloads [14], [19], [20], [13], [12].

D. Insights

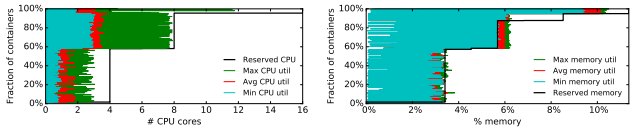
Based on the observations, we infer the following. (1) A majority of the long-running containerized interactive services stay inactive (at least during the 12-hour trace period); this

[4]The wide dark blue stripe showing no container activity is due to that Alibaba intentionally leave a portion of machines as buffer area which, if the long-running applications (containers) are of low load, are solely designated for running batch jobs. When online service are experiencing high load, batch jobs running in buffer area can be evicted and resources can be preempted by the containers [8].



(a) CPU usage. (b) Memory usage.

Fig. 8: Container job resource usage (hour 11–23).



(a) CPU. (b) Memory.

Fig. 9: Reserved CPU and memory vs. their actual usage. We break down each container's 12-hour usage changes into {Min, Avg, Max} to show the dynamicity. The X-axis is sorted by the reserved resource amount.

finding is consistent with other interactive workload studies [15][5], implying a demand for elastic interactive service systems. (2) Long-running services are relatively more memory-intensive; ample CPU resources reserved through resource over-provisioning are needed to provide performance guarantee (stringent latency and throughput requirement); this is especially true for in-memory computing whose first-priority resources are essentially memory[6]. (3) It is possible to make accurate resource usage prediction based on the temporal usage dynamicity profiling [22], especially for memory which has a relatively stable usage pattern (see Figure 9(b)); the prediction can be used for more informed resource management decision making such as container re-scheduling/migration. (4) Our observation indicates a need for container orchestration frameworks such as Sigma and Kubernetes [6] to support (i) dynamic container re-scheduling or migration for long-running containerized applications, and (ii) flexible and low-latency scheduling for short-lived tasks (such as machine learning), of which lack a robust resource scheduling support.

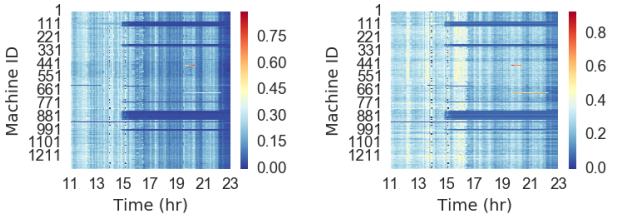
V. Batch Job Workloads

A. Resource Usage

The batch job workloads exhibit different resource patterns compared to that of containerized long-running job workloads. We calculate the batch job workload resource usage shown in Figure 10 by subtracting the usage of containers (Figure 8) from the overall usage (Figure 2) of the cluster. We confirmed the accuracy of the results by comparing it against the sum of all batch task instances' runtime usage values under a certain timestamp. The vertical stripes (batch job waves) in Figure 10 is due to the dynamic nature of batch job workloads—task

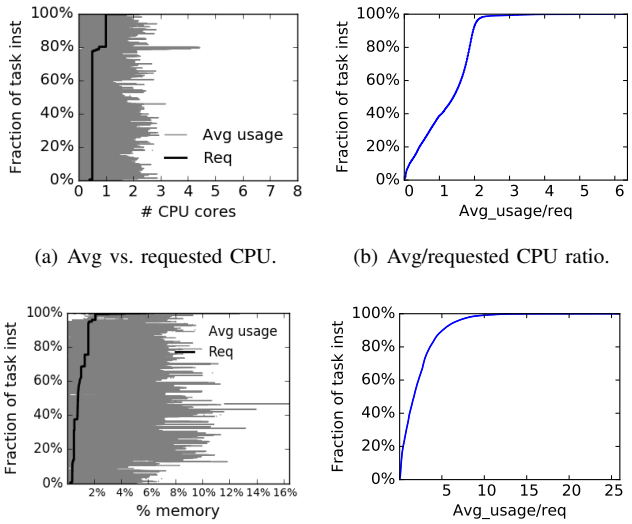
[5]E.g., Facebook's largest Memcached pool ETC deploys hundreds of Memcached servers but only absorbs an incredibly low average of 50K queries per second [15].

[6]Being consistent with the observations made in the Memcached workload study [15], [20], Memcached is memory-bound rather than CPU-bound.



(a) CPU usage. (b) Memory usage.

Fig. 10: Batch job resource usage (hour 11–23).



(a) Avg vs. requested CPU. (b) Avg/requested CPU ratio.

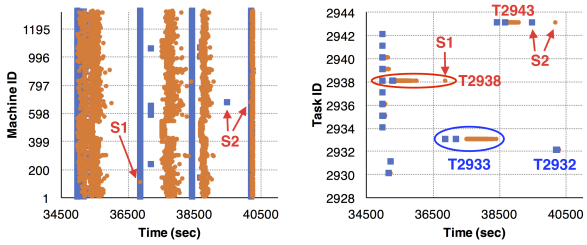
(c) Avg vs. requested memory. (d) Avg/requested memory.

Fig. 11: CDF of the average CPU and memory usage relative to the resource request for the corresponding task instance.

instances are transient and most of them finish in seconds. We can also observe that, within a single wave, the CPU and memory resource usage are roughly balanced across the cluster, except for some regions with no batch jobs scheduled (i.e., the horizontal, dark blue stripes). This is because: (1) Fuxi is not constrained by data locality thanks to compute and storage disaggregated infrastructure at Alibaba (for batch jobs all data are stored and accessed remotely [3]), hence task instances can be scheduled anywhere there is enough resource[7]; (2) Fuxi adopts an incremental scheduling heuristic that incrementally fulfills the resource demands at per-machine level [40].

Figure 11 depicts the CDFs of requested CPU amount vs. runtime CPU usage. The over-commitment trend is clearly shown in Figure 11(a) and Figure 11(c). Quantitatively, $\sim 30\%$ task instances use CPU cores more than requested (Figure 11(b)), while more than 70% task instances have an overcommitment ratio (avg_usage : req ratio) from 1–5 (Figure 11(d)). This is understandable, as a large number of short-task-dominated, transient batch jobs can *elastically over-commit* resources that are originally reserved by the *over-provisioned long-running jobs*.

[7]Batch job trace has no disk statistics capturing intermediate data storage usage.



(a) Execution waves. (b) DAG.

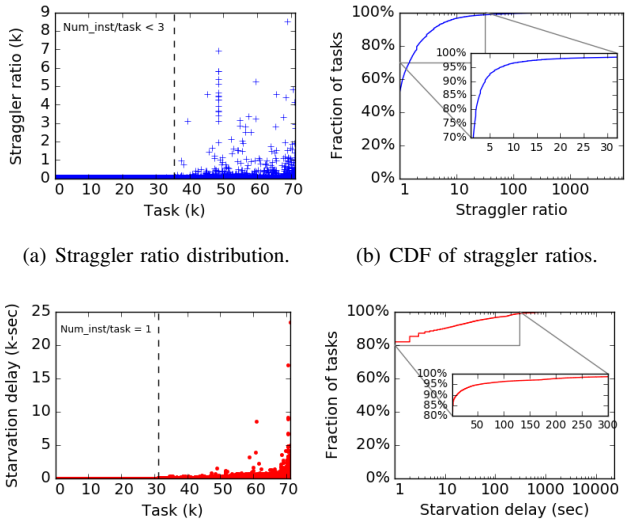
Fig. 12: Job 556's execution profile; blue squares correspond to the start of the makespan for a specific task instance, and orange circles correspond to the end of the makespan. Figure 12(a) depicts the task waves with Y-axis showing the machine IDs. Figure 12(b) plots the job DAG with Y-axis showing the task IDs. `S1`: Straggler 1; `S2`: Straggler 2; `T2933`: Task 2933. We can infer the DAG dependencies as follows: T2932 waits for T2943 to complete; T2943 depends on T2933; T2933 waits for all tasks of the 1^{st} wave to finish.

B. Task Scheduling

Trace files `batch_task.csv` and `batch_instance.csv` contain the detailed batch job profile information including job composition (how many tasks per job and how many instances per task), spacial (how task instances are mapped to the machines) and temporal information (how long each task instance runs), and average/max resource usage (though not complete). By combining the spacial/temporal and job composition information, we can easily infer the DAG structure of a specific batch job.

One question we want to answer is whether the well-studied issue [11] still persists in Alibaba's datacenters–The answer is surprisingly yes. To illustrate the impact of stragglers on job performance, Figure 12 visualizes Job 556's execution. We can easily spot two stragglers. `S1`, which has a starting timestamp same as the rest other instances of the same task `T2938` but an end timestamp way behind the rest, results in delayed scheduling and execution of `T2933` (the 2^{nd} wave in Figure 12(a)). `S2`, belonging to `T2943`, has a lagged starting timestamp greater than that of the rest; a direct result is the delayed execution of `T2932` (the 4^{th} wave in Figure 12(a)). To distinguish between these two typical cases, we call the `S1` kind of task instances as **stragglers**, and the `S2` kind of tasks instances **starvers**.

We then scan the execution profiles of all tasks included in the trace. We iterate through all tasks and calculate the *straggler ratios* (defined as the ratio of the maximum and minimum instance makespan of the corresponding tasks) and *starvation delays* (defined as the difference between the largest and the smallest instance starting timestamp of the corresponding tasks). Figure 13(a) and Figure 13(c) depict the straggler and starver distribution, respectively. As the task size (i.e., number of instances per task) increases, the straggler ratio and starvation delay increases accordingly. As shown in Figure 13(b), around 50% tasks have a straggler ratio of 1, because half of them have 1 or 2 instances. $\sim 7\%$ tasks have a straggler ratio greater than $5\times$, with the highest as $8522\times$.



(a) Straggler ratio distribution. (b) CDF of straggler ratios.

(c) Starvation delay distribution. (d) CDF of starvation delays.

Fig. 13: Straggler and starver analysis. In Figure 13(a) and Figure 13(c), the X-axis is sorted smallest to largest by the number of instances per task.

Figure 13(d) shows that $\sim 4\%$ tasks have a starvation delay longer than 100 seconds, with the longest as 23379 seconds.

We further investigate the reasons by looking into the detailed task traces. Straggler patterns included in the trace and possible causes are classified as follows. (1) Some straggler is significantly slower than the rest, which seems a common case due to either misconfiguration (e.g., disabling speculative execution [5]) or severely imbalanced load (e.g., a task instance getting more work to do than the rest). (2) Some task instances have long execution time and hence suffer a higher chance of getting failed; failed instances get re-scheduled (caught in the trace) and as a result suffer long starvation delay. (3) A straggler is spotted and interrupted[8] at a very late time (one extreme example in the trace: few hours after 99% of task instances of have finished the execution, while most of other task instances finish in seconds) by Fuxi; as a result, Fuxi launches a speculative backup instance, which may finish quick or take very long time to finish; either way, this scenario results in a false positive long starvation delay, or much worse–an extremely large straggler ratio. (4) Some task instance is interrupted and gets indefinitely starved (possibly due to low priority) while waiting for re-scheduling, resulting in surprisingly long starvation delay.

C. Insights

Based on the observations, we infer the following. (1) Workloads of transient batch jobs with many short-lived tasks exhibit resource patterns perfectly complementing that of over-provisioned, long-running, mostly CPU-inactive, containerized applications; those free resources not yet consumed by the long-running applications have to be efficiently utilized by

[8]Fuxi's speculative execution technique where Fuxi interrupts long-tail instance and launches backup instance [40].

batch jobs; essentially, this is key to improve the overall cluster utilization; furthermore, users do not quite care about accurately estimating the actual resource usage of the to-be-submitted batch jobs; (2) The notoriously persisting straggler issues might be eliminated by enforcing/adding more comprehensive configuration settings at the administrator/developer side or via more careful data partitioning planning at the user side. For example, for interrupted task instances that are suffering from starvation (Fuxi failed to re-schedule them in time), some priority-based scheme (e.g., MLFQ-liked scheduling heuristic [21]) may help. Users also take responsibilities. Users should carefully plan on data partitioning to avoid imbalanced input assignments.

VI. Conclusion and Future Work

Aiming at improving the overall resource utilization, workload co-location results in exponentially increased complexity for warehouse-scale datacenter resource management. Analysis of the Alibaba cluster trace reveals such challenges, for which new resource scheduling approach that has a deeper sense of co-location will likely be necessary.

In our characterization study we answer pertinent questions about each of the two co-located workloads–a long-running containerized workloads serving online production jobs, and a short-lived, dynamic, and transient, batch job workloads serving offline non-production jobs. We find that, though two workloads are complementary to each other due to their static and dynamic natures, it would benefit to the overall efficiency if the static long-running job schedulers could add some level of flexibility by incorporating optimizations such as container re-scheduling or migration. We also find evidences implying disjoint existence of Sigma and Fuxi, stressing the need for a more integrated, co-location optimized solution.

Acknowledgments We thank Haiyang Ding from Alibaba for his valuable feedback on an early version of this manuscript. This work was sponsored in part by an AWS Cloud Research Grant.

References

[1] Alibaba cluster management system Sigma (in Chinese). https://goo.gl/7dLBso.
[2] Alibaba production cluster data. https://github.com/alibaba/clusterdata.
[3] Alibaba Vendor BoF. https://www.usenix.org/conference/fast18/bofs#alibaba.
[4] Google cluster workload traces. https://github.com/google/cluster-data.
[5] Hadoop Tutorial: Fault Tolerance. https://developer.yahoo.com/hadoop/tutorial/module4.html.
[6] Kubernetes. https://kubernetes.io/.
[7] Linux Containers. https://linuxcontainers.org/.
[8] Maximizing CPU Resource Utilization on Alibaba's Servers. https://102.alibaba.com/detail/?id=61.
[9] Pouch Container Engine. https://github.com/alibaba/pouch.
[10] The Datacenter as a Computer. https://doi.org/10.2200/S00516ED2V01Y201306CAC024.
[11] Ananthanarayanan, G., Ghodsi, A., Shenker, S., and Stoica, I. Effective straggler mitigation: Attack of the clones. In *NSDI 13* (2013), USENIX.
[12] Anwar, A., Cheng, Y., Gupta, A., and Butt, A. R. Mos: Workload-aware elasticity for cloud object stores. In *ACM HPDC '16*.
[13] Anwar, A., Cheng, Y., Gupta, A., and Butt, A. R. Taming the cloud object storage with mos. In *ACM PDSW '15* (2015).
[14] Anwar, A., Mohamed, M., Tarasov, V., Littley, M., Ruprecht, L., Cheng, Y., Zhao, N., Skourtis, D., Warke, A. S., Ludwig, H., Hildebrand, D., and Butt, A. R. Improving docker registry design based on production workload analysis. In *USENIX FAST 18)* (2018).
[15] Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., and Paleczny, M. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS* (2012).
[16] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. In *ACM SOSP '03* (2003).
[17] Boutin, E., Ekanayake, J., Lin, W., Shi, B., Zhou, J., Qian, Z., Wu, M., and Zhou, L. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *USENIX OSDI 14* (2014).
[18] Burns, B., Grant, B., Oppenheimer, D., Brewer, E., and Wilkes, J. Borg, omega, and kubernetes. *Queue 14*, 1 (Jan. 2016).
[19] Cheng, Y., Gupta, A., and Butt, A. R. An in-memory object caching framework with adaptive load balancing. In *ACM EuroSys '15*.
[20] Cidon, A., Rushton, D., Rumble, S. M., and Stutsman, R. Memshare: a dynamic multi-tenant key-value cache. In *USENIX ATC 17)* (2017).
[21] Corbató, F. J., Merwin-Daggett, M., and Daley, R. C. An experimental time-sharing system. AIEE-IRE '62 (Spring), ACM.
[22] Cortez, E., Bonde, A., Muzio, A., Russinovich, M., Fontoura, M., and Bianchini, R. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *ACM SOSP '17* (2017).
[23] Dean, J., and Ghemawat, S. Mapreduce: Simplified data processing on large clusters. In *USENIX OSDI '04* (2004).
[24] Delimitrou, C., and Kozyrakis, C. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *ACM ASPLOS '13* (2013).
[25] Delimitrou, C., and Kozyrakis, C. Quasar: Resource-efficient and qos-aware cluster management. In *ACM ASPLOS '14* (2014).
[26] Di, S., Kondo, D., and Cirne, W. Characterization and comparison of cloud versus grid workloads. In *2012 IEEE International Conference on Cluster Computing* (Sept 2012), pp. 230–238.
[27] Garefalakis, P., Karanasos, K., Pietzuch, P., Suresh, A., and Rao, S. Medea: Scheduling of long running applications in shared production clusters. In *ACM EuroSys '18* (2018).
[28] Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R., Shenker, S., and Stoica, I. Mesos: A platform for fine-grained resource sharing in the data center. In *USENIX NSDI '11* (2011).
[29] Janus, P., and Rzadca, K. Slo-aware colocation of data center tasks based on instantaneous processor requirements. In *Proceedings of the 2017 Symposium on Cloud Computing* (New York, NY, USA, 2017), SoCC '17, ACM, pp. 256–268.
[30] Liu, H. A measurement study of server utilization in public clouds. In *2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing* (Dec 2011), pp. 435–442.
[31] Liu, Z., and Cho, S. Characterizing machines and workloads on a google cluster. In *2012 41st International Conference on Parallel Processing Workshops* (Sept 2012), pp. 397–403.
[32] Lu, C., Ye, K., Xu, G., Xu, C. Z., and Bai, T. Imbalance in the cloud: An analysis on alibaba cluster trace. In *IEEE BigData '17* (2017).
[33] Mars, J., Tang, L., Hundt, R., Skadron, K., and Soffa, M. L. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *ACM MICRO '11* (2011).
[34] Ousterhout, K., Wendell, P., Zaharia, M., and Stoica, I. Sparrow: Distributed, low latency scheduling. In *ACM SOSP '13* (2013).
[35] Reiss, C., Tumanov, A., Ganger, G. R., Katz, R. H., and Kozuch, M. A. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *ACM SoCC '12* (2012).
[36] Schwarzkopf, M., Konwinski, A., Abd-El-Malek, M., and Wilkes, J. Omega: Flexible, scalable schedulers for large compute clusters. In *ACM EuroSys '13* (2013).
[37] Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., and Wilkes, J. Large-scale cluster management at google with borg. In *ACM EuroSys '15* (2015).
[38] Yang, H.-c., Dasdan, A., Hsiao, R.-L., and Parker, D. S. Map-reduce-merge: Simplified relational data processing on large clusters. In *ACM SIGMOD '07* (2007).
[39] Zaharia, M., Hindman, B., Konwinski, A., Ghodsi, A., Joesph, A. D., Katz, R., Shenker, S., and Stoica, I. The datacenter needs an operating system. In *USENIX HotCloud '11* (2011).
[40] Zhang, Z., Li, C., Tao, Y., Yang, R., Tang, H., and Xu, J. Fuxi: A fault-tolerant resource management and job scheduling system at internet scale. *Proc. VLDB Endow.*.