TOPOSCH: Latency-Aware Scheduling Based on Critical Path Analysis on Shared YARN Clusters

Chunming Hu[12], Jianyong Zhu[12], Renyu Yang[32]✉, Hao Peng[42], Tianyu Wo[12],
Shiqing Xue[12], Xiaoqiang Yu[12], Jie Xu[32], Rajiv Ranjan[5]
[1]SKLSDE Lab, Beihang University, China
[2]Beijing Advanced Innovation Center for Big Data and Brain Computing (BDBC), Beihang University, China
[3]School of Computing, University of Leeds, UK
[4]School of Cyber Science and Technology, Beihang University, China
[5]School of Computing, Newcastle University, UK
{*r.yang1, j.xu*}*@leeds.ac.uk*; {*zhujy,hucm,woty,penghao, xuesq,yuxq*}*@act.buaa.edu.cn*; *raj.ranjan@newcastle.ac.uk*

Abstract—**Balancing resource utilization and application QoS is a long-standing research topic in cluster resource management. Big data YARN clusters need to co-schedule diverse workloads on shared resources including batch processing jobs, streaming jobs, and other long-running applications such as web services, database services, etc. Current resource managers are only responsible for resource allocation among applications/jobs but completely unaware of runtime QoS requirements of interactive and latency-sensitive applications. Prior works to maximize the QoS of monolithic applications ignore inherent dependencies and temporal-spatio performance variability of components, characteristics of distributed applications primarily driven by microservices. In this paper, we present TOPOSCH, a new resource management system to adaptively co-locate batch tasks and microservices by harvesting runtime latency. In particular, TOPOSCH tracks full footprints of every request across microservices over time. A latency graph is periodically generated for identifying victim microservices through an end-to-end latency critical path analysis. We then exploit per-microservice and per-node risk assessment to gauge the visible resources to the capacity scheduler in YARN. Execution of batch tasks are adaptively throttled or delayed, thereby avoiding latency increase due to node over-saturation. TOPOSCH is integrated with YARN and experiments show that the latency of DLRAs can be reduced by up to 39.8% against the default capacity scheduling in YARN.**

Index Terms—**latency sensitivity, workload co-location, microservice, cluster management**

I. INTRODUCTION

It is a long-standing challenge to achieve a high degree of resource utilization in cluster scheduling. Workloads co-location – physically co-scheduling diverse tasks onto the same host server – has become a common practice in improving resource utilization and cost efficiency. In big data YARN clusters, workloads typically encompass batch processing jobs, stream processing jobs, and other long-running applications such as web services, NoSQL, etc. With the advancement of microservice and container techniques, distributed long-running application (DLRA) has been of the upmost importance and criticality due to the independent function decoupling and in-between lightweight communications.

A DLRA typically comprise multiple microservices, which are deployed on multiple nodes subject to their resource requirements. Multiple transactions within a DLRA have strong dependencies across multiple microservices. Load variability, however, indicates a temporal-spatio behaviors over time and across nodes [1][2][3][4]. A user request (e.g., an application request, a database query, a file access operation) will transverse a collection of microservices before being responded. Therefore, end-to-end (E2E) response latency is broadly used to indicate the execution time of any operation to complete.

Current cluster managers [5] [6][7][8][9] are designed for short-running tasks within batch jobs, whose performance is minimally affected when launching additional tasks. The central resource manager (RM) is application-agnostic and completely unaware of runtime QoS requirements of interactive and latency-sensitive applications; RM is only responsible for resource allocation among jobs but leaves all application-specific logic to application managers. Existing solutions of workload co-location either aim at reducing the performance interference through resource partition and isolation [10][11][12] or leverage QoS-aware scheduling to place different jobs/applications by minimizing interference [13][14][15]. However, they are optimized towards the monolithic application and have indirect effects on DLRAs that have more sophisticated component dependencies and performance variations (e.g., latency) due to a vast number of requests across entire system components.

To address above problems, we present TOPOSCH, a resource management system that can continuously harvest the status of massive requests and track them across different microservices, and employ per-microservice and per-node risk assessment of QoS violation to adaptively schedule resources to batch jobs and DRLAs. TOPOSCH adopts a latency-driven methodology to navigate the task placement under the capacity scheduler in original YARN in order to coordinate workloads' performance. Specifically, to capture the spatio-temporal variations and localize performance hotpots, we exploit instrumentation to trace each request and record footprints of all requests across different microservices. We then calculate the average sojourn (processing) time on individual microservice

✉Renyu Yang is the corresponding author

and average transmission time between microservices. Based on the aggregated tracing information, we form a latency graph and periodically analyze the critical path – the chain of invocations with the longest end-to-end latency across all microservices – to find out the victim microservices that tend to have higher risks of QoS violation due to co-location. Node-level risk assessment is further employed to gauge the visible resources to be scheduled to batch tasks and task scheduling is adaptively delayed to give way to microservices without over-saturating the node resources. We modify the state-of-the-art YARN capacity scheduler and experiments show that the average latency of DLRAs can be reduced by up to 39.8% against the default capacity scheduling in native YARN. Particularly, the main contributions of this paper are as follows:

- A mechanism for tracing and breaking down the E2E latency of a request among constituent microservices of DLRAs.
- A per-microservice and per-node risk assessment method by exploiting latency critical path analysis.
- An adaptive task placement with adjustment of visible resources and delay scheduling of batch tasks to reduce the probability of violating microservices' QoS.

TOPOSCH is open-sourced and can be downloaded from `https://github.com/MSDS-ABLE/toposch`.

Organization. We firstly depict the background and challenges facing the design of TOPOSCH in §2 and introduce the key design and architecture in §3. More technical details are presented in §4 to §6. Experiments are shown in §7. Following a review of related work in §8, we finally draw the conclusions and discuss future works.

II. BACKGROUND AND MOTIVATION

A. Background

Cluster resource management. Cluster scheduling systems typically separate the resource management layer from the job-level logical execution plans. YARN[16] and Fuxi[8] share the following components: *Resource Manager (RM)* is the centralized resource manager, tracking resource usage, node aliveness, enforcing resource quotas among tenants through either capacity or fairness control. *Application Master (AM)* is an application-level scheduler which coordinates the logical plan of a single job by requesting resources from the RM, generating a plan from received resources, and coordinating task execution. *Node Manager (NM)* is a daemon process within each cluster node and responsible for managing task life-cycle and monitoring node information.

Workload Colocation. In YARN clusters, resources are usually consumed by various workloads mainly including batch jobs and long running applications (LRAs). Batch analytic jobs are big data processing applications that are insensitive to latency [8][17]. They are mainly measured by the E2E completion time, and thus deadline-constrained. A job can be typically segmented into a large number of short-lived tasks with only subsecond or seconds duration. LRAs typically encompass transaction analytics, online web services, or

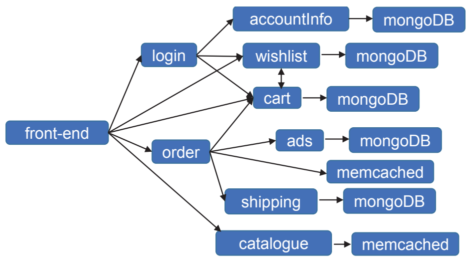

Fig. 1. An e-commerce DLRA for online clothing store [18]

database services (e.g., HBase, Memcached, MongoDB, etc). Their durations range from hours to months, and are typically latency-sensitive. Response latency and throughput are the key performance indicators and applications must meet strict QoS.

Latency-sensitive applications and microservices. Microservice architectural style is an approach to constructing a single application as a set of small interconnected services. Each microservice runs individually and communicates with each other via light-weight protocol, e.g., HTTP resource API. In this context, a *Distributed Long Running Application* (DLRA) is referred to such application that consists of a set of interactive microservices. Each functional microservice is an indispensable *component* of the application. Compared to monolithic applications, massive communications are generated and any network turbulence would coherently affect the overall response time of a given request.

Fig. 1 illustrates an example of a simple but typical e-commerce application of online store. This representative DLRA consists of nine business microservices (ranging from account related services to order management services) and seven data warehouse microservices. The arrow represents a calling relationship. After logging in the system, customers can browse the inventory through *catalogue* or add items into the *cart* before finishing an *order*. *Shipping* service will also be connected with the order service so that one can check the shipping status of a given order. All information needs to be queried and fetched from underlying database services.

B. Motivation

Response latency has been of great importance in QoS assurance for Internet services. Pinpointing microservice QoS violation is more significant because a latency in a single microservice can promptly propagate across all dependent microservices and ultimately result in the entire performance slowdown. Current cluster managers in YARN, Mesos and Fuxi are solely designed for short-running tasks of batch jobs; launching additional tasks would have negligible impact on their performance. RM is only responsible for resource allocation among applications/jobs but leave all application-specific logic to AMs. Hence, RM is application-agnostic and completely unaware of runtime QoS requirements of latency-sensitive applications.

Unawareness of application-level latency at runtime could lead to node over-saturation – too many co-located batch tasks tend to compete for resources with microservices – making neighboor microservices experience performance outliers, i.e., tail latency. These victims are vulnerable to further resource contention, and thus need particular protections: to avoid placing additional tasks onto the node or to evict running tasks to make sufficient rooms. Specifically, the system requirements encompass the following aspects:

[Q1] *How to capture the spatio-temporal variations and localize performance hotspots from DLRAs?* Since a QoS violation of a single microservice may propagate quickly and lead to cascading violations across the entire system, it is imperative to effectively trace the hotpots and extract the casualty among massive requests. **[Q2]** *How to identify the most vulnerable microservices?* At the core of this question is to find out which microservices have request backlogs and experience an increase in their latency. This helps to determine how to break down the overall e2e latency. **[Q3]** *How to mitigate the performance degradation of the victim microservices?* Node saturation is observably the main reason for increased latency. Therefore it is essential to adaptively throttle the number of back-end batch tasks that saturate the node resources or proactively delay their execution.

III. System Overview

A. Key Idea

We present a new YARN scheduling mechanism – with the help of a set of techniques including per-application-basis E2E latency tracing technique and critical path analysis – for scheduling batch tasks to adapt to dynamic DLRA's QoS status, thereby reducing QoS violation. Notably, TopoSch employs a latency-driven methodology to intervene the procedure of capacity scheduler used in native YARN.

E2E latency tracing and breakdown. In response to **[Q1]**, we measure the E2E latency from a user initiating a request to receiving the response as the performance of DLRA. In case of inter-connected microservices, the overall E2E time of a request can break down into time slices – including the time spent on average in different microservices, and the transmission time between dependent microservices. To be precise, the Mean Sojourn Time (MST) is the amount of time that a user request spends on average in each microservice; the length of MST is equal to the mean waiting time plus the mean service time. As a microservice may provide its clients multiple APIs, hundreds of thousands of requests are performed and aggregated through the API gateway before routing to specific microservices. TopoSch exploits instrumentation to trace each request and record footprints of all requests through each microservice. We can then calculate the average sojourn (processing) time on individual microservice and average transmission time. We detail them in §4.A.

Identification of vulnerable microservices via latency critical path. Critical path analysis (CPA) is the most effective means to navigate and breakdown the E2E response time. The requests in DLRA are unpredictable but traceable in a short period of time [19]. In our context, a *critical path* is referred to as the chain of invocations with the longest E2E latency across all microservices. To address **[Q2]**, TopoSch periodically constructs a request calling graph based on the microservice dependency graph and extracts components on the critical path. Those components are regarded as performance victims and have higher risks of further slowdown and failures as longer stay time of multiple requests has already been observed. This indicates a reduced suitability of co-locating other tasks. We describe the details in §4.B.

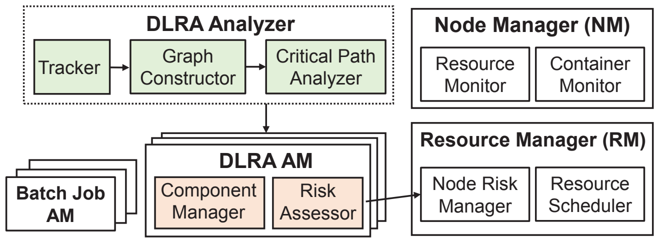

Fig. 2. Architecture overview of TopoSch

Adaptive adjustment of visible resources and the allowed number of co-located tasks in the scheduling. To cope with **[Q3]**, TopoSch recalculates and throttles the resource amount visible to YARN capacity scheduler – according to the current risk assessment on per-node basis – so that only a fraction of real available resources can be assigned to batch tasks. Equivalently, the active number of batch tasks should be controlled for adaption to the changing saturation degree of the hosting node. Relevant details are depicted in §5.

B. System Architecture

TopoSch is a YARN-based resource management system that takes into account both microservice's latency and batch job's throughput. TopoSch inherits the main modules and terminologies from YARN and employs a loose-coupled design via lightweight RPC communication. The main components are depicted in Fig. 2.

DLRA and Job Master. To align with the design of AM in YARN, we have designed a specific programming framework for launching a DLRA consisting of microservices and requesting resources from the central RM. The working mechanism is similar to the AM of DAG jobs; users can outline the topological relationships among microservices and specify resource amount in the configuration file.

DLRA Analyzer. It is the key component that can track the request footprints, and then conduct the critical path monitor by aggregating the trace data. *Tracker* is responsible for collecting massive distributed request logs generated within a certain time frame. *Graph Constructor* will read the aggregated data and build a weighted DAG that depicts the calling relationship and encompasses request's sojourn time through different microservices and the transmission latency between dependent microservices. *Critical Path Analyzer* will output the transient critical path and the pertaining microservices that contribute the most latency and are susceptible to resource saturation.

RM and NM. To enable the awareness of DLRA-level latency, the distinct departure from the default RM is that all available nodes reporting to RM are labelled and scored, primarily

TABLE I
DEFINITIONS OF IDENTIFIERS

Parameter	Meaning
url	DLRA-level API
serviceName	Microservice name in DLRA
requestID	Unique identifier of the request, i.e., UUID in DLRA
callID	Unique identifier for the given call
nextServiceID	The down-streaming microservice of the request
timestamp	Timestamp of event occurrence
eventType	Event type (i.e., send or receive)
statusCode	Event status (i.e., success or failure)

TABLE II
SYMBOL NOTATIONS

Symbols	Descriptions
t_k^i	inbound timestamp of ith request to the microservice s_k
$\hat{t_k^j}$	outbound timestamp of ith request to the microservice s_k
ST_k^i	the sojourn latency within microservice s_k
$TT_{k\to l}^j$	transmission latency of the a request j between s_k and s_l
G_{kl}	the set of requests between microservice s_k and s_l
G_k	the set of requests sent to s_k
E_k	the set of error requests sent to s_k

based on the estimation of victim microservices on it. Batch task placement can be therefore intervened when allocating resources within the RM once node's risk of performance degradation is perceptible. We inherit the main functionalities of default NM and further employ Docker containers to support the execution of microservices.

IV. Latency-aware Microservice Risk Analysis

This section discusses how to track latency footprints and investigate the key microservices at risk of latency increase.

A. End-to-end Request Latency Tracing

To obtain as many footprints as possible, we aim to record per-request and per-microservice latency. We instrument the incoming requests and output responses by tracking information including endpoints destination, inbound/outbound timestamp and request status. We design and implement a set of identifiers to depict the information of each RPC call including *url*, *requestID*, *serviceID*, *callID*, *eventType*, *nextServiceID*, *timestamp*, *statusCode* (see Table I).

We are able to infer the elapsed latency of a specific request within a microservice. Those traces will be aggregated into a centralized database, e.g., redis (`https://redis.io`), for its negligible overheads in storing and adhoc querying trace data, particularly on the occasion of periodical data update. TOPOSCH integrate *redis* with DLRA's AM to ensure effective data access whilst reducing the memory consumption of RM.

B. Finding the Longest Latency Path

The aggregated requests/responses over a period of time constitute the latency trace graph (LTG). Formally, $LTG = (\mathcal{V}, \mathcal{E}, \phi)$ comprises: a set of microservice vertices $\mathcal{V}$ and a set of edges $\mathcal{E}$ denoting the interconnection links between microservices, i.e., $\phi : \mathcal{E} \to (s_i, s_j) | (s_i, s_j) \in \mathcal{V}^2 \wedge s_i \neq s_j$ where an incidence function maps each edge to an ordered pair of distinct microservices.

There are a number of hierarchical execution entities in the system. A microservice provides multiple access points and massive requests attempt to access those RESTful APIs. A physical node can simultaneously hold multiple microservices. TOPOSCH estimates the average sojourn time per request on microservices and transmission time between microservices. We then use these timing statistics to set weights in the graph. The critical path problem will be formulated as the longest path problem in the DAG.

Request sojourn and transmission latency. t and $\hat{t}$ represent the inbound and outbound timestamp. Through the latency instrumentation and tracing, we can easily obtain the entrance/exit timestamps of a given request into a microservice.

For a given request, the sojourn latency within microservice s_k and the transmission latency of the a given request j between microservice s_k and s_l can be calculated using two adjacent timestamps (Eq. 1):

$$\begin{aligned} ST_k^i &= \hat{t_k^i} - t_k^i \\ TT_{k\to l}^j &= t_l^j - \hat{t_k^j} \end{aligned} \tag{1}$$

Latency trace graph and critical path. At the core of LTG generation is weight setting of vertice and edges. We assign the edge weight as the mean transmission latency $\overline{TT}_{k\to l}$ (Eq. 2).

$$\overline{TT}_{k\to l} = \frac{\sum_{j\in G_{kl}} (t_l^j - \hat{t_k^j})}{|G_{kl}|} \tag{2}$$

where G_{kl} is the set of requests between microservice s_k and s_l, and the size is denoted by $|G_{kl}|$. Notably, we do not differentiate the latency among different endpoints within a microservice based on the assumption of uniform RPC communication. Similarly, we assign the weight of a single vertex as the mean sojourn latency of all requests passing through the microservice s_k.

$$\overline{ST_k} = \frac{\sum_{i\in G_k} (\hat{t_k^i} - t_k^i)}{|G_k|} \tag{3}$$

where G_k is the entire request set of microservice s_k.

To implement *LTG*, we divide vertices into two distinct categories: *functional vertices* and *auxiliary vertices* to embed the sojourn latency and transmission latency, respectively. To facilitate the graph algorithms, we retain main attributes including the *service_id* and relevant microservices *upstream_id*/*downstream_id*, and the timing information. We exploit Bellman-Ford [20] for finding the longest path of LTG.

Working Example. Take an e-commerce DLRA (structured as Fig. 1) as an example. At a certain time, the dependency graph has been generated by aggregating request traces over a fixed period. As shown in Fig. 3, the values on the vertices represent the sojourn latency of request, while the values on the edges represent the transmission latency between the microservices. The longest latency path `A-B-E-G-H` will be output.

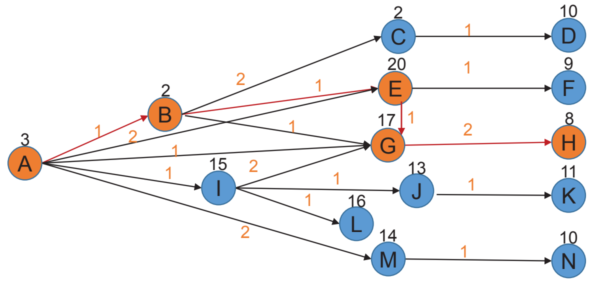

Fig. 3. An example of critical path analysis

C. *Risk Analysis of Victim Microservices*

The goal of microservices risk analysis is to identify and sort out the victim microservices. The estimation is on the premise of an important implication – victim microservices tend to exist on the critical path, since any increases in their latency will be amplified to the overall responsiveness. To distinguish further the specific risk level of those victim microservices, we take into account the following several factors:

- *Request sojourn time.* A risky microservice tends to consume longer time to process and respond to requests.
- *Request failure rate.* Higher request rate indicates a reduced reliability of request handling on the microservice. Without further resource adjustment, those microservices have higher risks of QoS violation.
- *API calling frequency.* The microservice with a higher number of requests would have a greater impact on the overall QoS. We differentiate the weight of request by recognizing the calling frequency of the pertaining DLRA-level API url.

We count the request number to the url u and calculate the proportion against the overall request number. $\omega_u = \frac{|G_u|}{\sum_{u \in URL}(|G_u|)}$ where G_u and $|G_u|$ are the request set and its size within the url u. The weight of request ω_i will have the identical weight of its url ω_u.

Specifically, the risk level r_k of a given microvervice s_k is based on the weighted sojourn proportion (WSP) over all requests and the weighted request failure proportion (WFP), as demonstrated in Eq. 4 and Eq. 5.

$$WSP_k = \frac{\sum_i \omega_i TT_k^i}{\sum_{l \in S} \sum_{j \in G_l} \omega_j TT_l^j} \tag{4}$$

$$WFP_k = \frac{|E_k|}{|G_k|} \tag{5}$$

We integrate them into the risk assessment by setting a configurable weight α, which indicates a consideration balance between sojourn latency and failure rate.

$$r_k = \alpha * WSP_k + (1 - \alpha) * WFP_k \tag{6}$$

V. Adaptive Resource Scheduling

In this section, we introduce how to maximize the probability of meeting performance requirements of DLRAs and batch jobs in scheduling.

Algorithm 1 Adaptive Scheduling Algorithm

```
Input: $\mathcal{Q}$ – task waiting queue consisting of pending batch tasks
1:  for $task$ in $\mathcal{Q}$.sort(waiting_time) do
2:    for $n$ in $\mathcal{N}$ do
3:      $R_n^{vis} = R_n^{real}(1 - \mathcal{A}_n)$
4:      if $R_n^{vis} >= task.resReq$ do
5:        $assign(task, n)$
6:        break
7:      if $locality(n, task)$ & $task.retry >= 1$ do
8:        $assign(task, n)$;
9:        break
10:     if ! $locality(n, task)$ & $task.retry >= 1$ do
11:       $randomRisk \leftarrow generate\_random\_num(0, 1)$
12:       if $\mathcal{A}_n < randomRisk$ do
13:         $assign(task, n)$
14:         break
15:   $task.retry$ += 1
```

A. *Node-level Risk Assessment of QoS Violation*

At the essence of task scheduling is to find a match between tasks that await resources and nodes with available resources. TopoSch infers the risk level of QoS violation on a per-node basis – aggregating the risk score of each microservice i.e., $R_n = \sum_{k \in G_n} r_k$, where G_n is the microservice set of node n and normalizing the overall risk level R_n (e.g., using min-max normalization) among all running nodes. Estimating the QoS violation risk is an effective means to reduce unnecessary tasks placed and co-located with victim microservices from a holistic cluster view. All node information over a fixed time frame are maintained within RM and used in resource allocation when new tasks arrives or available resources are released. RM transforms the obtained risk information into a dynamic adjustment of the amount of available resources for batch tasks and reserved resources exclusively for DLRAs. This resource elasticity ensures TopoSch can permit suitable resources to batch tasks.

B. *Task Delay Scheduling under Resource Reservation*

The most critical step is to specify the resource reservation for QoS assurance. We mainly use the overall risk assessment to calculate the resource reservation ratio $\mathcal{A}_n$. Meanwhile, we enable a balancing configuration $avoid_{factor}$ for cluster administrators to specify to what degree the proposed risk-based QoS protection mechanism is applied within the scheduling. Zero value indicates the 100% switch-off of TopoSch and the default YARN scheduler is enabled, while setting 1 means the per-node risk assessment and latency-aware scheduler is entirely activated. As shown in Eq. 7, the $avoid_{factor}$ will further tweak the reservation degree when calculating available resources.

$$\mathcal{A}_n = avoid_{factor} * R_n \tag{7}$$

Alg. 1 describes the procedure of resource allocation and task placement. We select the task from the waiting queue in a descend order by the waiting time (Line 1) and we will go through all potential nodes and filter out a node list $\mathcal{N}$ where each node has sufficient capacity to meet the task's requirement. The scheduler will calculate each node's *visible available* resource (R_n^{vis}) based on the *real available* resource (R_n^{real}) and the reservation ratio (Line 3). If the visible resource surpass the requested amount, it is safe to place this task (Line 4-6). We will delay the task scheduling for only once (by adding up the retry times) if no current visible resources are satisfactory.

Despite a round of delay, this mechanism will prioritize the QoS protection without longer delay of batch task executions. Once a task await the second time resource, TOPOSCH tries best efforts to allocate resources as soon as possible, even having to breach the resource reservation strategy for DLRAs. Specifically, tasks with data locality requirements will be directly placed onto any nodes with enough resources (Line 7-9). For tasks without locality specifications, TOPOSCH is *more likely* to place the task onto a node with lower risk level to reduce the impact of co-location on the increased latency. To achieve this, we adopt a random number based approach to implicate the tendency of choosing low risk nodes with higher probability (Line 11-14). Furthermore, to dominate DLRA's QoS, RM has the privilege to preempt and evict running batch tasks to rescue the detected QoS degradation. Jobs with lower degree of completion and jobs without data locality requirement are likely to be preempted.

VI. IMPLEMENTATION

Module Implementation. We have implemented TOPOSCH in around 3K lines of Java and integrated with YARN 3.0-Beta1.

We exemplify an implementation of DLRA AM. The topology of microservices is specified in a configuration file `DAG_SERVICE.xml`. The AM is firstly responsible for collecting request footprints in a shape of identifiers shown in Table. I and storing them into a *redis* key-value database. We provide a tracker probe package for the program in any AMs to easily record the request and response information and persist in the *redis*.

We also implement DLRA Analyzer as an independent service, which fetches data records of a given DLRA from *redis* and fulfills the core functionalities of §4. By using the programming libraries provisioned in the service, any user-defined DLRA AM can call and obtain the output of per-microservice risk level. AM instantiates an exclusive DLRA Analyzer and uses it to periodically calculates microservices' risk level at a time interval such as 60s or 120s. AM then maintains the dynamic information or microservices, their host node, and the results of risk assessment. One can follow the same steps to implement an individual DLRA and integrated with our TOPOSCH resource scheduler. We modify the RM to realize node risk assessment (§5.A) and the delay scheduling with an adaptive resource reservation (§5.B). RM obtains and aggregates per-microservice risk level from all running AMs.

Parameter Setting. Finding a suitable system parameter configuration is a non-trivial task. One common practice based on our large-scale engineering experience is to initially set conservative $avoid_{factor}$ for validation in a small-scale test system that has the same hardware configurations before deploying into larger-scale production. This procedure can significantly help towards understanding system behavior in a controlled manner. We can start from 1.0 and gradually relax the parameter to allow for more co-located batch tasks by a step of 0.1 while observing the latency variations (e.g., slowdowns or failures) through daily regression tests. This procedure can help us gradually revise the configuration with a small step until all regression tests deliver stable outputs and achieve acceptable performance level of both latency-sensitive applications and batch jobs.

TABLE III
EXPERIMENT ENVIRONMENT

Software	Kernel version	Linux version 4.15.0 -54-generic
	Release version	Ubuntu 7.4.0-1
Hardware	CPU version	Intel(R) Xeon(R) E5-2630 v3 CPU @2.4GHz
	CPU physical cores	16 (2 physical CPUs * 8 cores/CPU)
	CPU logical cores	32 (2 physical CPUs * 16 logical cores/CPU)
	Memory	125GB

VII. EXPERIMENTS

A. Experiment Setup

Workloads. We select Piggymetrics [21] benchmark as a representative DLRA and BigDataBenchmark [22] to generate batch jobs. The testbed environment is detailed in Table III.

• *Piggy Metrics.* It is a personal financial management service consisting of 12 components. Each component in Piggy Metrics is encapsulated in a docker image. We embed the instrumentation and tracing mechanisms detailed in §4.A into each component. We use JMeter [23] to generate workloads to Piggy Metrics and leverage TPC-W [24] to simulate user behaviors – *new users* register their accounts and log into the system to complete a series of transactions, whilst *old users* directly log into the system before browsing and visiting; *tourist users* are only permitted to browse some basic functionalities.

• *Batch jobs.* To differentiate workload types and data locality sensitivity, we use three batch jobs generated by BigDataBenchmark, including *WordCount* (IO-intensive with data locality), *PI calculation* (CPU-intensive without data locality) and *Kmeans* (CPU-intensive with data locality). Specifically, each *PI* job is set to has 600 mappers and each mapper contains 150 millions sampling points. Each *WordCount* job conducts an analysis of 70G of wiki textual data while *Kmeans* job performs a cluster analysis over an 8G Facebook graph data with initial number of clusters set to be 10.

Methodology. We first deployed a Piggy Metrics instance and then submitted a set of HTTP requests and batch jobs. To minimize the noise, we repeat each experiment 10 times independently and compute the average running time or performance. TOPOSCH is compared against the following baselines:

• *YARN.* The native capacity scheduler of Apache YARN used for default co-location.

• *Run-Alone.* The run-alone case where Piggy Metrics or batch jobs are independently executed without interference.

We measure the latency of Piggy Metrics to reveal the effectiveness of QoS assurance during co-location. Meanwhile,

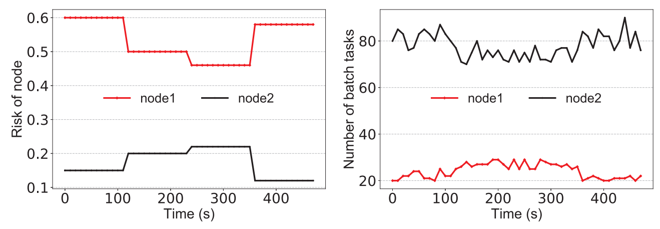

Fig. 4. The number of tasks and corresponding scores of nodes

we monitor the overall makespan of submitted batch jobs to evaluate the impact of TOPOSCH mechanisms on the performance of batch jobs.

B. Evaluation

Effectiveness of adaptive scheduling. We further demonstrate the effectiveness of TOPOSCH in QoS guarantee through functional analysis of adaptive scheduling. In this experiment, we submit mixed batch workloads during the execution of the latency-sensitive application, and then constantly observe the change of per-node risk level and the number of co-located tasks using TOPOSCH. The frequency of calculating critical path in the DLRA Analyzer is set to be 2 minutes.

Fig. 4 visualizes the changing risk evaluation of two nodes (one is the node that has the highest risk level and the other is a random node) and the resultant batch tasks that can be launched and executed on the two nodes. It is observable that the number of batch tasks that can be launched on a node is negatively correlative to the change of per-node risk level. As a result of an increase in node risk assessment, our adaptive scheduler will then reduce the resources available for new task assignments, thereby diminishing the performance interference between batch tasks and victim microservices.

Effectiveness of QoS assurance for DLRAs. Fig. 5 illustrates the results of latency when the Piggy Metrics benchmark co-exists with different batch jobs (PI, WordCount, and KMeans). Compared against native YARN, the average latency of TOPOSCH is observably reduced by 33.9%, 23.4% and 20.9% in the co-location case of PI, WordCount and KMeans jobs, respectively. Correspondingly, the 99th percentile latency is reduced by 39.8%, 35.3% and 28.2%. This is because *PI* tasks have no locality requirements and thus are likely to be placed on low-risky nodes, thereby having the most significant latency improvement. Meanwhile, our policy has to balance the execution delay and QoS assurance, i.e., tasks with locality preference will not be delayed twice and task placement will be then performed with the node risk level loosened. This results in a lower improvement in WordCount and KMeans jobs compared against the cases of colocation with PI jobs. Due to the nature of CPU-intensive, compared with the case of WordCount, KMeans will incur additional CPU overhead when co-locating with DLRAs. The increased CPU contention will slow down the coexisting DLRA, leading to a reduced degree of improvement.

Fig. 5 also depicts the detailed cumulative distributed function (CDF) results. In particular, TOPOSCH is much closer

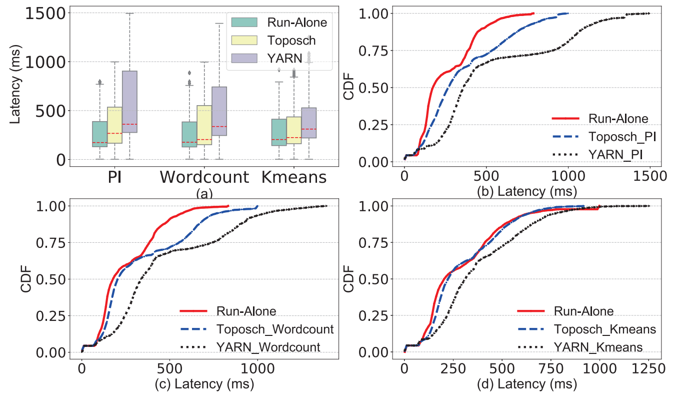

Fig. 5. Latency of Piggy Metrics when co-locating with different batch jobs

to the case of Run-alone; the latency below 300ms accounts for more than 60% of all samples. By contrast, no more than 30% samples can be observed within 300ms in YARN. This is because TOPOSCH reduces the probability of accumulating more batch tasks with the key microservices without further diminishing its QoS through risk assessment and the following adaptive scheduling. As a result, the performance interference caused by resource contention can be minimized, with the DLRA performance guaranteed.

Impact on the execution of batch jobs. Fig. 6 shows the impact of TOPOSCH on the execution time of different batch jobs. As TOPOSCH is integrated with native YARN, the value of $avoid_{factor}$ can tune the degree of adoption of the proposed scheduling based on risk assessment. We evaluate the impact of parameter setting in the adaptive scheduling algorithm on the job performance. We gradually increase the value of $avoid_{factor}$ and examine the execution time.

There is an increasing trend in the makespan of all batch jobs when the value of $avoid_{factor}$ grows. Specifically, the E2E makespan of PI jobs in the case of TOPOSCH with entire QoS assurance has increased by 53% compared against the zero case. In comparison, the Kmeans jobs and WordCount jobs experience a 24.4% and 18% increase, respectively. This increment can be regarded as the sacrifice of a given batch job for the QoS guarantee of latency-sensitive applications. Task s without data locality (such as PI tasks) can be delayed for multiple times. It is more likely to be throttled or evicted for provisioning sufficient resources for victim microservices. Tasks with data locality requirement, on the other hand, will be directly launched from the second retry for rapid task startup, even if the node is risky. This will lead to reduced pending time and makespan of Kmeans and Wordcount jobs, although the latency of the co-existing microservices is increased. This is certainly aligned with the results of QoS assurance.

C. System Overhead

We analyze a per-AM overhead from DLRA Analyzer in terms of time complexity and memory consumption. *(i) Time Consumption.* As shown in Fig. 7, the time cost linearly increases but slows down when the amount of trace data

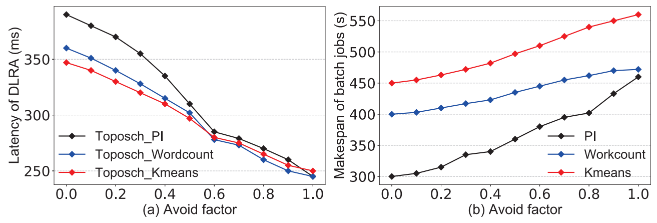

Fig. 6. The performance of both latency-sensitive applications and batch jobs

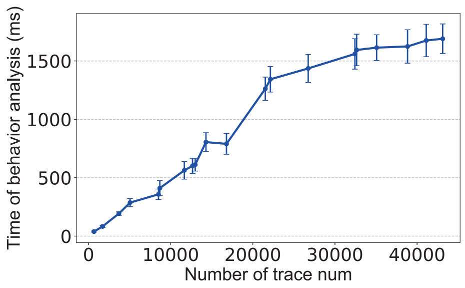

Fig. 7. Time consumption for critical path analysis

reaches 30,000. The maximal measured time is no more than 1.6 seconds. Considering the overall time consumption in the resource allocation, the incurred increase to the scheduling latency is less than 1% compared with the native YARN. *(ii) Memory Cost.* The additional memory used for fast data access using *redis* is roughly 126MB, less than 2% increase compared against native YARN. Given the intrinsic diversity in request number and arrivals, the number of traces for tracking latency in TOPOSCH over a given period can be customized in AM to balance the scheduling precision and the incurred overhead. It is worth noting that the overhead analysis is on a per-AM basis but can be naturally extended to cases of multiple DLRAs.

For cases of multiple DLRAs, memory cost will be increased by multiple times due to *redis* is instantiated to support multi-tenancy; each AM of DLRA will independently store its own request tracing information. Each AM will be encapsulated in a Docker container, and thus the AM can separately run with stringent resource isolation and negligible interference.

VIII. RELATED WORK

Scheduling for co-located workloads. The ability to co-locate jobs (i.e., execute within the same CPU or GPU) has been identified as a means to address under-utilization problem. Understanding and achieving high resource utilization or high energy efficiency for heterogeneous workloads in cloud computing is an important topic [25][26][27][9][28][29].

Existing work on effective co-location of latency-sensitive applications and batch jobs has two distinct categories: (i) reducing the probability of resource contention by either granting isolated execution environments to LRAs [30][31] or adjusting task placement to reduce the resource contention on a certain node [32][33], primarily for runtime QoS of LRA. (ii) reducing performance interference caused by resource contention through performance prediction and resource inference, prioritizing the resource requests of latency-sensitive LRAs [32][10][11][12][26][34][35][36]. However, since they are merely applicable to guarantee performance for monolithic applications, none of them can be directly adopted to resolve performance interference caused by co-location of DLRAs and batch jobs due to the tempo-spatial latency fluctuations. By contrast, TOPOSCH exploits the latency of requests across microservices to intervene the original scheduling of batch jobs and is able to infer suitable co-location degree on a per-node basis.

Performance tracing and diagnostics. Many prior work are devoted into anomaly diagnosis and behavior analysis of large-scale distributed applications. They can be classified into two categories: (i) *black-box* approaches using external application states to infer and analyze the problems. [2][37] rely on a tremendous number of log files to extract performance information and infer the dependency models. [38] trains models to predict and localize latent errors in microservices based on log information comprising a set of predefined features. [39] uses fault injections to measure the execution and data flows of distributed applications and find the bottlenecks for diagnosis. (ii) *white-box* approaches by monitoring causality within microservices instead of inferences through statistical analysis. [40][41] infer the execution path of the application based on the static analysis and symbolic execution. [42][43] provide developers with tracing frameworks to add trace-points within the application to collect runtime footprints. In comparison, TOPOSCH uses a white-box methodology to track and trace the requests over the whole DLRA and avoids over-dependencies upon prior diagnosis conditions, typically pre-defined in black-box approaches. Instead of using existing fine-grained tracking instrumentation, TOPOSCH adopts a light-weight tracking method to trace DLRA-level latency data, thereby significantly reducing per-DLRA runtime overhead.

IX. CONCLUSIONS AND FUTURE WORK

Balancing cluster utilization and applications' QoS is a non-trivial task. In this paper, we present TOPOSCH, a scheduling system to adaptively co-locate latency-sensitive applications and batch jobs. TOPOSCH periodically identifies the risk level of running microservices by monitoring and analyzing the critical path of a large number of requests and their end-to-end latency. we then propose an effective mechanism for the upcoming task placement that can prioritize and the QoS assurance of DLRAs. Scheduling of batch tasks are delayed on risky nodes, thereby reducing latency increase as a result of node over-saturation. TOPOSCH is integrated with YARN and experiments show that the latency of DLRAs can be reduced by up to 39.8% against the default capacity scheduling in YARN. Main conclusions can also be drawn as follows:

- *Tackling workload co-location plays an increasingly crucial role in resource management and job scheduling.* Improving the resource utilization and guaranteeing the QoS of running applications has become a severe dilemma which requires constant efforts to resolve. It is still challenging to realize innovations in terms of interference-aware job scheduling and fine-grained resource management in uncertain and extra-dynamic environments.

- *Application behavior analysis is an effective means to fundamentally analyze the performance of concurrent systems.* The procedure usually requires a model of application behavior that includes the causal relationships between components and node behaviors that are produced from observations of component logs.
- *It is imperative and challenging to understand an end-to-end request in a dynamic, highly-concurrent and Internet-scale distributed system.* There is a trend whereby increasing numbers of cloud-based stateful applications may overwhelm conventional batch jobs, particularly boosting the requirement for strict QoS guarantees and interference throttling.

In the future, we plan to investigate the sensitivity of different microservices to the allocated resources and fine-grained resource contentions such as CPU/LLC, so that we can further optimize the co-location number and workload type that are most suitable for co-locating in a certain node.

Acknowledgment

This work is supported by National Key Research and Development Program of China (2016YFB1000503), NSFC (61421003), the EPSRC (EP/T01461X/1), and Beijing Advanced Innovation Center for Big Data and Brain Computing (BDBC). For any correspondences, please contact Renyu Yang.

References

[1] F. Nwanganga and N. Chawla, "Using structural similarity to predict future workload behavior in the cloud," in *IEEE CLOUD*, 2019.

[2] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch, "The mystery machine: End-to-end performance analysis of large-scale internet services," in *USENIX OSDI*, 2014.

[3] Z. Wen, T. Lin *et al.*, "Ga-par: Dependable microservice orchestration framework for geo-distributed clouds," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 1, pp. 129–143, 2019.

[4] R. Yang *et al.*, "Intelligent resource scheduling at scale: a machine learning perspective," in *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*. IEEE, 2018, pp. 132–141.

[5] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *ACM EuroSys*, 2015.

[6] V. K. Vavilapalli, A. C. Murthy, C. Douglas *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *ACM SoCC*, 2013.

[7] B. Hindman, A. Konwinski, M. Zaharia *et al.*, "Mesos: A platform for fine-grained resource sharing in the data center." in *USENIX NSDI*, 2011.

[8] Z. Zhang, C. Li *et al.*, "Fuxi: a fault-tolerant resource management and job scheduling system at internet scale," in *VLDB*, 2014.

[9] X. Sun, C. Hu *et al.*, "Rose: Cluster resource scheduling via speculative over-subscription," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2018, pp. 949–960.

[10] D. Lo, L. Cheng *et al.*, "Heracles: Improving resource efficiency at scale," in *ACM ASPLOS*, 2015.

[11] Y. Sfakianakis *et al.*, "Quman: Profile-based improvement of cluster utilization," *ACM TACO*, 2018.

[12] P. Lama, S. Wang *et al.*, "Performance isolation of data-intensive scale-out applications in a multi-tenant cloud," in *IEEE IPDPS*, 2018.

[13] C. Delimitrou and C. Kozyrakis, "Quasar: resource-efficient and qos-aware cluster management," in *ACM ASPLOS*, 2014.

[14] C. Delimitrou *et al.*, "Tarcil: reconciling scheduling speed and quality in large shared clusters," in *ACM SoCC*, 2015.

[15] Y. Zhang, G. Prekas *et al.*, "History-based harvesting of spare cycles and storage in large-scale datacenters," in *USENIX OSDI*, 2016.

[16] Apache hadoop yarn 3.0.0. [Online]. Available: https://hadoop.apache.org/docs/r3.1.1/index.html

[17] Apache Tez. [Online]. Available: http://tez.apache.org/

[18] Y. Gan, Y. Zhang, and K. Hu, "Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices," in *ACM ASPLOS*, 2019.

[19] M. Fazio, A. Celesti, R. Ranjan, C. Liu, L. Chen, and M. Villari, "Open issues in scheduling microservices in the cloud," *IEEE Cloud Computing*, 2016.

[20] R. Bellman, "On a routing problem," *Quarterly of applied mathematics*, 1958.

[21] Piggymetrics. [Online]. Available: https://github.com/sqshq/PiggyMetrics

[22] W. Gao, J. Zhan, L. Wang, C. Luo, D. Zheng, X. Wen, R. Ren, C. Zheng, X. He, H. Ye *et al.*, "Bigdatabench: A scalable and unified big data and ai benchmark suite," *arXiv preprint arXiv:1802.08254*, 2018.

[23] JmeterEB/OL. [Online]. Available: https://jmeter.apache.org.

[24] Tpc-w[eb/ol]. [Online]. Available: http://www.tpc.org/tpcw/specs.asp.

[25] C. Delimitrou and C. Kozyrakis, "Quasar: resource-efficient and qos-aware cluster management," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 127–144, 2014.

[26] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *IEEE/ACM MICRO*, 2011.

[27] Q. Chen *et al.*, "Prophet: Precise qos prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers," in *ACM ASPLOS*, 2017.

[28] R. Yang, C. Hu, X. Sun, P. Garraghan, T. Wo, Z. Wen, H. Peng, J. Xu, and C. Li, "Performance-aware speculative resource oversubscription for large-scale clusters," *IEEE TPDS*, 2020.

[29] W. Xiao *et al.*, "Gandiva: Introspective cluster scheduling for deep learning," in *USENIX OSDI*, 2018.

[30] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: flexible, scalable schedulers for large compute clusters," 2013.

[31] P. Delgado, F. Dinu, A.-M. Kermarrec, and W. Zwaenepoel, "Hawk: Hybrid datacenter scheduling," in *USENIX ATC*, 2015.

[32] H. Kasture and D. Sanchez, "Ubik: efficient cache sharing with strict qos for latency-critical workloads," in *ACM ASPLOS*, 2014.

[33] P. Garefalakis, K. Karanasos, P. Pietzuch, A. Suresh, and S. Rao, "Medea: scheduling of long running applications in shared production clusters," in *ACM EuroSys*, 2018.

[34] J. Zhu *et al.*, "Perphon: A ml-based agent for workload co-location via performance prediction and resource inference," in *Proceedings of the ACM Symposium on Cloud Computing*, 2019, pp. 478–478.

[35] R. Yang, I. S. Moreno *et al.*, "An analysis of performance interference effects on energy-efficiency of virtualized cloud environments," in *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, vol. 1. IEEE, 2013, pp. 112–119.

[36] I. S. Moreno *et al.*, "Improved energy-efficiency in cloud datacenters with interference-aware virtual machine placement," in *2013 IEEE eleventh international symposium on autonomous decentralized systems (ISADS)*. IEEE, 2013, pp. 1–8.

[37] A. Pi, W. Chen, X. Zhou, and M. Ji, "Profiling distributed systems in lightweight virtualized environments with logs and resource metrics," in *ACM HPDC*, 2018.

[38] X. Zhou, X. Peng, T. Xie, and J. Sun, "Latent error prediction and fault localization for microservice applications by learning from system trace logs," in *ESEC/FSE*, 2019.

[39] C. Pham, L. Wang, B. C. Tak *et al.*, "Failure diagnosis for distributed systems using targeted fault injection," 2016.

[40] C. Zamfir and G. Candea, "Execution synthesis: a technique for automated software debugging," in *EuroSys*, 2010.

[41] D. Yuan, H. Mai, W. Xiong *et al.*, "Sherlog: error diagnosis by connecting clues from run-time logs," in *ASPLOS*, 2010.

[42] Systemtap. [Online]. Available: https://sourceware.org/systemtap/

[43] J. Mace, R. Roelke, and R. Fonseca, "Pivot tracing: Dynamic causal monitoring for distributed systems," 2018.