January 2019

Data-Driven Intelligent Scheduling For Long Running Workloads In Large-Scale Datacenters

Guoyao Xu
Wayne State University, et8130@wayne.edu

DATA-DRIVEN INTELLIGENT SCHEDULING FOR LONG RUNNING WORKLOADS IN LARGE-SCALE DATACENTERS

by

GUOYAO XU

DISSERTATION

Submitted to the Graduate School

of Wayne State University,

Detroit, Michigan

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

2019

MAJOR: COMPUTER ENGINEERING

Approved By:

_____

Advisor Date

_____

_____

_____

_____

DEDICATION

Dedicated to my loving parents, lovers, family and friends for their persistent encouragement and consistent support.

ACKNOWLEDGMENTS

The study for PhD in the past 6 years and Master for 2 years is full of pressures, adventures, frustrations, obstructions and challenges. The 8-year precious time and experiences would be an invaluable wealth in my whole life. I could not have achieved anything without help from many people. I am sincerely grateful to my great advisor, Dr. Cheng-Zhong Xu, for his invaluable guidance, support, and encouragement throughout this work. His devotement, enthusiasm, patience, persistence, abundant knowledges and experiences, rigorous logic and creative thinking for research motivated me in the Ph.D. pursuit. He devotes countless time, energy and resources to train me to become a qualified researcher and a potential scientist.

I would also like to thank Dr. Carol Miller for her support, help and guidence since 2011. I have gained countless research, engineering and industrial experiences from fields like intelligent energy efficiency and environmental protection techniques by working with her. It helps me open fields of vision and think in a macro view. I would also like to thank Dr. Song Jiang for his outstanding contributions on guiding my scientific and logical thinking during my first research project. Under Dr. Jiang's help, I overcome the most frustrated and dark time of Ph.D period and successfully enter the world of science and research. His integrity, purity, honest, smart, wisdom, rigorous logic and creative thinking have a deep influence on me. I obtained invaluable scientific way of thinking and experiences of research, experiments and paper writings. They have become my powerful weapons to conquer further challenges, and even affect my values and thinking in my daily life. I would also like to thank Dr. Weisong Shi for his guidances and supports during my master degree. With the help of Dr. Shi, I successfully and rapidly transfer my major and background from electronic engineering (EE) to computer science (CS). I learned programming, fundemental CS technique knowledges, operator and maintenance experiences of system and cluster, research, expression and presentation skills from him. He is my first mentor of CS. He helps me not

only in my academic career, but also in my life when I was new to a foreign country. It is really my great honour to work with these admired people and intelligent talents.

I am also grateful to my committee members: Dr. Nabil Sarhan, Dr. Carol Miller and Dr. Song Jiang for their time, interest, and helpful suggestions to improve this work. I would also like to thank Mr. Stephen Miller, Mr. Loch McCabe, Dr. Caisheng Wang for helps of projects. I am thankful to my colleagues and friends, Kun Wang, Xingbo Wu, Kejiang Ye, Jianqiang Ou, Yang Wang, Xi Zhang, Bo Peng, Yudi Wei, Yuehai Xu, Xiangping Bu, Dajun Lu, Yun Wang, Jie Cao, Mingyang Xu and so many friends, for their friendship and help.

Finally, I want to give special thanks to my parents and family for their understanding and unconditional support for my study during happy and hard days. They endure the same extent of negative emotions and challenges as me. Without their help and dedication, I would never finish this work.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1 INTRODUCTION

With the rapid growth of big data-driven artificial intelligence and cloud based industrial innovations, IT techniques are becoming the most critical factors to booming evolutions and future success of various industries, which is named the Fourth Industrial Revolution. Many emerging industries like autonomous cars, blockchain, fintech, new energies, robotics, quantum information, as well as traditional industries (electric power, manufacturing, medical, finance, biology, biomedical, environment, architecture, catering, retail and estates), are deeply affected by data and intelligence era. They inevitablly leverage cloud based big data and artificial intelligence techniques to improve effectiveness, study technological innovations, create new business models and replace original industry chain. For a fast rising rocket of society, diverse cloud-based big data distributed computing systems turned to be the engines. Abundant algorithms (machine learning, statistics, control algorithm, game theory...) are the boosters, while the immeasurable accumulated big data become the fuels.

Generally, big data are being processed by various computing engines running on the infrastructures of global cloud datacenters, which are becoming fundamental facilities of society. The infrastructure's role of cloud datacenter to industry is equal to the essential status as electricity to human society. The abundant hardware resources, as well as computation, storage and adatively learning abilities, are most critical for human beings towards big data-driven intelligent era. As fast evolutions of society utility, its scale would be rapidly increasing and incredibly tremendous.

Large-scale public or private cloud datacenters spreading millions of servers, as a warehouse-scale computer [53], are supporting most business of Fortune-500 companies and serving billions of users around the world. Typical infrastructures construction always involves tens of billions of capital investment, which accounts for 60% of total IT budget [53, 56, 66]. Servers purchasing dominate the total cost of ownership (TCO) (50%-70%) [53, 131]. The operating expenditures including energy cost occupy the rest. During the limited life span of 5 and 10 years for both servers and datacenters [53, 56], maximize

resource efficiency is critical to improve return on investments (ROI) and reduce TCO of infrastructures. Unfortunately, recent studies [60, 69, 74, 75, 126, 131, 135, 150, 165, 179, 185] reveal that industry-wide average utilization is as low as 6% [66] to 12% [19]. Low utilization not only negatively impacts operational and capital components of cost efficiency, but also becomes the scaling bottleneck due to the limits of electricity delivered by nearby utility. Improve resource efficiency of the large-scale fundamental infrastructure of society would bring unmeasurable values. Our society not only benefits from cost-efficiency and economic savings, but also from energy efficiency, environmental protection and emancipation of labour.

Recently, with the great commercial success of diverse big data analytics services, enterprise datacenters are evolving to host heterogeneous computation workloads including online web services [39], machine learning [40, 144, 203], streaming processing [13, 116, 176, 204], interactive query [50, 55, 168, 192] and graph computation [93, 119, 133, 138] on shared clusters. Unlike the traditional batch jobs [14, 71, 107] that rely on individual short-lived container (milliseconds to seconds) to run every task, these workloads benefit from long-lived containers (in the order of hours or months) to execute the entire application. These containers stay alive until an application's all subsequent multitier web or across-DAG-stage tasks scheduled on them are completed, so as to speedup execution by caching iterative intermediate data and avoid repeated container initialization costs. Generally, there are two types of long-running workloads. The data-intensive offline computation workloads are named long-running applications (LRAs) [86, 194, 196]. The online user-facing enterprise services like eCommerce, maps or finances that run inside non-stopped containers are named long-running services (LRSs). Observed from public enterprise cluster traces [11, 26] and analytic results [64, 127, 128, 135, 165, 185], most workloads in modern datacenters are either LRAs or LRSs.

Most previous works focus on maximizing the cluster efficiency for short-lived tasks in batch processing system like Hadoop [14,71] by optimizing scheduling algorithm (reschedule)

or achieving better resource management mechanisms (capacity planning, resource provision, auto-scaling, migration) [57,58,69,72,73,75,77,78,94–96,105,109,110,112,113,120,155,156, 160,162,163,173,177,184,185,190,196,209,210]. However, most of them are designed for past cluster scenarios and do not work well for modern long-running workloads, which have great spaces to be improved. It is critical and urgent to develop effective scheduling and resource allocation approaches to maximize efficiency in modern large-scale enterprise datacenters.

In the dissertation, we innovatively define the problems and scenarios of scheduling and resource sharing for diverse long-running workloads. As our best knowledges, we are the first of works to abstract and specify the scheduling problems and model of long-running workloads in modern datacenter. We aim to design and implement a cloud datacenter scheduling and management mechanism to maximize modern cluster resource efficiency of multi-dimensions (CPU, Memory, Network, Disk I/O) and strictly guarantee quality of services (QoS) for LRAs and LRSs.

Motivation and Background

Evolutions of Resource Management and Scheduling over Decades.

Figure 1.1 demonstrates the evolutions of scheduling problems during decades. Originally, resource management are focus on scheduling of hardware resources in local operating system or distributed system (High-performance Computing, Grid Computing, Cluster Computing). Jobs are running inside processes and the scheduling unit is process or threads.

With the emerging of virtualization and conterization techniques, we could achieve better isolations and convenient management. It significantly increases resource efficiency of both local OS and cloud datacenter. It motivates the new commercial infrastructure as cloud computing. In cloud computing era, the scheduling unit are switching to virtual machines (Hadoop). They are always as black box, and cluster operator usually does not know what workloads they are running. The scheduling optimization are mainly relying on historical runtime statistics.

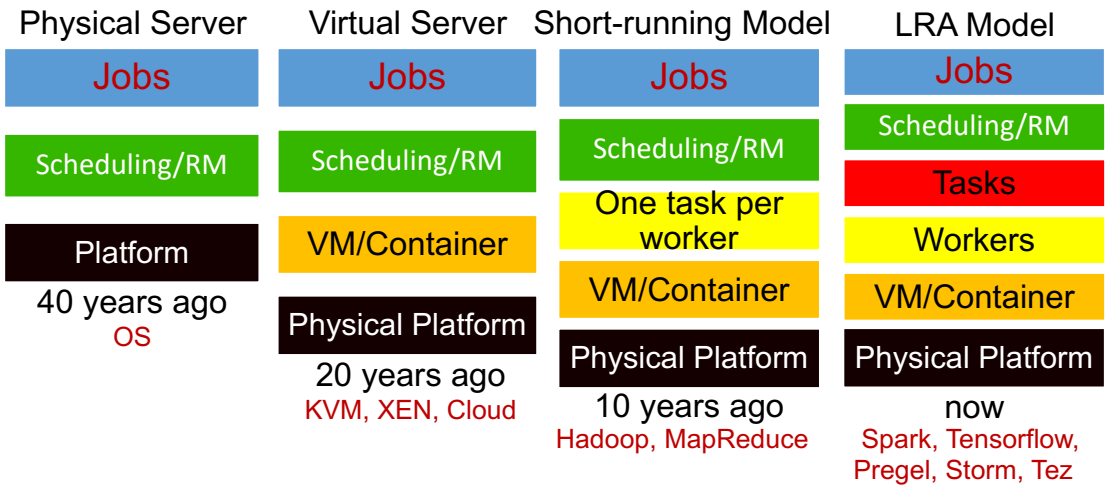


Figure 1.1: Evolution of Scheduling Problems Over Decades.

With the booming evolution of big data analytics systems and techniques, private cloud datacenter are becoming the trends adopted by most IT companies. They always hold a public cloud datacenter as products and services providing for customers, and a private cloud datacenter used for their workloads and applications (Amazon, Alibaba, Microsoft, Google, IBM...). There is not a obvious spatial gap between public and private cloud. They might locate at the same physical global datacenters, just statically segmented by different cluster managers logically.

Datacenter workloads are towards long running ones. Tasks are running inside another layer of long-lived workers, which cache intermediate data and enable multiplexed data sharing between tasks from different DAG stages, so as to significantly speedup iterative executions (Spark, Storm, Pregel, Tensoflow, GraphX, Impala or user-facing online microservices) [40, 55, 93, 138, 138, 149, 176, 203, 204]. Each long-lasting worker is always running as a process (Java virtual machine) executing multiple concurrent tasks inside a container. It helps reduce the initial costs of container startup if run one task per container as in cloud computing era. Services are packed as a complete package (microservices) running as worker processes inside containers. The scheduling subject and unit of cluster is switching to workers, and tasks scheduling are managed and handled by each application's

master. Schedulers need to pre-provision workers before task scheduling to minimize startup overheads. Therefore, it needs predictively decide how much resources each worker might need before execution, and where to dispatch those workers to avoid future potential resource contentions and interferences. Demands and interference estimation abilities are becoming Unprecedented important.

As the famous insight that "Any problem in CS can be solved by another layer of indirection, except the indirection itself" by D. Wheeler. Although virtual machine/container techniques greatly improves modern system or datacenter efficiency, it incredibly complicates resource management. Schedulers need to consider and co-tunning different layers together to guarantee smooth executions of workloads. In modern scheduling model, it requires a bundle of predictive behaviors like pre-provision, pre-reserve and QoS-aware (interference-aware) scheduling. It proposes another level requirements for more delicacy Management and refined techniques of profilings, modeling and predictions.

The booming evolution of datacenter computating engines and resource managers over last decades force us to explore predictive scheduling methodology by artificial intelligence technologies, which we named intelligent scheduling.

Challenges, Problems and Scenarios of Scheduling and Resource Management for Long-running Workloads.

Existing large-scale in-memory computing systems like Spark [203] and Flink [13] rely on a cluster resource manager like YARN [181], Mesos [103] or Borg [185] to effectively perform resources sharing between diverse LRA containers of multi-tenants in shared data-centers. Figure 1.2 exhibits Spark's two-level scheduling on YARN. It schedules the execution of tasks in a two-level model: ahead of job execution and task scheduling, Yarn creates containers for workers and pre-reserves specific amount of CPU and memory resources for them based on user's requests (step 1). Every long-lived container only belongs to one application,

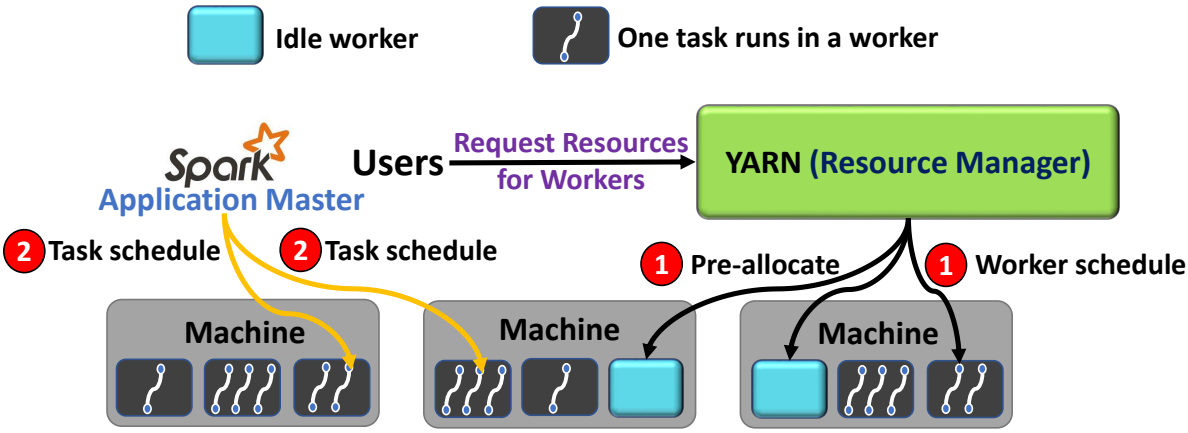


Figure 1.2: Spark's two-level scheduling upon Yarn.

and stays alive until the application's all subsequent DAG tasks scheduled on them (step 2) are completed.

An application of a Spark-like in-memory computing framework, does not expose its tasks to the underlying resource management system, like YARN and Mesos. Instead, the concept of worker is introduced as the scheduling instance in these systems. Once workers of an application are launched on servers by the systemâĂŹs scheduler, the applicationâĂŹs scheduler is responsible for scheduling its tasks to these pre-allocated workers. Specifically, an worker is usually a Java virtual machine (JVM) and tasks are threads running on the JVM. Each Spark application has a set of workers scheduled by the resource manager to different servers and they stay alive until all tasks of the application are completed. This two-level scheduling is adopted for two reasons. One is to cache a subset of data in memory to enable in-memory reuse of data across tasks in an worker in a fault-tolerant manner. The other is to significantly reduce overhead of launching tasks, which is critical for in-memory computing. In contrast, in a Hadoop application each task runs on a dedicated JVM, which is scheduled by the systemâĂŹs resource manager. While there are two levels of scheduling for in-memory computing, the workersâĂŹ scheduling plays a more performance-critical role as it represents the resources allo- cation and sharing between applications.

Traditional batch job releases short-lived container after one task terminates. Task scheduling binds to container dispatching. Job execution is along with numerous container placements. Program workflow is specific during resource reservation. Container provision is relatively easy through profiling [196], and not obtains many attentions since schedulers have numerous opportunities to reschedule and re-provision to achieve better quality.

Cluster scheduling subject is switching from tasks to workers in LRA. Worker scheduling binds to container placement, which is seperate from task scheduling that managed by application's master. Cluster schedulers need to predictively pre-reserve resources for containers before task scheduling, which always remain unchanged during long-term DAG execution. Container dispatching per application is one-time at job start, and extremely critical to ensure smooth execution of the whole application. It needs to consider long-time varying demands and colocating placements to avoid future potential contentions and interferences [196].

Tasks per stage are executed in pipeline and multiplexing in container by random scheduling. They always do not start or end at the same time. Multiple colocating tasks' combined memory usages (container's demands) in a container are highly uncertain. Meanwhile, a computation can involve many different tasks from multiple DAG stages that inhabit a container. The varying number and type of colocating tasks make containers' memory demands greatly change over time. It is non-trivial to accurately estimate just right memory reservation when task placements per container are unknown.

These schedulers solve the non-trivial allocation problem of predictably determining right amounts of memory for workers (containers) by requiring users to make explicit reservations[1] ahead of job executions. Users release the controls of their applications after submissions and know little about when their workloads to be scheduled and how they are executed (dataflow, distributed computing, interference, resource contention). Applications' time-varying demands, complex codebases or workflows are invisible to users. They

[1]Containers are being killed if their resource usages exceed reservation limits. Others could not use the underutilized reserved memory.

tend to over- or under-estimate application's demands before execution, leading to over- or under-provisioned memory reservations.

A container runs multiple batches of tasks from multi-DAG-stages, their memory demands greatly change over time [194,196]. An insufficient fixed reservation could slow down the application as much as 12 times in multiple stages or even cause job failures as shown in following section. In contrast, overprovisioning (e.g. peaks) would occupy substantial unutilized resources for a long time and leave massive pockets of resource fragmentations. Most cloud datacenters usually operate at an extremely low memory utilization state (ranging between 10% and 50%) due to these mis-provisions [53, 69, 130, 131].

To make matters worse, workers[2] of in-memory cluster computing are always Java Virtual Machine (JVM) running inside long-lived containers. Their fixed heap sizes hinder the effectiveness and feasibility of container auto-scaling mechanism [173] to improve memory efficiency. Determine a just right memory reservation for long-lived container is extremely critial to achieve both good performance and memory efficiency in in-memory computing cluster.

For traditional resource management in operating system or batch processing cluster (MapReduce) [14, 71], tasks and their runtime environment like containers or processes are scheduled at the same time. Task runtime demands are specific during resource allocation for their short-lived containers. They have numerous opportunities to dynamic adjust and incrementally re-allocate resources to achieve better quality of scheduling. Nevertheless, long-lived containers of LRAs need to be pre-deployed before task placements to minimize tail latency [70] and overheads. They require predictable reservations that always remain unchanged during future long-term execution. Tasks that satisfying data locality [182, 202] and dependency constraints are random dispatched on these distributed containers later. It is non-trivial to accurately estimate just right reservation when task placements per container are unknown beforehand.

[2]The worker is called executor in Apache Spark [15, 203] and TaskManager in Apache Flink [13].

An efficient reservation estimator provided for cluster scheduler to perform effcient resource provison is in urgent need. It is critial to accurately infer how much reservations are just right to achieve the optimal performance for LRAs and LRSs, and which LRA workers should be co-located together to minimize resource fragmentations and avoid over-allocations.

Methodology

Inherent Problems and Three Critical Abilities Towards Intelligent Scheduling and Cluster Delicacy Management to Resolve NP-hard Cluster Scheduling Problem

Modern datacenter resource scheduling is an online scheduling problem, which is NP-hard. All of existing works solve the issues by developing heuristic algorithms towards different objectives like resource efficiency [74,75,77,177,194,196], fairness [87,94–96,108,163] and quality-of-services (QoS) [46, 59, 65, 86, 99, 112, 115, 132, 142, 147, 152, 160, 193].

User first submits their jobs to the datacenter with constrained resources, and they release the control of their applications. They fully rely on schedulers to manage and execute workloads. Afterwhile, schedulers need to judge how much resources per application would need, and where to dispatch them. After they scheduled the workloads, they release the control of jobs and fully rely on local operating system (OS) to run them. However, local OS always do not total understand the concerns, constraints and objectives of schedulers. The consequence is that it always occur frequent unexpected resource contentions and intereferences between co-located applications. The runtime results are always out of the expections of both users and schedulers. So the scheduler should have the predictive ability to make allocation and placement decisions, and it should also have the ability to control OS to towards its objectives together.

There are six critical reasons leading to the difficult NP-hard scheduling problems: (1) We do not know when and what new applications would be submitted to the waiting queue.

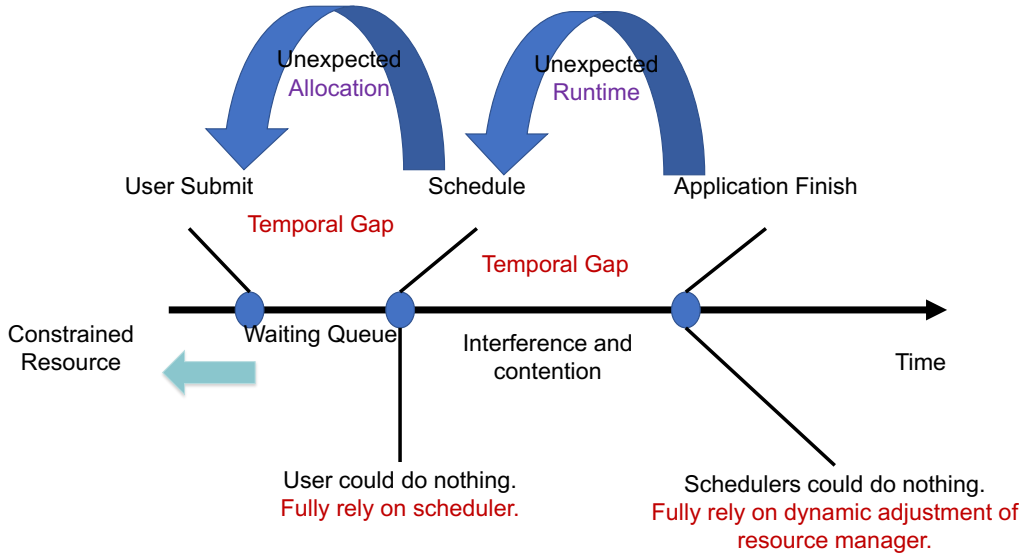


Figure 1.3: Resource management and scheduling temporal workflow.

(2) We do not know how much resources that each application in the waiting queue is just needed before execution. (3) We do not know which co-located jobs would have resource contentions and interferences with each other on the same machines, so we do not know which jobs are best to co-schedule. (4) We do not know when the running jobs would be completed after scheduled. (5) We do not know if these jobs could be normally executed or might be disturbed by some runtime exceptions like server failures, out-of-power or network faults. (6) Since the unknown knowledges of (1) - (5), we do not have a global view and optimal methodologies to resolve the online scheduling problems.

Scheduling is inherently a predictive behavior. The most critical problems of existing schedulers are lacking of predictive knowledges and inherent understanding of regular scheduling patterns and laws, which are the bottlenecks and barriers to resolve above issues. Normally, jobs are out-of-control once uers submitted them and schedulers dispatched them. Neither users or the datacenter operators have sufficient abilities to manage and control workloads at 100% percentage. Schedulers either do not have a clear judgment if their decisions are perfect or have some obvious drawbacks or trade-offs. There are not enough confidences

even in modern mature industry cluster manager like Borg [185], Fuxi [210] or Apollo [57]. We could only rely on runtime results to evalute our solutions as a feed-back control mechanism. Most of the time, we are just performing trial-and-error interaction procedures. Even prior accumulated successful experiences probably be restrictively effective to some specific domains such as High-performance Computing (HPC), Grid Computing, Cloud Computing and Datacenter Computing. When a new field like Edge Computing emerges, our previous experiences and solutions might be useless and need to repeatively develop new approaches for these traditional issues. We fall in an infinite loop during last decades in the scheduling problem field.

We need some general approaches that could be applied to any scheduling scanerio and domain. Once we could have the abundant and adequate knowledges of workloads and status of datacenter in a near future, we could have a global view and fully control of both cluster resource usages and application performance. We are also able to plan ahead to make an global optimal solution. Data-driven artificial intelligence (machine learning) are shown to be a powerful solution to fulfill above goals. It is so-called intelligent scheduling.

We listed three critical abilities that taking advantages of intelligent scheduling:

- (a) Scheduler should have the ability to predict resource-to-performance model before dispatching. The requirements to the ability includes three points: (1) Accurate profiling workloads. (2) Workloads runtime estimations. (3) Quantify the effects of interference on performance. It is critical to know how much runtime would be during interferences.

- (b) Scheduler should have the ability to know which tasks/workers are best to co-locate on the same server to achieve best utilization (operator-oriented). It needs to predict which co-located applications have the minimum interferences, resource contentions (utilization, bandwidth, capacity), fragmentations (application performance: user-oriented).

- (c) Scheduler should have the ability to predict the performance and utilization effects of dispatching new coming applications in the existing servers on running workloads. Schedulers should also predictively locate, discover and predict interferences in real time. It should also dynamic and adaptively adjust interferences in a feed-back control format. It should also automatically adjust parameters and setting of operating system such as memory reclaim ratio.

These three abilities are critical to any methodology. Most of the scheduling mechanism need and would benefit from these abilities. Schedulers could perform predictively scheduling algorithms based on estimated powerful knowledges. For example, auto-scaling needs (a) and (c) to know how much resources it would need to adjust. Rescheduling and migration need (b) and (c) to know when and where (machine and server) running tasks should migrate to, so as to minimize overheads.

Once schedulers have these three abilities, the online NP-hard scheduling problem would be transferred to a resolvable offline problems. Through the predictive knowledges, we could have a global view of near-future resource availability and usages, as well as occupied time. By given any specific objective, we are always capable of offline planning ahead and finding an optimal solution in polyomial time, and make mathematics proof as we did in offline scheduling field over last decades.

Therefore, we developed several innovated works by leveraging data-driven intelligent methodologies as a general solution to resolve the online NP-hard scheduling problems. We are on the way to rely on sufficient predictive knowledges and general prediction approaches to transfer online scheduling issues to offline scheduling to thoroughly resolve the NP-hard problem in any scheduling and resource management scenario or domain. Prophet leverages the abilities of (a)(b)(c) to perform prediction-based scheduling and develop an optimal algorithm through global view at a limited time window, so as to resolve the (3)(4)(5)(6) problems. MEER (Prometheus) employing the abilities of (a)(c) is designed to solve (2)(5)(6) issues. They make use of intelligent scheduling and resource allocations approaches. They

are great representative works to resolve the NP-hard scheduling problems for long-running applications by tranferring the online scheduling issues to offline scheduling problems through predictive knowledges.

One thing need to be mentioned. Modern artificial intelligence is named weak artificial intelligence. They could only be an automatic and efficient solution to resolve the problems that we already know. We do not impractically expect or fully rely on them to resolve the problems that we do not figure out or understand yet. Above three abilities and six issues are the inherent problems and essenses that we concluded and abstracted from past experiences of decades. We leverage artificial intelligent approaches to perform accurate predictions for specific motivations. That's why Prophet and MEER work effectively. Intelligent scheduling is only worked as an efficient solution. The problem solver subject is we domain experts. How effectively we exploit artificial intelligence techniques mainly depends on how much we understand the problems and what innovated directions and ideas we could think out to resolve problems.

Data-driven Intelligent Scheduling Methodology.

Most of existing works either rely on pure black-box solution or specific white-box-based prediction. Black-box approaches are general, at the cost of long-term exploration, model adjustment and training. It does not need to figure out the causality of the problem. White-box solutions are always rapidly effective if we study and understand deeply about origins and critical issue of problem. We could model the issue by mathematics method. However, different problems such as capacity planning, scheduling, and runtime migration, reschedule or preemption in distinct contexts (long-running workloads, virtual machine, HPC..) are toally different. We need to design and model distinct problems mannually one by one even for different systems (batch: Hadoop [14], stream: Storm [116, 176]/Flink [13], graph: Pregel [138], ML: Tensorflow [40]) and workloads under the same context. We detail

introduces how modern works leverage prediction or other intelligent approach to perform scheduling in Section .

We developed Prophet annd MEER (Prometheus) to employ above three abilities to perform intelligent scheduling as a good demonstration. They are both designed to resolve the allocation and scheduling problems for long-running workloads in modern datacenter. Prophet predicts resource usages and runtime duration per stage in future execution for both waitting workers in the queue and running workers on the server. It then has the abiliti to know future resource availability of the cluster, as well as network and disk I/O bandwidth usages and capacity. They could prevent over-allocation and perform predictively I/O contention- and interference-aware intelligent scheduling through prediction approaches.

The prediction technique of Prophet is fully rely on pure black-box estimations of SVM and logistic regression methods through historical runtime statistics. It works well since most modern datacenter big data analytics workloads are recurring. Morever, all tasks from every DAG stage per application execute the exactly same codes and similar amounts of data. It enables the accurate predictions for repeative tasks by effective profilings. They naturally has the trend to be accurately predicted. Prophet leverages (a)(b))(c) of above three abilities to perform predictive scheduling.

MEER (Prometheus) is mainly designed to resolve the optimal memory provision and reservation problem for long-running workloads. Optimal provision is always motivated to find the just right resource allocations (minimal necessary ones) that could achieve optimal performance. We found this problem is different from intelligent scheduling issue itself since every workload has a unique resource reservation-to-performance model. It is impractical and infeasible to build black-box models for each one. Moreover, since any new-submitted, non-recurring or varied workload (variations of input datasets, source code tweaks or parameters tuning) lacks of aggregated datasets and offline training opportunities, they could not benefit from the black-box based provisioning solutions and would lose the fundamental ability and opportunity to achieve best efficiency.

Additionally, we find a general and critical property for long-running workloads about the provision elasticity. That is LRAs could execute perfectly even under sizable reducations of resource reservations. Based on this property, we are able to find a inflection of point (knee) of the resource-to-performance curve as the optimal reservation that achieves both best efficiency and optimal runtime. Under this property, any type (graph, streaming, batch, ml, ad-hoc query) of LRAs could be modeling by a specific white-box approach and a feedback control loop to adaptively refine runtime predictions. This property we found for LRAs enables the white-box approach could be generally suitable for any type of workload. MEER performs estimation only based on the collected footprint data through two pilot runs without historical knowledges. It utilizes statistical methods by histogram analysis, expectation and confidence intervals of usages to effectively find the near-optimal allocation.

Through a runtime adjustment mechanism to find the knee of the curve (like stochastic gradient descent), we are able to continously provide accurate prediction of optimal reservations in a short recurring executions. MEER leverages (a)(c) of above three abilities to perform estimations for right provisioning.

Figure 1.4 illustrates how Prophet and MEER (Prometheus) support intelligent scheduling by predictively network and disk I/O contention-aware scheduling and optimal memory reservation estimations. Through this two systems, LRAs could be efficiently scheduled achieving both the best cluster efficiency and optimal performance. These three critical abilities are as three-steps towards eventual intelligent scheduling.

Dynamic Operating Systems Management and Control.

Modern datacenter infrascttructure technique is far more complicated than scheduling and resource management, it also involves in the intelligent control over underlying operating system (OS). There is a gap between cluster scheduler and local OS. OS does not fully understand the objectives, constraints and consideration of upper schedulers. Scheduler could also not dynamic adjust OS mechanism (dynamic memory reclaim, CPU throttle, memory

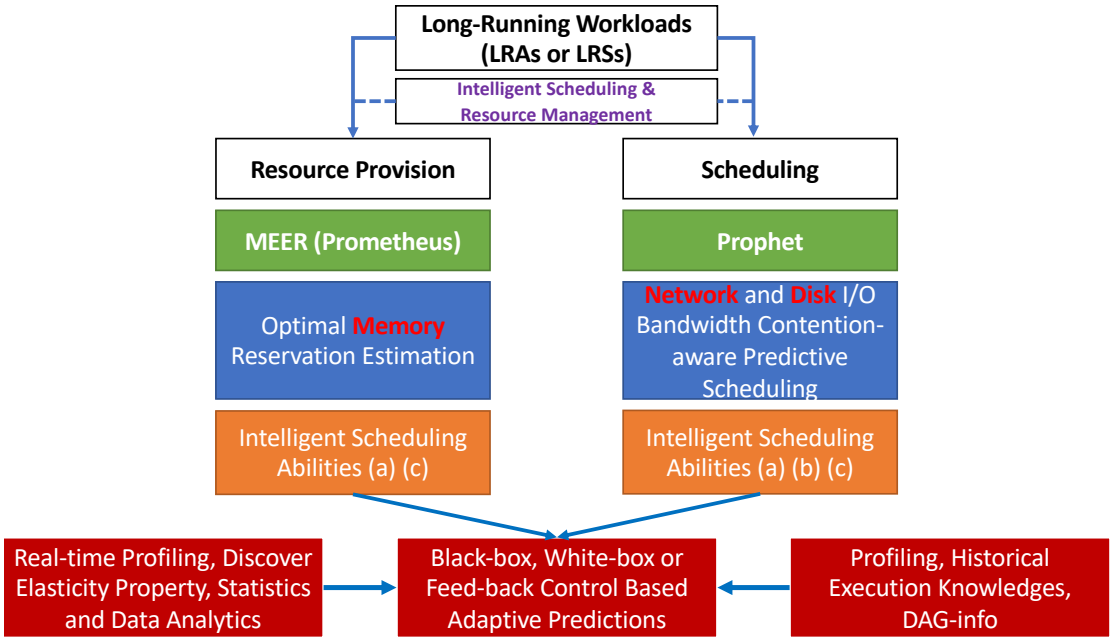


Figure 1.4: Intelligent Scheduling for LRAs or LRSs by Prophet and MEER (Prometheus).

and network bandwidth control, LLC control by CAT...) during runtime when contentions or unexpected interference occur. Schedulers lose the control of workload executions after placements. Current schedulers lack of the ability (c) of intelligent scheduling.

Therefore, we are on the way to abstract another layer of management for local OS, to provide adequate predictive knowledges to support upper intelligent scheduler. It also helps schedulers to effectively manage local OS to fulfill their objectives during runtime to strictly ensure end-to-end objectives achievement. This new layer between local OS and cluster schedulers is significantly critical to enable fully control by schedulers between jobs submitted till job completions.

We detail introduced this new layer of infrastructure in Section as part of an enterprise large-scale datacenter co-location techniques of infrastructure in Alibaba. We are on the way to thoroughly fulfill this layer of this infrastructure.

Optimal Resource Provisioning for Long-running Workloads (Memory)

In general, the relationship between resource allocation and performance for any workload is inherent non-linear and not easy to model as shown in previous works [58, 162], which have been studied for decades. Nevertheless, for long-running workloads, we found a critical inflection point between resource and performance curve. we discover an important elasticity property that could help us to stably achieve best efficiency and optimal performance at the same time.

Most recent works of capacity planning only consider resource usages instead of its relationship with performance. In our observations, containers of in-memory computing workloads would trigger modest minor garbage collections (GCs) and execute spilling-to-disk operations to effectively reclaim spaces and release memory pressures under insufficient allocation during load spikes or peak usages. Unlike the observations for short-lived batch tasks [82,106,150,151,187], we found the performance of these containers are not sensitive to sizable reductions of reserved memory. It is because the increased GC and spill overheads due to some tasks are only a negligible portion compared to the application's long-time executions. A moderate decrease of reservation would have little negative impacts on performance, but significantly reduces the wastes of excessive memory usages by 5 to 10 times.

That is, containers are allowed to run with significantly less memory reservation than they would ideally need (peak or average usages) while only paying a moderate performance penalty. We refer to this property as *memory reservation elasticity*. It is a general property for diverse types of in-memory computing workloads. Consequently, applications are able to achieve nearly optimal performance under a minimum reservation that is just large enough to satisfy their majorities of base demands. We name these capacities as *optimal reservations*. They are capacity cut lines to divide the reservation between over-provisioning and under-

provisioning. It also provides opportunities to well balance the trade-offs between memory efficiency and performance.

For recurring applications (periodically run repeated jobs on same or similar newly arriving data), the procedure to explore optimal reservations is essentially a search process over correlated memory allocation and performance. Recent studies [42,57,60,69,83,110,112, 196] revealed that a majority of production applications exhibit recurring execution pattern. The recurrent nature enables the feasibility for searching methods through multiple offline runs. However, recurring jobs usually encounter variations of input datasets, source code tweaks or training parameters tuning. Their optimal reservations would change over time under such variabilities. They need to start all over new searches.

Existing offline searching solutions do not fully address the challenges. For example, Elastisizer [101], Ernest [183], CherryPick [45] and Clipper [68] offline trained performance models based on historical executions to search best cloud configurations for recurring jobs. If simply adopting these model-based searching approaches, we need to retrain models for every type of variability, which is too expensive and impractical for tens of thousands jobs in an enterprise cluster.

Estimation of optimal memory reservations for newly submitted or non-recurring jobs is even more challenging. Due to lack of relevant runtime history information, searching algorithms become ineffective. Online executing a plenty of profiling runs for searching is simply too time-consuming and the results would become useless for online scheduling. Datacenter schedulers are in urgent need for an efficient online estimation approach for optimal reservations.

Through experiement, we found a reservation size that just accommodates the majorities of base demands is the optimal one, which is just large enough without unnecessary wastes. The mathematical expectation of memory usages that implies the average demand at future executions, are accurate enough to represent base demands. This rationale offers

opportunities to find optimal reservation of containers on line in one step, by performing probability density analysis and estimating expectation of footprints in profiling runs.

Due to random scheduling, pipelined executions and staggered aggregate memory usages of concurrent tasks on containers, the profiled footprints of long-lived containers display randomness and its expectation lacks of ability for generalization. In comparison, we observed memory usage patterns per task are highly stable and often in a Gaussian distribution. Since the usages per container is the aggregate ones of its all co-locating tasks, we could obtain robust confidence intervals of containers' random footprints by aggregating stable confidence intervals of task memory usages. By using confidence levels for the predictions, we eliminate negative effects of footprints' randomness on estimations.

We concluded our contributions as followings:

- To our best knowledge, we are the first to specify, identify and demonstrate a general property of memory reservation elasticity for diverse types of workloads in in-memory computing. We studied its origins by performing quantitative analysis of benchmarks' memory footprints.

- We discovered a long-tail relationship between reservation and performance derived from elasticity, and revealed there exists an optimal reservation for long-lived containers in terms of maximum memory efficiency and optimal application performance.

- We quantified the relationship between footprints and optimal reservation, and demonstrated optimal reservations are able to be efficiently predicted by estimating expectations of memory usages. We use confidence intervals to resolve randomness of container footprints. Through robust profilings in pilot runs, we perform highly accurate (over 80%) initial estimations through only one-step without runtime history. By exploiting self-decay property and recursive search, the accuracy was improved to over 95%. MEER should be the first kind of systems to predict optimal memory reservations.

- We implemented MEER as an extension to YARN. It efficiently assisted YARN to online make optimal reservations and speed up diverse types of in-memory computations by 2 to 6 times on average, while improving cluster memory utilization by 40%.

Contention and Interference-Aware Co-Scheduling (Co-locating) for Long-running Workloads (Network and Disk I/O)

Applications running on today's large-scale data-parallel processing frameworks, such as Apache Hadoop [14], Dryad [107] and Spark [203], usually have DAG of stages in their execution durations, such as map and reduce stages. Each stage consists of a number of tasks conducting the same type of data processing. A task requires multiple resources for its running, including CPU, memory, as well as disk and network bandwidths. While tasks belonging to the same stage are of similar demands on each of the resources, those belonging to different stages can have very different demands on different resources. For example, machine learning applications, such as K-Means and KVM (Support Vector Machine) [34, 123], tasks of their map stage are I/O- and CPU-intensive while tasks of in the following reduce stage are only network-intensive. The frameworks, such as Hadoop and Spark, usually run on the compute platforms, such as YARN [181] and Mesos [103], responsible for resource allocation and sharing. It is critical for task schedulers on the platforms to efficiently schedule the tasks of vastly diverse multi-resource demands onto a cluster of servers, so that both applications' execution time and the cluster's throughout can be maximized.

Scheduling tasks with multi-resource demands onto servers of limited amount of resources (CPU, memory, disk, and network) is often formulated as a multidimensional bin packing problem. As long as these demands are known a priori or can be accurately estimated, this problem can be solved heuristically in a polynomial time [94]. A common technique used for this estimation is to profiling tasks by leveraging the fact that jobs of an application are recurring and they âĂIJrepeat hourly (or daily) to do the same computation on newly arriving data [94]âĂİ.

Such a profiling strategy is not sufficient to fully address the issue by itself in practice, as the demands measured during a taskâĂŹs run vary (sometimes dramatically). While it is known that a multidimensional bin packing problem is NP-hard and has to be solved with heuristics, it is almost impossible to accommodate time-varying demands into the model to efficiently produce an effective scheduling decision. A conservative alternative is to use peak usage of a resource to represent the varying demands of a task during running to prevent resource over-allocation [94]], which occurs when aggregate demand from all running tasks exceeds available resources. It often leads to interference between tasks and serious performance degradation. However, this conservative approach generates risk of resource fragmentation, which occurs when resources are idle but tasks with demands on them that ready for scheduling cannot use them.

To achieve high scheduling efficiency, a scheduler has to simultaneously minimize fragmentation and over-allocation of resources [8]. When each application can have a large number of tasks and each task has a relatively short execution time, using peak demand may not create extremely large pockets of fragmentation in terms of wasted resource time. However, this becomes a serious issue with in-memory computing frame- works, such as Spark and Storm, where scheduling units have long execution time with varying demands.

Long-running workloads' workers pre-reserve specific amount of CPU and memory resources as described in Section . However, the network and disk bandwidth could not be fully isolated. Long-running workloads need a scheduler that could schedule workers by considering the contentions mitigation of network and disk I/O bandwidth. Only few previous works [94] consider network and disk bandwidth isolation for short-lived tasks in Hadoop. Recent work like Tetris [94] exploits the knowledge of future (peak) resource demands of tasks to perform bin-packing algorithm. However, it cannot be applied directly on workersâĂŹ scheduling. When an worker becomes the scheduling object, the rationale made by existing schedulers based on peak resource usage to represent the objectâĂŹs varying resource demand is less likely to be valid. A worker runs multiple batches of tasks belonging

to different DAG stages may have (very) different resource demands. Therefore, using the peak demand to represent different demands of a resource during the lifetime of an executor for resource allocation can cause serious resource fragmentation (or wastage).

Additionally, for a smooth run of tasks in an executor without interference from other application executors, it might be desired to have all four major required resources (CPU, memory, disk, and network) pre-allocated or reserved. Users only need to pre-specify their resource demands on CPU (num- ber of cores) and memory (size of memory) for a worker. As these demands usually represent the bottom line of a userâĂŹs requirement on quality of service, the requested resources are pre-reserved at the time of executor scheduling. However, network and disk resources are shared among workers on a server without isolation or reservation. They are more likely to incur over-allocation, and tend to cause disk seeks or network incast that may significantly compromise systemâĂŹs throughput. In addition, neither users nor current cluster managers [181,185] would specify network and disk demands of workers, let alone consider their highly variable demands. This may lead to application performance degradation and poor resource efficiency.

To improve cluster efficiency and speed up individual applicationsâĂŹ performance for in-memory computation, we design an executor scheduler, namely Prophet, which can select an executor whose scheduling would result in the smallest amount of fragmentation and over-allocation of network and disk resources. With the knowledge of an executorâĂŹs future varying (peak) disk and network demands at any stage during its lifetime and of each stageâĂŹs start time and its duration, Prophet can estimate resource availability at any time frame in the near future and make an informed scheduling decision accordingly to minimize resource fragmentation and over-allocation. To deal with unexpected resource contention, Prophet selects task(s) in an executor to back off to adaptively ameliorate the contention.

In summary, we make the following contributions in the paper.

- We identify a performance-critical issue about the ex- ecutor scheduling on in-memory data parallel computing platforms. We show that without considering resource demand

variation within an executor, one can hardly enable an effective scheduling. By showing stability and predicability of resource demands in an executor, we make it possible to take the dynamics on the resource demands into account.

- We design an online executor scheduler, named Prophet, that adopts a greedy approach by choosing the currently optimal executors in terms of expected resource fragmentation and over-allocation to dispatch. It also dynamically avoids severe resource contention and subsequent dra- matic performance degradation due to unexpected over-allocation with its task backoff mechanism.

- We have implemented Prophet on YARN and Spark 1.5 to support Spark and evaluated it on a 16-server cluster. Experiments show that Prophet can minimize resource fragmentation while avoiding over-allocation. It can substantially improve cluster resource utilization, minimize application makespan, and speed up application completion time. Compared to YarnâĂŹs default capacity and fair schedulers, Prophet reduces the makespan of workloads in SparkBench [4] by 39

Datacenter Co-locations Techniques for Long-running Workloads at Large-scale

Besides resource scheduling for LRAs or LRSs as shown in Section and Section , a more effective way to improve resource efficiency in modern datacenter is to co-locate both offline batch jobs, LRAs and LRSs in the unified shared infrastructure. However, it has another higher level of technique requirements from IDC, network, server hardwares, storage, virtulization and containerization, co-locating scheduling and so on. We introduced the background, motivation, feasibility and technique details of an enterprise co-location techniques at large-scale in Alibaba Group, which daily scheduling tens of millions of diverse heterogeneous workloads across tens of datacenters and millions of servers. We mainly introduce how to eliminate or mitigate runtime interferences once contention occurs.

The resource wastes and inefficiency issues are becoming serious in modern internet companies like Alibaba Group. The total utilization was even lower than 9% during several past years. It is due to the over-provisioning of online business services on the dedicated cluster to guarantee service stabilities. However, recent resource management and scheduling studies of datacenters mainly focus on supporting offline workloads of big data processing system, which are data-intensive and naturally have a high average cluster utilization over 60%. Most daily business applications of industry are online long-running services (LRSs) consisting of a long chain of multiple middleware components like web services, memcache, RPC (Remote Procedure Call) and databases. They are sensitive to tiny abnormality of operating system or datacenter network due to resource contentions or faults. One unexpected fluctuation of system load could block multiple online processes on that server. It causes large-scale time out and failures of downstream services, which become unbearable catastrophes in production environment. How to effectively schedule and run online LRSs to satisfy their strict requirements of Service Level Agreements (SLAs) while maximizing resource efficiency should obtain more attentions.

Recent studies [170,185] reveal that co-locate online services and offline data-intensive workloads on the shared infrastructure is feasible to improve efficiency while guarantee stabilities of LRSs. It brings numerous extra challenges and much higher requirements to datacenter techniques. In this paper, we introduce a large-scale enterprise-wide colocation techniques as our solutions to effectively improve datacenter efficiency. It involves in the upgradation and evolutions of full technique stack of infrastructures across the entire group, including layers of idc, network, server hardwares and architecture, operating system, resource manager and scheduler, containerization and applications.

We concluded contributions of the paper as followings:

- We introduce the evolvement of challenges and architectures of Alibaba including scenes of "Double 11" in the past years, as well as the detail motivations and techniques of large-scale colocations. The diverse workloads of both LRSs and offline jobs including

eCommerce, finance, maps and navigation, digital entertainment and social media are unique. We also introduce the workload characteristics, resource management and scheduling scenes of both online LRSs, real-time applications and offline data-intensive workloads in Alibaba global datacenters and infrastructures.

- We are the first to introduce the characteristics and resource management of online LRSs in industry rather than the offline data-intensive long-running applications (LRAs) described in recent works [86, 126, 194, 196]. Since online LRSs across 7000+ types of services from 60+ departments in entire Alibaba Group are all containerized, we have one of the largest scale of LRSs scenes in the world.

- We are the first to introduce how to effectively co-locate our unique and diverse online LRSs and offline data-intensive workloads in the large-scale shared industry datacenter in detail. Compared with Borg and Omega [170, 185], we introduce the colocation technique from the view of workload resource management and scheduling instead of technique architectures.

- We believe the unique large-scale industry scenes in Alibaba Group could be a great help to the research field of datacenter architecture and resource management. We would also continuously provide updated open cluster trace of colocations [38] to encourage more studies in this field. It includes diverse online LRSs and data-intensive workloads running on co-located production cluster of 4000 machines for 8 days [38]. It includes abundant resource usages data of machines and containers, and runtime statistics of workloads with DAG information. It could be a good verification of Alibaba co-located datacenter.

We show the colocation techniques could improve utilization from originallly 10% averagely to stably 50%, and guarantee strict SLAs and stabilities for diverse services. It saves billions of costs of TCO for Alibaba and becomes the core infrastructure technique for the next-generation architecture of Alibaba.

Evaluate the Effectiveness of Long-running Workloads Co-location Techniques by Analysis of Alibaba Datacenter Trace

To evaluate and demonstrate the feasibility and effectiveness of co-location techniques between diverse LRAs, LRSs and other short-running applications on improving cluster efficiecny at large, we perform a deep analysis on a newly released trace dataset by Alibaba Group in September 2017, consists of detail statistics of 11089 online service jobs and 12951 batch jobs co-locating on 1300 machines over 12 hours. We reveal several critical insights for co-location techniques of long-running workloads at large-scale production cluster.

Alibaba Cloud is one of the largest public cloud platforms in the world, on which processing millions of tasks acrossing hundreds of data centers everyday. This trace includes runtime statistics of a hybrid cluster, on which online service and offline batch jobs are co-locating. As we know, it is the unique one having hybrid runtime information among all public traces.

To the best of our knowledge, this is one of the first work to analyze the public Alibaba trace. We explored runtime status of the hybrid cluster, and showed several important insights about **imbalanced utilization** and **resource inefficiency** in the cloud.

Our analysis reveals several important insights about different types of *imbalance* and *resource inefficiency* in the Alibaba cloud. Such imbalances exacerbate the complexity and challenge of cloud resource management, which might incur severe wastes of resources and low cluster utilization. 1) Spatial Imbalance: heterogeneous resource utilizations across machines and workloads. 2) Temporal Imbalance: greatly time-varying resource usages per workload and machine. 3) Imbalanced proportion of multi-dimensional resources (CPU and memory) utilization per workload. 4) Imbalanced multi-resource demands between online service and offline batch jobs. Additionally, the trace demonstrated that Alibaba cluster is operating at extremely low utilizations for online services (less than 10% CPU and 45% memory average utilizations). We believe accomodating such imbalances during resource

allocation is critical to improve cluster efficiency, and will motivate the emergence of new resource managers and schedulers.

They are listed as followings:

- Spatial Imbalance: heterogeneous resource utilization across machines and workloads.
- Temporal Imbalance: greatly time-varying resource usages per workload and machine.
- Imbalanced proportion of multi-dimensional resources (CPU and memory) utilization per workload.
- Imbalanced resource demands and runtime statistics (duration and task number) between online service and offline batch jobs.

Many modern resource managers are designed under the assumption of ideal cluster environment. The commonly occurred imbalance phenomenons in Alibaba trace would lead to significant resource inefficiency and wastes. We believe it is critical to accomodate such imbalances during resource allocation to improve cluster efficiency. They will also motivate the emergences of new resource managers and schedulers.

By analysis of the public cluster trace, we demonstrate the large-scale co-location technique gains a great success to improve average utilization of Alibaba global datacenter from 10% to 50% averagely, and guarantee the strict SLAs of LRSs.

Dissertation Organization

The rest of this dissertation is organized as follows:

Chapter 2 gives an overview on existing approaches of resource management and scheduling in distinct contexts over recent decades. We compared the differences between previous schedulers designed for short-lived tasks or virtual machine dispatch, and scheduling for long-running workloads (LRAs and LRSs).

In Chapter 3, we introduce our optimal reservation estimation system named MEER (Prometheus). It could rapidly and efficiently make just right memory reservation for LRAs

through two pilot runs without historical knowledges. It performs predictions by profiling, histogram analysis, confidence interval infer of task and worker memory footprints. It achieves the maximal cost-efficiency, resource effieicny and optimal application performance at the same time for LRAs.

In Chapter 4, we present our worker scheduler named Prophet. It performs bin-packing-like algorithms to achieve minimal fragmentations and over-allocations by accommodating long-lived time-varying demands of LRA workers. It leverages machine learning based predictable techniques to know workers' network and disk bandwidth usages ahead, as well as stage durations of LRAs. By performing a time-space packing algorithm, it could guarantee best efficiency at the expenses of little ignorable contentions.

In Chapter 5, we study the large-scale co-location techniques of long-running workloads to make huge improvements on efficiency in large-scale datacenters. We introduce Alibaba LRAs and LRSs workload characteristics at a full map. We also illustrate motivation, feasibility, challenges, as well as effective elastic resource sharing and isolation techniques for co-locations in Alibaba datacenter. We improve average datacenter utilization from 10% to averagely 50% due to this efficient technique.

In Chapter 6, we demonstrate our evaluation results of large-scale co-location techniques for long-running workloads by analyzing Alibaba public cluter traces. We are the first work to illustrate the co-location effectiveness of Alibaba global datacenters, and attract a lot of attentions.

Chapter 7 concludes this dissertation with summaries of our contributions, methodologies and directions for future work.

CHAPTER 2 RELATED WORK

Resource management and scheduling have been studied for decades ranging from various contexts including High-Performance Computing (HPC), Cluster Computing, Grid Computing and Cloud Computing. We compare the relevant works of schedulers in large-scale datacenter.

Figure 2.2 demonstrates the evoluation of scheduling and resource management techniques over decades. They varies along with the evoluation of big data distributed computation systems in cloud datacenter as shown in Figure 2.1.

Towards Intelligent Scheduling

There have been many studies on task scheduling on the data-parallel computing platforms. There are also studies on leveraging history resource usage to predict future resource demands for improved data locality and execution efficiency. In addition, studies on virtual machine placement and migration are also related on the aspect that Spark's executors are actually Java virtual machines. In the below we show how our works are related to previous works and why it represents a unique contribution.

Cluster Schedulers: The issue of task scheduling in large-scale data-parallel systems has been extensively studied recently [71, 107, 203]. Quincy and delay schedulers are designed to improve data locality of individual task while maintain fairness of different applications [108, 202]. Both Fair and Capacity Scheduler [27, 28] are Yarn's default schedulers [181] designed for slot-based resource allocation. They conduct tasks for high scalability and fairness. Dominant Resource Fairness (DRF) utilizes max-min fairness to maximize the minimum dominant share for all users when allocating multiple resources [87]. Implementations of DRF or earlier schedulers only consider CPU and memory in their resource allocation. Tetris is the first task scheduler that packs tasks based on multiple resource demands including CPU, memory, network, and disk to avoid resource over-allocation and minimize fragmentation [94]. However, none of existing works are designed for scheduling executors of

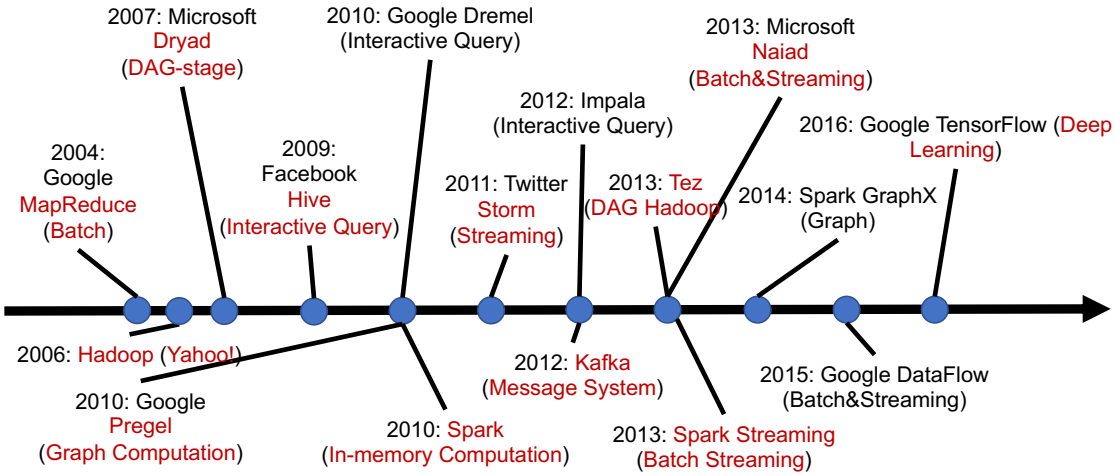


Figure 2.1: 15-years Booming Evolution of Datacenter Computation Engines.

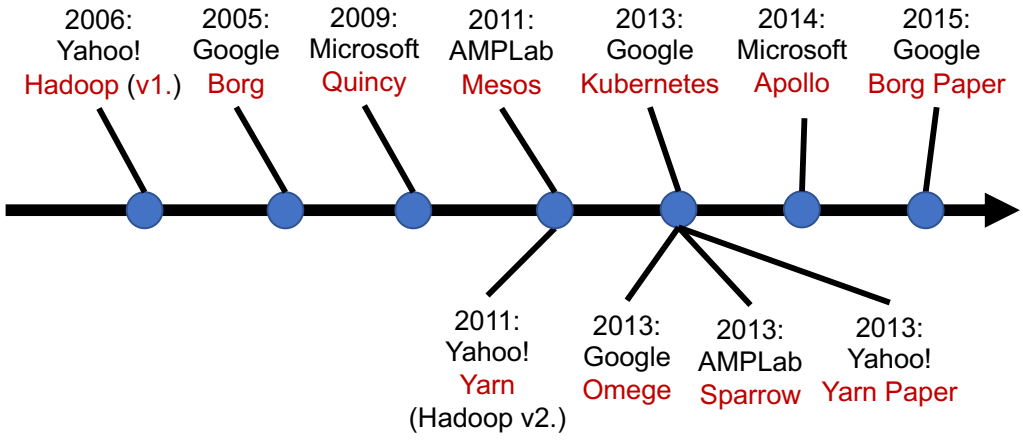


Figure 2.2: 15-years Booming Evolution of Resource Scheduling in Datacenter.

Spark applications. While each executor has many stages with possibly distinctly different resource demands, their assumption on stables demands in a task does not hold. We show that scheduling executors with existing task scheduler would incur serious disk and network fragmentation and over-allocation, leading to poor makespan, application completion time, and low resource utilization.

Plan-ahead scheduler techniques: While there are a large number of recurring applications in a large-scale data-parallel production environment, it is known that future resource demands and execution times are predictable and there optimization techniques take advantage this observation to plan ahead and accordingly make scheduling decision. Apollo estimates task execution time from historical task runtime statistics to perform estimation-based task delay scheduling for improved data locality and reduced task completion time [57]. Tetris estimates tasks peak multi-resource demand from previous runtime statistics to conduct multi-dimensional task-packing scheduling [94]. Both Jockey and ARIA predict the completion time of a running application through past execution profiles and a control loop estimating application's progress, to automatically adjust resource allocation to meet application's SLO [83, 184]. Corral predicts future application arrival time and characteristics such as input and intermediate data sizes from recurring application statistics, to jointly coordinate input data placement with task placement to improve data locality and reduce cross-rack network data transfer [110]. While these works show that prediction on application behaviors based its history can be highly effective for informed scheduling, none of these works apply the technique for scheduling of Spark's executors. While in-memory computing platforms, including Spark, are very popular, the efficient scheduling based on prediction is on high demand, and Prophet makes a timely contribution.

VM Packing/Schedulers: To some extent the issue of virtual machine (VM) scheduling is similar to the executor scheduling. Both need to consider demands of multiple resources and both will host multiple processes or tasks to run. Their actual resource demands can also be highly variable. Both need to avoid resource over-allocation and frag-

mentation. There have a number of works on VM scheduling. Among them, AutoControl packs virtual machines with multi-resource demands using dynamic feedback-based VM fine-grained resource allocation [155]. Sandpiper migrates VM to alleviate overload condition to maximize resource utilization [191]. It does not adopt predictive plan-ahead packing placement strategy because future resource demands of the VM's Web-based or interactive applications are highly unpredictable. However, neither of those approaches could be applied to scheduling of Spark executors, because it can be too expensive to (frequently) migrate often data-intensive executors and incur transferring of large volume of data.

Current works solve the problem through different aspects. Modern schedulers frameworks such as YARN [181], Mesos [103], Omega [170] and Borg [185] adopt centralized or distributed scheduling with diverse algorithm motivated to improve data locality [202], packing efficiency [94],fairness and capacity [108].This facilitates vastly concurrent sharing between diverse types of applications ranging from batch jobs to long running services.

Optimal Provisioning By Intelligent Scheduling

As a core technique in resource management of large-scale datacenter, predictions of performance or resource demands have been widely studied in various contexts for a long time. A large number of cluster schedulers [57, 69, 72, 73, 75, 77, 78, 94–96, 105, 109, 110, 112, 113, 120, 155, 156, 160, 163, 177, 184, 185, 190, 196, 209, 210] leverage the knowledge of future resource availability to perform intelligent scheduling, so as to meet SLOs while achieving maximum utilization. Other works rely on predictions to speedup execution [42,83,164,183], assist fault detection [199] or mitigate stragglers [47–49,197] and interferences [74,99]. They are not aware of the memory reservation elasticity and optimal reservation of long-lived containers in in-memory computing systems like Spark. MEER is an dedicated prediction technique to these in-memory computing workloads, which is complementary to previous works.

The others that rely on resource-to-performance modeling to support SLO [83, 112, 184] and tunning configurations [45, 101, 102, 160, 183] are complementary to MEER. Morpheus [112], PerfOrator [160], ARIA [184] and Jockey [83] leverage telemetry of historical runs to estimate SLO and corresponding skyline of demands. They dynamically adjust allocations in order to meet deadlines. Elastisizer [101], Ernest [183], CherryPick [45], Clipper [68] and BestConfig [211] monitor resource usages and use profiling or historical traces to search optimal configurations. Paragon [74] and Quarsar [75] employ classification techniques and historic runtime data to perform online co-locating, so as to avoid interferences. PerfOrator [160], Graphene [96] and Apollo [57] make predictions relying on white-box modeling by analyzing sizes of task inputs in disk and reproducing parallelism, which ignores irregular sizes of in-memory data and uncertain memory demands. Most of these works build prediction models offline for short-lived tasks of MapReduce based on massive profiling runs or historical executions statistics of recurring applications. Their methodologies cannot provide an efficient online estimation of optimal memory reservation for newly submitted or non-recurring in-memory computing workloads, and are too time-consuming for online scheduling. Additionally, optimal reservations of recurring applications tend to change over time with variations of input datasets or algorithmic parameters. Their offline model-based searching is ineffective. Cloud online schedulers need MEER to perform accurate online prediction from only two pilot runs and to handle newly submitted applications as well as the variations of recurring runs.

Under above near-monotonic trend, the procedure to dis- cover optimal memory demands for recurring jobs has been transferred to an online search problem. It seems like we could naively adopt brute-force or random search used in [101, 102, 201] that randomly chooses initial allocation and continuously reserves diminishing memory with a fixed step size. We would find the runtime inflection point beyond which performance starts to drastically degrade. However, if the chosen starting point is far away from the optimal demands or the step size is improperly small, it would take hundreds of executions to reach the des-

tination. Such long-time search means extremely high overheads and becomes impractical for online configura- tion. In contrast, adopting an improperly large step size might cause estimations of large errors.

Other Bayesian Optimization [45] or machine learning based [160, 183] search needs multiple profiling runs or prior historical execution statistics. They may suffer inaccurately cold-start for new submissions due to lack of relevant history. Further, these methods are designed to explore best instance configuration within a given space of discrete candidate choices. They are not applicable to online search of specific targets among such boundless spaces and consecutively varied candidates of memory capacities. Their loss functions might take long-time recurring executions to converge with high search overheads.

Moreover, all above searching methods are only effective for recurring jobs without variabilities. It is time-consuming and infeasible to perform multiple runs for newly submitted or non-recurring jobs to online finding of the optimal memory demands. The recurring applicationsâĂŹ knees of memory versus performance curves also change over time along with varia- tions of input datasets, parameters or source code. It becomes too expensive and infeasible to start over new online recursive searches for every variability.

In this paper, we develop an online histogram frequency analysis algorithm to efficiently infer preliminary optimal memory demands through only one pilot (profiling) run. It could be applied to newly submitted or non-recurring jobâĂŹs worker demands estimation, with a high accuracy of more than 80%. Its benefits come from analysis and modeling of the frequency of past runtime memory footprints per time unit under unconstrained memory reservation. We could distinguish base demands and unnecessarily excessive memory usages [106] under such wasteful over-provisioning. Allocation of base demands tend to achieve near-optimal performance, so as to approach optimal demands.

The histogram analysis algorithm has an intrinsic property of self-decay. We exploit this property to recursively perform searching during subsequent recurring executions. It obtains stepwise refinement and rapidly approaches to a near-optimal estimation (over 92%

accuracy). We demonstrate this online recursive search method outperforms alternative solutions from both accuracy and overheads.

Other works that rely on resource-to-performance modeling to support SLO [83, 112, 184] tuning configurations [45, 101, 102, 160, 183] and perform intelligent scheduling [74, 75] are complementary to Prometheus. Morpheus [112], PerfOrator [160], ARIA [184] and Jockey [83] leverage telemetry of historical runs to derive SLO and corresponding skyline of demand. They dynamically adjust allocations in order to meet deadlines. Elastisizer [101], SLARM [164], Ernest [183] and CherryPick [45] monitor resource usages and use profiling or historical traces to search optimal configurations. Paragon [74] and Quarsar [75] employ classification techniques and historic performance data to perform online scheduling, so as to avoid inferences.

Most of the previous works build models for tasks based on multiple profiling runs or historical executions statistics of recurring applications. These methodologies cannot be directly applied to PrometheusâĂŹs in-memory computation scenarios because the optimal memory demands for workers of recurring jobs tend to change over time with variability of input datasets or algorithmic parameters. We need Prometheus, performing accurate online estimations from one pilot run, to handle these variabilities and newly submitted applications.

LRA and LRS Scheduling in Large-scale Datacenter

Schedulers for Colocations. Recent studies propose approaches that colocate latency-critical (LC) and batch applications to maximize efficiency [57, 74, 75, 86, 90, 122, 131, 185]. Borg and its open source version of Kubernetes [51, 185] are the first enterprise-wide scheduler for colocations, which daily dispatching millions of batch and long-running applications (LRAs). Bistro [90] introduces a hierarchical of data and computational resources to enable resource constaints for online cluster and efficient parallel scheduling for offline workloads. They prefer effective colocations rather than optimal allocation objectives to

guarantee strict SLAs for LC services as in Sigma. Quasar [75] and Paragon [74] employ classification techniques and historic performance data to perform online scheduling, so as to avoid interferences. Medea [86] relies on two types of schedulers to make global optimal decisions and satisfy expressive constraints for LRAs, while guaranteeing low latency for batch tasks. Apollo [57] uses task-duration estimation and opportunistic executions of best-effort tasks to plan ahead and boost utilization. None of them are studying scheduling for LC LRAs during extremly load spikes as in Alibaba. We are facing more challenging industry issues of scarce resources and low latency requirements.

Deadline-Aware Scheduling and Resource Efficiency. Other works rely on resource-to-performance modeling and capacity planning to allocate right resources and tuning configurations, so as to catch deadlines and support SLO. Rayon [69] declares a reservation-definition language and formalizes planning of future resources as a Mixed-Integer Linear Programming for batch production jobs on YARN to catch deadlines. TetriSched [177] leverages runtime estimation and deadline information of to perform space-time-aware global allocation for data analytics applications upon Rayon. HCloud [77] employs hybrid provisioning of on-demand and reservation to handle sensitive services and insensitive batch jobs, so as to maximize efficiency while ensure SLA for LC services. Morpheus [112], PerfOrator [160], ARIA [184] and Jockey [83] leverage telemetry of historical runs to derive SLO and corresponding skyline of demand. Prophet [196] leverages historical profilings to predict future I/O demands and make efficient packing for batch LRAs. They also dynamically adjust allocations as in CloudScale [173] to meet deadlines. However, most of these works are designed for offline analytics products instead of user-facing LC services. They are not aware of extreme load spikes of production services either.

Offline Batch Scheduling. Since big data analytics workloads become increasingly popular, the underlying resource management gain a lot of attentions. Quincy [108] and Firmament [92] regard batch task scheduling as graph and network flow model to provide fairness and data locality-aware scheduling. Delay Scheduling [202] proactively delays alloca-

tions to achieve better data locality while guarantee fairness through round-robin allocation. YARN [181], Mesos [103] and Fuxi [210] are designed to support diverse analytics workloads such as batch [71, 168, 203], machine learning [40, 144], graph computation [93, 138], ad-hoc queries [50, 55] and stream processing [116, 176, 204]. They adopt reservation and admission-control based two-level scheduling to incrementally allocate resources for tasks. They are focus on throughputs and makespan instead of strict SLA and unpredictable load spikes for online production services.

Altruistic Schedulers and Efficiency. In multi-tenants cluster environment, fairness becomes increasingly important to guarantee reasonable shares of multi-resources and prevent starvations. DRF [87] adopts an economics algorithm to satisfy max-min fairness including sharing incentive, strategy-proofness, envy-freeness and pareto efficient for multi-dimension resources. Choosy [88] is an evolution version of DRF that takes resource constraints such as placement locations and hardwars into account. Tetris [96], Prophet [196], Graphene [96] and Carbyne [95] extends the fairness of CPU and memory in DRF to I/O bandwidth. They designed a comprehensive scheduling approach to simultaneously satisfy fairness, packing efficiency and minimal slow down. These works are designed for short-lived batch containers and are not friendly to LRAs since shortest job first (SJF) algorithm always blocks long applications. Sigma prefers load balances to ensure SLAs for LC services, which against the skew-preferred mechanisms to resolve above bin-packing scenarios.

Distributed Schedulers. Due to modern sub-second task runtime requirement and large-scale cluster scale, centralized schedulers could not achieve low scheduling latency of millseconds level. Distributed schedulers start to take over the cluster [57, 113, 122, 153, 170]. Omega [170] employs pessimistic lock to resolve the conflicting decisions between distributed share-state schedulers. Sparrow [153] make opportunistic distributed allocations based on sample profilings. Mercury [113] and Hawk [73] provide rich resource management API based on a hybrid design of central and distributed schedulers to balance the trade-offs between execution and scheduling efficiency. Tarcil [122] leverages sample-based statistical

approach on loads to allocate resources for long and short jobs, while reconcil scheduling speed and quality. These works focus on architecture design of distributed schedulers rather than resolving the non-trivial allocation and placement issues for LRAs as in Sigma.

Data-Driven Intelligent Schedulers. Recent works try to leverage deep reinforcement learning (RL) [139] to adaptively train schedulers and allocate resources. They rely on runtime rewards inference and benefit from accurate performance estimation. It is designed for batch analytics jobs. Sigma innovatively uses RL to pack LC LRAs under constraints during extreme load spikes.

Large-scale Datacenter Trace Analysis

Google released a 29-day trace of over 25 million tasks across 12,500 heterogeneous machines in 2011 [37]. There are several important works on analyzing Google trace from different perspectives. Zhang et al., focused on characterizing run-time task resource usages of CPU, memory and disk [206]. Reiss et al., characterized cluster resource requests, distributions, and the actual resource utilizations. They found heterogeneity and dynamics are two important characteristics. [167] [166]. Liu et al., characterized how the machines in cluster are managed and when the workloads submitted during a 29-day period behave. They focus on the frequency and pattern of machine maintenance events, job and task-level workload behaviors, and how the overall cluster resources are utilized [129]. Abdul-Rahman et al., considered user behaviors in composing applications from the perspective of topology, maximum requested computational resources, and types of workloads [41]. Sharma et al., focused on the task placement constraints in Google compute cluster and developed methodologies for incorporating task placement constraints and machine properties into performance benchmarks of large compute clusters [172]. Di et al., compared the differences between a Google data center and other Grid/HPC systems, focus on loads of jobs and machines [80].

While other works use machine learning method, such as k-means clustering, to study the workload characteristics. Mishra et al., described an approach to workload classification

based on k-means and its application to the Google Cloud Backend [145]. Di et al., computed the valuable statistics about task events and resource utilization for Google applications, based on various types of resources (such as CPU, memory) and execution types (e.g., whether they can run batch tasks or not). They also classified applications via a K-means clustering algorithm with optimized number of sets, based on task events and resource usage [79]. Chen et al., identified common groups of jobs by k-means clustering. They also did correlation analysis between job semantics and job behavior, leading to helpful perspectives on capacity planning and system tuning [63].

While our work is one of the first analysis on Alibaba trace, which is released in September 2017. Furthermore, we analyze this dataset from a new perspective and find several interesting *imbalance* phenomena in the cloud.

CHAPTER 3 MEER: Online Estimation of Optimal Memory Reservations for Long Lived Containers in In-Memory Cluster Computing

Introduction

Modern in-memory computing systems like Spark create long-lived containers to execute diverse types of applications. They rely on a cluster manager like YARN or Mesos to perform resource allocation to the containers. The cluster manager or scheduler requires users of the containers to reserve resources beforehand. It is a challenge to estimate just right amounts of memory to run the applications before execution, so as to avoid over- or under-provisioning of memory space. We discover a general property of *memory reservation elasticity*, which allows applications to run with a reservation limit smaller than they would ideally need while only paying a moderate performance penalty. Based on the property, we designed a system, namely MEER, which performs online estimation of minimum necessary amount of memory limit that achieves nearly optimal performance. We referred to it as *optimal reservation*, which divides memory over-provisioning from under-provisioning.

It is non-trivial to efficiently estimate optimal reservations on line through one step without runtime history. MEER uses a two-step approach to dealing with the challenge: 1) Do robust profiling and probability density analysis of applications' memory footprints in two pilot runs. By using confidence levels for the predictions, we reduce the negative effects of container footprints' randomness and achieve a highly accurate online initial estimation (over 80% accuracy) of optimal reservation. 2) By exploiting a self-decay property of the analytical results, MEER adaptively performs recursive search based on a feed-back control mechanism over subsequent recurring executions. We implemented MEER atop of YARN and evaluated the prototype by running 15 benchmark workloads on a 16-node local cluster. Evaluation results show that it achieves an average accuracy of more than 95%. By deploying MEER on schedulers and allocating memory according to the optimal reservations, one could improve

Table 3.1: Experiment configurations of 12 workloads

Benchmarks	Input Datasize	Benchmarks	Input Datasize
Terasort	93GB	PCA	184.3GB
SVM	75GB	PageRank	15GB
KMeans	87.5GB	SVD++	8GB
LogisticRegression	147GB	ConnectedComponent	8GB
LinearRegression	191.6GB	TriangleCount	8GB
DecisionTree	95.8GB	TPC-DS Query 7	100GB

cluster memory utilization by about 40%. It reduces individual application execution time by 2 to 6 times on average compared to the state-of-the-art approaches. A 90 times peak speedup for PageRank in comparison with the default Spark/Yarn is observed.

MEER could be integrated with most in-memory cluster computation frameworks, without much effort of abstraction modification. We evaluate MEER on a local 16-server cluster, and compare it to state-of-the-art memory demands estimation systems. We demonstrate MEER effectively avoids over- and under-provisioning of memory, simultaneously optimize application performance while maximizing memory efficiency.

The rest of the paper is organized as follows. Section demonstrates memory elasticity and its origins. It also identifies optimal reservation. Section illustrates the feasibility of estimation and challenges due to footprints' randomness. Section describes the design and implementation of MEER and the robust two-step prediction approach. Section describes the evaluation. Section **??** reviews the related works and Section concludes the paper.

Motivation

Balancing the trade-offs between effective resource usages and optimal application performance is a core challenge in the management of datacenters [75,112,160,196]. Essence of the issue is to determine the right size of memory reservation. The problem is especially important for containers due to the long-term static reservations [86,196].

To illustrate the relationship between distinct reservations and performance variations, we ran 12 representative Spark benchmark workloads as in Table 3.1 on a 16-server

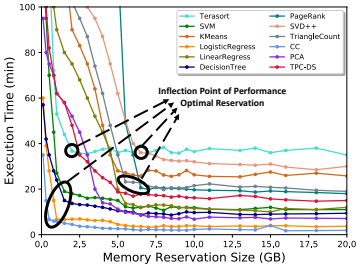


Figure 3.1: Long-tail variations of application runtime under degressive reservations.

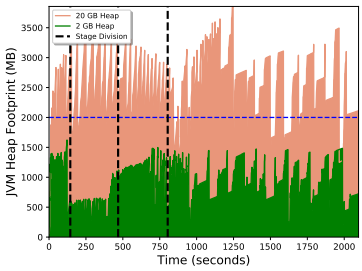


Figure 3.2: Memory footprints of Terasort under optimal (2 GB) and over-provisioned (20 GB) reservations.

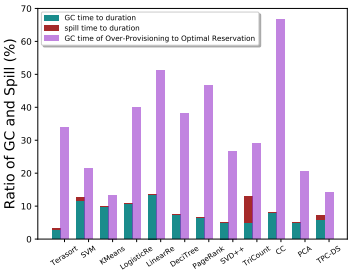


Figure 3.3: The ratio of aggregate GC and spill time to runtime due to optimal reservation and over-provisioning.

local cluster, most belonging to the industrial SparkBench and TPC-DS suite [34, 35, 123, 124, 159]. The detailed experiment settings are described in Section .

These diverse workloads range from graph computation (PageRank, SVD++, Connected Component, Triangle Count), machine learning (SVM, KMeans, Logistic and Linear Regression, Decision Tree, PCA), SQL-based query (TPC-DS) and batch processing (Terasort).

For every workload, we continuously changed memory reservation sizes of workers for 30 times to observe corresponding performance variations in the experiment. The cluster was dedicated to run each application for a total of 10 times per reservation size. Each application is set to deploy one worker per server to maximize data locality [182, 202].

We used Spark 2.0.2, Hadoop/Yarn 2.7.2 and OpenJDK-1.8.0-amd64. Other settings such as parallelism (5 cores per worker), input data size, algorithm parameters and program codes are fixed. We obtain average completion time of ten-times runs per application.

Figure 3.1 presents the execution time of different applications under various memory reservations. From the figure, it can be seen that there always exists an inflection point of performance (knee) in applications' long-tail reservation versus runtime curves. When the reservation of memory is less than a boundary, like 7 GB for PageRank and 2 GB for Terasort, a sharp drop of 4 (Terasort) to 20 times (PageRank) performance degradation and even program failure were observed. The steep decline of performance was caused

by thrashing under critical shortages of memory. The frequent long and useless garbage collections (LUGC) [82] are continuously triggered during runs, leaving little time for task execution. These long-lived containers are unable to effectively release memory in time, and ultimately crash because of JVM out-of-memory errors.

After exiting the crash region, the performance seldom fluctuates even under tremendous over-provisioning. It means that diverse applications are not very sensitive to the change of memory reservation. The average runtime variance is around 5% while largest is no more than 10% in the case of KMeans. It implies that when the majorities of base memory demands are satisfied, they have little impact on runtime due to the shifting of bottleneck resource types. These kinds of over-provisioned reservations waste resources without distinguished runtime rewards.

By effectively spilling of data into secondary storage and triggering minor GCs under moderately insufficient memory, applications only pay moderate performance penalty under considerable reductions of reservations. It is the memory **reservation elasticity** that offers opportunities to achieve maximum memory efficiency and optimal performance at the same time. The minimal reservation size of the tail is the boundary to divide over-provisioning from under-provisioning. These desired configurations are named **optimal reservations**. Its allocation would achieve the runtime inflection. Figure 3.1 reveals that the turning point is predictable from multiple runs. As accumulating of recurring executions, we are capable of recursively searching optimal reservations that just occur before performance dramatically decreases.

Why elasticity exists: inspiration by footprint. From Figure 3.1, we can see the runtime of over-provisioned applications differ little from the one of optimal reservation. The largest 10% variance is insignificantly small and might be caused by systematic errors. Applications only pay marginal performance penalty under elastic reductions in memory reservations. Long-lived containers are able to effectively mitigate memory pressures by

triggering modest GCs and spilling data into disk during load spikes, so as to avoid runtime degradation.

To better understand elastic and optimal reservation, we compared the memory footprints due to optimal and over-provisioned reservation for diverse workloads. Figure 3.2 gives the footprints in the execution of Terasort under optimal reservation and over-provisioning. From the figure, we can observe that the average heap utilization is as high as over 80% under optimal reservation (It is more than 75% in other workloads). Stages are segmented by time in black dotted lines. Compared to the average of 17.5% and less than 15% utilization due to over-provisioning, wastes of underutilized memory were significantly reduced.

Additionally, peak demands of Terasort (over 3200MB) only occupied a small fraction of 3% during execution, which was significantly higher than the average usage μ of 2356 MB. The standard deviation σ of footprint was as little as 547 MB, indicating the major usages were located within an interval around μ, from 1809 MB ($\mu - \sigma$) to 2903 MB ($\mu + \sigma$). The observation is identical for all workloads. Simply adopting unnecessary high reservation to satisfy peak demands is wasteful and unworthy.

When the peak demands could not be satisfied under a small heap limit, workers trigger moderate minor GCs and spilling operations to effectively reclaim resources and reduce pressures. Figure 3.3 gives the results about the aggregate GC and spill percentage of total runtime, and the ratio of GC overheads due to optimal reservation and over-provisioning. From the figure, we can observe the aggregate GC time of over-provisioning is less than 30% of the ones under optimal reservation for most workloads. Especially in the cases of KMeans, TPC-DS and SVM, the GC overheads of over-provisioning were only 13%, 15% and 20% of the ones under optimal runs respectively.

Despite the GC and spill increase the overheads, their rare trigger frequencies led to an insignificantly small fraction of total execution time due to the rare peak demands (3%). Under optimal reservation, the largest GC percentages were only 13%, 12% and 10% in the cases of Logistic Regression, SVM and KMeans, respectively compared to application

runtime, while the percentages of spills were as little as 1%, 9% and 2% of SVM, Triangle Count and TPC-DS. Not every workload relied on spilling-to-disk operations to mitigate pressures. Different from YARN-ME [106], we showed spills that lead to sawtooth-like shape of memory elasticity for short-lived MapReduce tasks are not effective for long-lived containers. In addition, unlike the common observations that GC contributes at least 50% of execution time in short lived containers of a MapReduce system [82,89,91,136,137,150,151], we found the increased overheads caused by GCs and spills are only a negligible percentage for long-lived containers in in-memory computation when their major demands are satisfied.

Motivation. Our motivation is to efficiently estimate optimal reservation size for long-lived containers, and make schedulers reserve memory accordingly before the applications get into crash zone. It achieves nearly optimal performance and minimizes unnecessary memory wastes.

Rationale of Methodology

It is time-consuming and infeasible to execute multiple runs for newly submitted or non-recurring jobs to online find the optimal memory reservations. Under the long-tail curves of Figure 3.1, the procedure to find optimal memory reservations for recurring jobs has been transformed to an online search problem. We could simply use random search as in [101, 102, 201] to randomly select initial allocation, and continuously reserve decreasing memory with a fixed step size like random gradient descent. However, if the chosen starting point is far away from the optimal reservations or the step size is improperly small, it would take hundreds of executions and search to reach the target. These expensive overheads of long-time search per application are unaffordable and impractical for online scheduling. In contrast, adopting an improperly large step size would make the result inaccurately far from the optimal reservation. An effective online estimation methodology is in urgent need.

Other bayesian optimization [45] or pure machine learning based [68, 160, 183] search needs a large number of profiling runs and prior historical statistics. They suffer from cold-

start and become ineffective for new or non-recurrent submissions due to lack of relevant history information. Moreover, the recurring applications' knees of memory limit versus performance curves also change over time along with variations of input datasets, parameters or source code. It becomes too expensive and infeasible to retrain models and start over new searches for every type of variability per application. An effective online estimation methodology is in urgent need.

In-memory computing systems like Spark are designed to process iterative workloads. Most Spark workloads consists of repetitive stages with same operations. The memory footprints of these representative stages could effectively reflect the total usages per application.

We summarize memory usages of representative iterative stages per workload under over-provisioned reservations of motivation experiments in Table 3.2[3]. These statistics are the average ones of 30 executions and memory limits are recommanded by the benchmark. It only displays the stage with maximum mean usages among all repetitive ones. It could accommodate other stages and reflect the maximum requirements of an application. Each number is an average one of all long-lived containers.

We found the optimal reservation size per representative stage of over-provisioning is very close to its mean usages. The pattern is similar for all workloads, which quantifies the relationship between footprints and optimal reservation. The average usage of past executions implies the mathematics expectation of future footprints, that is the probable average base memory demand at arbitrary future time. By reserving significantly less amount of memory limit than the peak and even average demands, applications could still achieve near-optimal performance due to elasticity. All other iterative stages show consistent trends. This rationale offers opportunities to predict optimal reservation through one step, by estimating an application's major base demands.

However, simply relying on profiled average usages to estimate base demands and optimal reservations would lead to high inaccuracy. The large prediction errors of most

[3]R, MU, OR, PU (%) and StDev indicate reservation size, mean memory usages per stage, optimal reservation, peak usages (percentage of frequency) and standard deviation of usages respectively.

Table 3.2: Container memory usages summary per stage

Benchmarks	R (GB)	MU (GB)	OR (GB)	PU (GB) (%)	StDev (GB)
Terasort	18	3.2	2	8.2 (4.2%)	2.8
SVM	18	4.2	3	8.4 (5.1%)	3.2
KMeans	18	6.5	5	11.2 (2.4%)	7.1
LogisRegre	8	1.2	0.7	7.3 (3.2%)	2.0
LinearRegre	8	4.2	3	7.6 (4.5%)	4.5
DecisionTree	9	3.9	3	8.2 (2%)	3.5
PCA	18	7.8	6	13.7 (5.3%)	8.1
PageRank	20	8.2	7	17.8 (8.2%)	11.2
SVD++	18	9.4	7	15.3 (6.2%)	8.6
CC	8	1.8	1	3.6 (8%)	1.95
TriangleCount	18	4.5	3	6.7 (3.3%)	4.6
TPC-DS	18	4.7	3.5	9.5 (4.2%)	3.8

workloads in Table 3.2 are about 30% to 45%. The root causes are followings: (1) We observed the standard deviations of stage footprints are large and even more than mean usages. It indicates container footprints are always non-uniform distribution and drastically fluctuating. Additionally, peak usages are usually 2 to 6 times higher than average ones, while these peak intervals are only a small proportion of less than 8%. The unstable average usages of containers' profiled footprints are far from representing their expectations and expected base demands. We need to perform probability density estimation of footprints.

(2) Spark tasks are scheduled on long-lived containers based on data locality [182,202]. The co-locating tasks placement per container are random. Despite concurrent tasks of all containers belong to the same stage with consistent operations, their memory usage behaviors might be different due to task data skew or variances of shuffled input sizes. Since the aggregate footprint per container is the sum of its all co-locating tasks' memory usages, they would be distinct for every container within the same stage. They are also random under repeated recurrent executions, as well as the average memory usage.

Additionally, task durations within a stage significantly varied. They are in pipelined execution and always do not start or end at the same time. The footprint per container consists of tasks' aggregate staggered usages would be highly random. The randomness of footprint per container makes its average usages ineffective to infer base demands. Robust profilng and effective analysis of random footprints are non-trivial. How to leverage pro-

filings of tasks' footprints to subtly handle the randomness and robustly predict optimal reservation? We answer these in the following section.

Design of MEER

We developed an online estimation system MEER, which employs a combination of techniques including robust profilings, histogram analysis, self-decay prediction and recursive search, to provide an accurate and stepwise refined online optimal reservation prediction.

Modeling and Robust Profilings

From Section , we know runtime memory usages per time unit t of every long-lived container j and individual task i are two sets of random variabile x_j^t and y_i^t. We define their conditional probabilities under arbitrary reservation size r as $P(x_j^t|r)$ and $P(y_i^t|\frac{r}{m})$ respectively (m is the concurrent task number per container). Since the memory usage per container is the aggregate ones of its all co-locating tasks that $x_j^t = \sum_{i=1}^{i=m} y_i^t$, its probability would also be the sum of tasks as Eq.(3.1):

$$P(x_j^t|r) = P(\sum_{i=1}^{i=m} y_i^t|r) = \sum_{i=1}^{i=m} P(y_i^t|\frac{r}{m}). \tag{3.1}$$

Understanding the memory usage probability distribution per task is critical to predict container's footprints.

We profiled every task's memory usages of motivation experiment in Table 3.3 by setting m as 1 and reservation size as $\frac{R}{m'}$ (R and m' are the ones of Table 3.2). We display similar statistics like mean μ (MU), standard deviation σ (StDev) and peak (P) of usages per task in the same representative iterative stage as in Table 3.2. The data are average ones of all tasks per stage. For robustness, they are obtained from the mean of repeated 30 executions per workload. We also illustrate the coefficient of variation (CoV) of these statistics among all tasks, as well as the CoV of tasks' inputs (I) and shuffled (S) sizes per stage.

Table 3.3: Task memory usages summary per stage

Benchmarks	R (GB)	MU (CoV)	StDev (CoV)	P (CoV)	I, S CoV
Terasort	3.5	1.1 (0.042)	0.4 (0.03)	1.5 (0.05)	0.12, 0.35
SVM	3.5	1.4 (0.047)	0.45 (0.06)	1.8 (0.05)	0.38, 0.45
KMeans	3.5	2.2 (0.003)	0.53 (0.008)	2.6 (0.004)	0.08, 0.3
LogisRegre	1.5	0.3 (0.029)	0.1 (0.056)	0.6 (0.05)	0.07, 0.23
LinearRegre	1.5	1.2 (0.012)	0.24 (0.02)	1.56 (0.006)	0.004, 0.44
DecisionTree	2	1.3 (0.054)	0.28 (0.07)	1.7 (0.03)	0.07, 0.52
PCA	3.5	2.4 (0.014)	0.58 (0.03)	2.9 (0.01)	0.28, 0.09
PageRank	4	2.1 (0.092)	0.55 (0.085)	2.5 (0.07)	0.09, 0.3
SVD++	3.5	2.7 (0.039)	0.32 (0.05)	3.15 (0.06)	0.5, 0.4
CC	1.5	0.4 (0.082)	0.19 (0.09)	0.8 (0.12)	0.63, 0.55
TriangleCount	3.5	1.3 (0.063)	0.54 (0.06)	1.9 (0.06)	0.3, 0.02
TPC-DS	3.5	1.6 (0.055)	0.35 (0.04)	2.1 (0.045)	0.4, 0.52

Probability distribution of task usages. For all workloads, we observed the variations of dispersion (CoV) of tasks' MU, StDev and P per stage are significantly small that less than 0.1. Most tasks have similar footprints with consistent probability distribution. Despite containers' footprints are random, the memory usage behaviors of every task within a stage is in a highly stable and repeated pattern. It is due to the numerous novel sampling techniques and advanced load balance optimization of input and intermediate shuffled data [43, 47, 81, 84, 98, 110, 117, 118, 161, 174, 182, 189, 205]. We could see the CoV of inputs and shuffled data sizes between tasks are mostly less than 0.5, one third of which are even less than 0.1. It indicates the input data skew and imbalanced shuffling problems are well resolved in modern in-memory computing system.

Additionally, memory usage per task fluctuates little that StDev are as small as one fifth to one tenth of average usages for all workloads. Peak usages are also close to mean consumption which are only about 30% higher. Most usages are located around average ones μ and within an interval of $\mu \pm \sigma$. More than 95% usages are within $\mu \pm 2\sigma$ while 99% accounts for $\mu \pm 3\sigma$. We found the probability distribution of footprint per task for arbitrary over-provisioned workload resembles to Gaussian distribution that $\hat{y_i^t} \sim N(\hat{\mu}, \hat{\sigma}^2)$, where $\hat{y_i^t}$, $\hat{\mu}$ and $\hat{\sigma}$ are the estimated usages, mathematical expectation and standard deviation of footprint per task. We could observe it from Table 3.2 and Table 3.3, and have its probability

density function:

$$P(\hat{y_i^t}|\frac{r}{m}) = \frac{1}{\hat{\sigma}\sqrt{2\pi}} e^{-\frac{(\hat{y_i^t}-\hat{\mu})^2}{2\hat{\sigma}^2}}. \tag{3.2}$$

By our repeated experiments of statistical hypothesis testing by combinations of multiple methods like u-, t-, F- and Chi-square tests, the Gaussian distribution of memory usages per task is shown to be robust for arbitrary workload and stage of over-provisioning. Its stable expectation in normal distribution sufficiently indicates base demands.

Predicting confidence intervals of Gaussian distribution. The tasks of different stages have distinct memory requirements and expectations. To predict memory usages $\hat{y_i^t}$ per task at arbitrary stage, we generate its confidence interval: an estimate of the range of values within which the true value should lie with a certain confidence level (a probability, γ) [60]. The higher the confidence level, the wider the confidence interval, and lower the risks of mis-predictions and mis-provisionings. The confidence interval calculation relies on the variance of the prediction errors and the confidence level γ. We define the significant level $\alpha = 1 - \gamma$, and the prediction interval ci^k of $\hat{y_i^t}$ at stage k is given by

$$ci^k = \hat{y_i^t} \pm \hat{\sigma} * z_{\alpha/2}, \tag{3.3}$$

where $\hat{\sigma}$ is the estimated standard deviation for the prediction errors, and that is tasks' StDev per stage in Table 3.3. $z_{\alpha/2}$ is the value for the $100 * \alpha/2$ percentile in the normal distribution. Since task footprints within a stage are stable, ci^k represents the intervals of all tasks per stage. We adopt a general approach to estimate $\hat{\sigma}$ by calculating the standard deviation from the prediction errors (residuals) when applying the fitted forecast model to the profiled usages data of different tasks per stage used for training.

It is difficult to effectively predict probability distribution of highly random and uncertain containers' footprints as shown in Section . However, since memory usages per container is the aggregate ones of co-locating tasks as in Eq.(3.1), its confidence intervals are

similar. By estimating ci^k of task usages, we could accurately estimate confidence intervals of containers' footprints at any stage k:

$$ci(x_t^{k,j}) = \sum_{i=1}^{i=m} ci^k. \tag{3.4}$$

In MEER, we performed two pilot runs to study footprints for arbitrary newly submitted application: one under user requested parallelism to collect containers' footprints, another under concurrency 1 to profile footprints per task. Compared to the random containers' profiles, task footprint is stable and its profiling is robust which only needs to be profiled once per recurring application.

Overview of Workflow and Implementation

MEER adopts a hybrid mechanism by combining an online estimation model of histogram frequency analysis and a recursive search loop. We define notations used in this paper in Table 4.7. Figure 3.4 gives an overview of MEER's workflow. First, the newly submitted application executes two pilot runs under over-provisioned reservations (step 1). We calculate the confidence intervals of containers' footprints by profiling tasks' footprints, and analyze its frequency of memory usages at every sampling point. By accumulation of usages multiplying corresponding frequency and confidence level, we obtain the expectation of containers' footprints, so as to estimate an initial base demand and near-optimal reservation R_*^j per application (step 2) and guide normal executions (step 3). Since tasks per stage with stable usages are repeated per run, these profilings could be used to effectively predict demands boundary per container of future recurring executions.

For user's n^{th} submission in subsequent recurring runs, MEER performs adaptive predictions and search (inner loop), and recursively adopting last estimation $R_*^j(n-1)$ as next reservation to execute applications (step 4), which generates new estimation $R_*^j(n)$ and runtime $ET(n)$ (step 6 & 7). Since the histogram analysis algorithm has an intrinsic property of rapid self-decay and searching loop starts from a near-target position of $R_*^j(0)$,

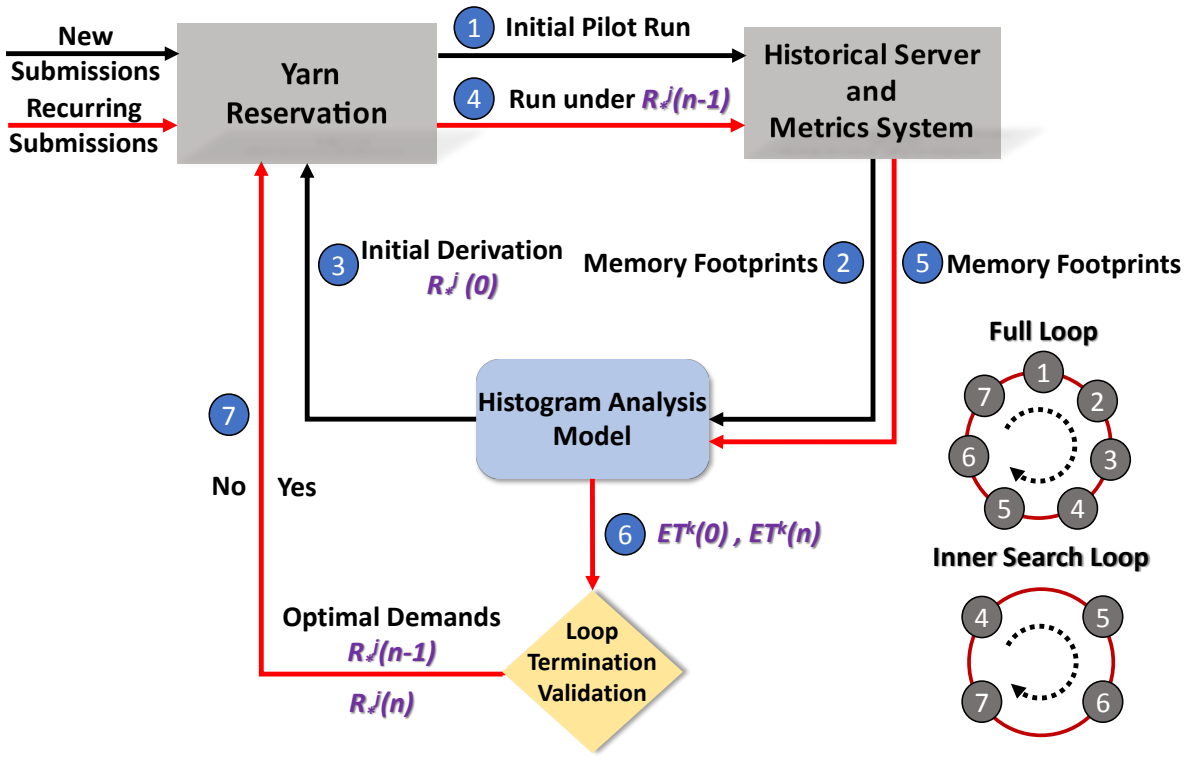


Figure 3.4: MEER's architecture and workflow.

MEER always efficiently approaches the optimal reservations in a few recurring executions with low search overheads. By comparing the execution time $ET(n)$ with upper-bound performance $ET(0)$ acquired from pilot runs, MEER determines whether the target is reached and searching loop is terminated (step 7). Any variability of recurring jobs would trigger a new searching loop to handle the migration of performance inflection points (full loop).

Implementation. We implemented MEER using Python as an extension to Apache Spark and Yarn/Hadoop in a non-intrusive way. Users submit their applications normally without extra efforts. MEER runs on Linux as a seperate process. It obtains and stores all metrics of usages and runtimes by communicating with Spark historical web server and real-time metrics system [16] through RESTFul API and FTP. Histogram analysis model of MEER spontaneously interacts with resource manager (RM) of YARN to trigger pilot runs through YARN API. RM relies on the estimated results from MEER to reserve just right memory for containers.

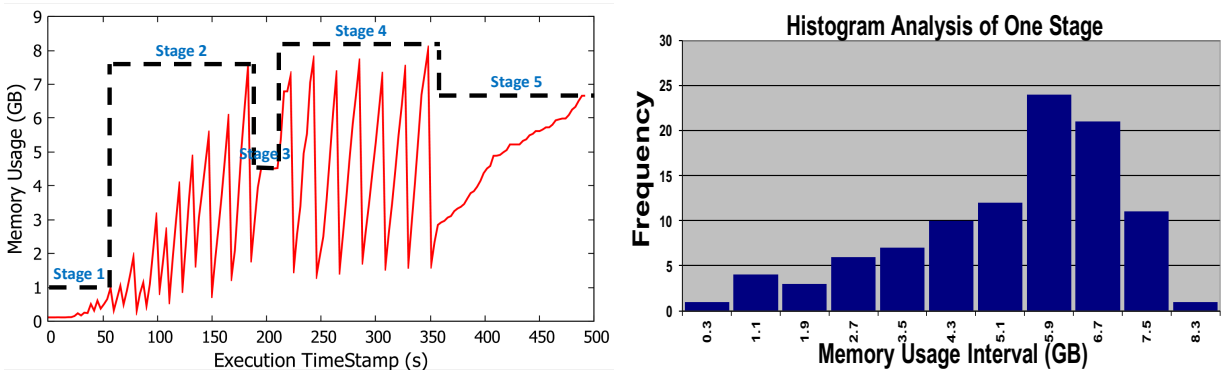
Table 3.4: Input Parameters Notations

R_*^j	Optimal memory reservation estimation of every container j per application
$x_t^{k,j}$	Memory usage of container j at sampling time t in stage k
$Interval^{k,j}$	Sampling intervals of memory usages for container j at stage k
$Count^{k,j}$	Sampling counts of container j at stage k (Number of $x_t^{k,j}$ points)
$ET(n)$, $ET^k(n)$	The execution time of the application and stage k at n^{th} recurring execution
$R_*^j(n)$	n^{th} optimal reservation estimation of every container j per application, which guides $(n+1)^{\text{th}}$ recurring execution
$peak_*^j(n)$	Peak memory usages among all stages and containers at n^{th} recurring execution
$HeapUtil^*(n)$	Average heap utilization among all stages and containers at n^{th} recurring execution
$GCTime^*(n)$	An application's aggregate GC time of all stages at n^{th} recurring execution

Histogram Frequency Analysis

Our goal of building histogram analysis model is to rapidly and accurately provide online optimal reservations estimation $R^{k,j}$ per stage for newly submitted and non-recurring applications through one step. It provides a good start point to facilitate recursive search in further recurring executions. The model's initial input data are acquired from only two pilot runs under over-provisioned reservations. We adopt user's pre-claimed number of containers and parallelism for allocation. The real-time memory footprints (sampled at a fixed time interval, e.g., 1 second in Spark by default.) are obtained from Spark metrics system [16]. They are segmented by stage timestamp that acquired from profiled runs data in historical log server. We introduce a number of notations as shown in Table 4.7.

To understand applications' memory usages per stage, we plot one container's time-varying footprints of over-provisioned SVM workload in Figure 3.5a as an example. We could see usages are always unstable and drastically fluctuate at arbitrary stage, and they seldom reach the peak. The usages between stages are distinct due to different operations. MEER explores each stage seperately. To better understand its probability distributions, we



(a) Footprint and peak usages

(b) Histogram frequency of the 4th stage

Figure 3.5: Footprint and histogram of SVM.

transform the time-domain footprints to a frequency-domain histogram. Figure 3.5b plots the frequency of its 4^{th} stage. We observed the peak usages are usually a small fraction of the whole stage. The frequencies of peak (more than 7.5 GB) and low usages (less than 1.9 GB) are only 1% and 8% respectively. In comparison, intervals of relatively large usages (from 5.1 to 6.7 GB) occupy a high percentage of 45%. Peaks are 2 to 6 times larger than the average ones as shown in Section , which would dominantly result in containers' wasteful base demands estimations. All other workloads' footprints have the consistent trends.

Simply predicting optimal reservations by omitting large usages in case of overstating their importances on base demands estimation is infeasible. They have a decisive impact on the performance. Doing so would slow down most tasks and incur massive stragglers during load spikes, which highly increases risks of out-of-memory problems and program failures. We need to adopt histogram analysis to perform probability density estimation on containers' profiled footprints to balance the trade-off between memory efficiency and performance.

We define the probability of $x_t^{k,j}$ that occurs at sampling time t is $P(x_t^{k,j})$:

$$P(x_t^{k,j}) = \frac{Freq(x_t^{k,j})}{Count^{k,j}}, where\ Count^{k,j} = \frac{ET^k}{Interval^{k,j}}, \tag{3.5}$$

and $Freq(x_t^{k,j})$ is the occurance times of $x_t^{k,j}$. Since the expectation $\mathbb{E}(x_t^{k,j})$ of all containers' footprints reflect the probable average consumption in future executions, it implies the

majorities of base demands and optimal reservation per stage:

$$E(x_t^{k,j}) = \sum_t (x_t^{k,j} * P(x_t^{k,j})). \tag{3.6}$$

From Section , we know containers' usages $x_t^{k,j}$ are highly random and its estimated probability $P(x_t^{k,j})$ of one-time profiling is not necessarily applied to future runs. However, we know its accurate confidence intervals $ci(x_t^{k,j})$ and levels γ from Section of arbitrary execution. We add a weight γ to $P(x_t^{k,j})$ to reflect its true probability. In case of excessively conservative and useless range, we pick the minimum intervals $min(ci(x_t^{k,j}))$ that $x_t^{k,j}$ locates. By using confidence level for the predictions, we eliminate the negative effects of container footprints' randomness and achieve stable estimations. For every stage k, MEER predicts optimal reservation $R^{k,j}$ per container j [4] as in Eq.(3.7):

$$\begin{aligned} &R^{k,j} = \sum_t (x_t^{k,j} * P(x_t^{k,j}) * \gamma), \\ &\text{where } x_t^{k,j} \subseteq min(ci(x_t^{k,j})). \end{aligned} \tag{3.7}$$

Histogram analysis model implicitly emphases the importances of peak usages with high frequency and compromises their immense impacts on performance. It also avoids empirically exaggerating the importances of low-frequency peak usages and prevents unnecessarily excessive base demands estimations.

The reservation is fixed across stages during executions. To guarantee smooth runs and optimal performance per stage, and thoroughly avoid failures due to insufficient allocations, we pick the largest $R^{k,j}$ among all stages as the estimated optimal reservation per application:

$$R_*^j = Max(R^{k,j}). \tag{3.8}$$

[4]The reservations of all containers per application are consistent.

The maximum optimal reservations of representative iterative stages could effectively accommodate and stand for the whole application. At the cost of little wasteful over-provisioning in few stages, MEER achieves distinguished reliability and memory efficiency.

Recursive Search Loop

The histogram model provides an accurate estimation of optimal reservation through one-time estimation. There are still portions of underutilized memory caused by overestimation and over-provisions that could be improved. For further recurring submissions, we adopt a recursive search loop based on a feed-back control machanism to achieve stepwise refined estimation, and gradually approach reservation of the optimal one.

Self-Decay Property. For recurring applications, MEER recursively profiles only containers' footprints and performs histogram analysis of every execution. We adaptively adopt last estimation R_*^j as next reservation. Since the estimation $R_*^j(n+1)$ represents the majorities of base demands at $(n+1)^{\text{th}}$ execution, it would be far less than its peak usages $peak_*^j(n+1)$. Meanwhile, the $peak_*^j(n+1)$ of arbitrary stage at $(n+1)^{\text{th}}$ execution should be less than its reserved (JVM heap) size, which is $R_*^j(n)$ generated from last estimation.

Consequently, the histogram analysis algorithm is shown to have an intrinsic property of rapid self-decay. This is an implicit relationship as expressed in Eq.(3.9):

$$\begin{aligned} &R_*^j(n+1) < peak_*^j(n+1) < R_*^j(n) < peak_*^j(n), \\ &Max(R_*^j(n)) = R_*^j(0) \ll \text{Unlimited Mem Resrv}, \end{aligned} \tag{3.9}$$

where reservation $R_*^j(n)$ is always significantly larger than its estimated $R_*^j(n+1)$. The estimation $R_*^j(n+1)$ decreases promptly per round and gradually approaches the vicinity of optimal reservation. It serves as the basis for rapid and efficient recursive search.

Online Recursive Search. The initially estimated $R_*^j(0)$ from pilot runs is provided to resource manager (Yarn) to guide the application's first-time execution. Afterwards, we recursively obtain new runtime $ET(1)$ and estimation $R_*^j(1)$. We observed applications

achieve a nearly upper-bound performance $ET(0)$ under significantly over-provisioned allocations. Runtime of subsequent reservations are closely approaching $ET(0)$ due to elasticity. MEER does not need any extra execution and overhead to perform search. Recursive search loop is inherently robust to the variabilities of recurring jobs. To accommodate the possibility of knee migration, arbitrary type of input data changes, parameter tuning or code tweaks would trigger a new search loop.

Terminal Condition. Through self-decay estimations and recursive search, we ultimately find performance inflection point $ET(n)$ when it dramatically drops to $ET(n+1)$ due to insufficient memory under reservation of $R_*^j(n)$. It also occurs frequent long and useless garbage collections (LUGC) [82], leaving little time for task execution. The current reservation $R_*^j(n)$ causes significant performance degradation and application gets into the crash zone. Afterwards, MEER terminates searching loop and adopts last proper estimation $R_*^j(n-1)$ as the optimal reservations, which satisfies the termination condition of Eq.(3.10):

$$\text{if} \begin{cases} \frac{ET(n+1)-ET(n)}{ET(n)} > \beta, \\ \frac{ET(n+1)-ET(0)}{ET(0)} > \beta, \\ \frac{GCTime^*(n+1)}{ET(n+1)} > \gamma, \\ HeapUtil^*(n+1) > 80\%, \end{cases} \tag{3.10}$$

$$\text{Optimal Reservations} = R_*^j(n-1).$$

We set the performance degradation threshold to 0.5 for β and 0.7 for γ; These parameter settings were widely used in previous studies [82, 136, 150, 151, 187]. They are sufficiently large to differentiate normal slow down from being in crash zone caused by LUGC. Under allocation of the optimal reservation, arbitrary workload could achieve repeatedly predictable optimal runtime and best memory efficiency, which is critical to guarantee SLO.

Table 3.5: Each set of 15 input sizes for 15 benchmark workloads

Benchmarks	TeraSort, WC, Sort, Grep	SVM	KMeans	LogisR, LinR	Decision Tree
Minimum Input Dataset	50GB	30GB	20GB	30GB	50GB
Incremental Interval	10GB	8GB	6GB	15GB	10GB
Maximum Input Dataset	200GB	150GB	110GB	255GB	200GB
Benchmarks	SVD++	PageRank	PCA	CC	TriangleCount
Minimum Input Dataset	0.5GB	1GB	60GB	0.5GB	0.5GB
Incremental Interval	0.5GB	1GB	16GB	0.5GB	0.5GB
Maximum Input Dataset	8GB	15GB	300GB	8GB	8GB
Benchmarks	TPC-DS 7				
Minimum Input Dataset	50GB				
Incremental Interval	10GB				
Maximum Input Dataset	200GB				

Evaluation

We evaluated MEER on a local 16-server cluster deployed with Hadoop Yarn 2.7.2 and Spark 2.2.0. Each server is configured with 24 cores, 32GB of memory, three 3.5TB 7200 RPM disk drives with a 110MB/s peak bandwidth. It is equipped with a 1Gbps NIC and runs Linux 3.16. We changed input data sizes (15 settings) of 15 benchmark workloads. In adddition to the ones listed in Table 3.5, we also included WordCount (WC), Sort and Grep from BigDataBench [188] by using real Wikipedia and Amazon productions reviews data. In total, we had 225 distinct applications for evaluation, which were executed one by one for a total of 15 times.

We separated the experiments into two parts: examine prediction accuracy and evaluate effectiveness of MEER on applications performance and cluster memory efficiency during batches running. Diverse types of workloads with different inputs have distinct memoy usage patterns and optimal reservations. We explored 225 optimal reservations as baselines manually. We measured how close an optimal reservation estimation is to ground-truth, which were obtained from manual exhaustive experiments and brute-force search like Section . We used the ratio of differences between prediction and ground truth to ground truth $\frac{Predicted - Actual}{Actual}$ as error metric.

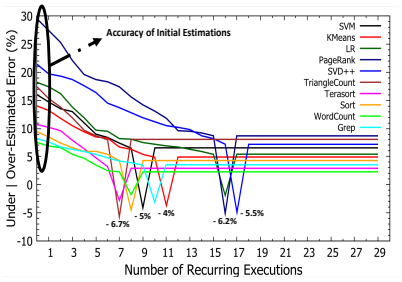


Figure 3.6: Under- and over-estimated errors during recurring executions.

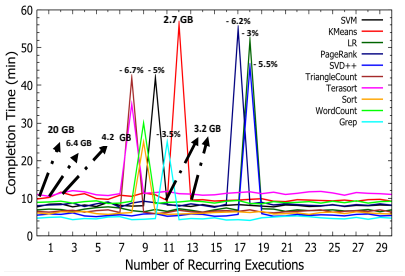


Figure 3.7: Variations of application runtime during recurring executions.

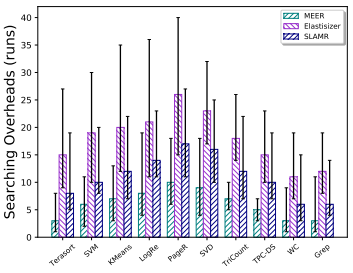


Figure 3.8: Searching overheads of MEER, Elastisizer and SLAMR.

We compared MEER with alternative solutions, including representative task memory estimator Elastisizer [101] of MapReduce optimizer Starfish [102] and the most related open source container demands estimator SLAMR [32, 164] for Spark. We demonstrated MEER averagely reached up to 5 times lower searching overheads and 8 times more accuracy. We also compared MEER to default YARN/Spark with user's empirical memory configurations, which is estimated based on input sizes per container as recommended by benchmark and Spark official website [17]. We illustrated MEER speeded up diverse applications by 2 to 6 times on average while improving cluster memory utilization by 40%.

Accuracy of Prediction

Initially, there was no relevant execution statistics in Spark historical server. We submitted 225 applications one by one, and MEER performed pilot runs. We evaluated predictions accuracy during executions.

1) Evaluation of MEER: Figure 3.6 presents under- and over-estimated errors of memory sizes during recurring runs. They are resulted from the executions of median configurations in Table 3.5.

Accuracy of initial estimations. We observed that all applications have over-estimated errors (prediction > actual, error is positive) at the beginning after pilot runs. Batches jobs (WordCount, Grep, Sort, Terasort) get errors within 9% while machine learning jobs around 16%. PageRank reaches almost 30% due to its complicated data parallelism. All workloads

achieve an accuracy as high as 80% after pilot runs. Since the base demands tend to be large values, initial under-estimations (prediction < actual, error is negative) seldom occur. We plan to dynamic adjust memory reservations based on estimation per stage instead of a static maximum one per application to reduce over-estimated errors in the future.

In the subsequent recurring executions, MEER recursively adopts last estimation to guide exploration for next reservation. For example, the completion time of the 1^{st} recurring execution is achieved under reservation of 0^{th} initial estimation, while the 0^{th} performance is resulted from an over-provisioned reservation (20GB per container). By means of a self-decay property of recursive search, these progressively diminishing estimations gradually reduce over-estimated errors until a negative under-estimated one occurs. They are generally as small as - 6.2%, -5% and -5.5% in SVM, PageRank and SVD++.

Figure 3.7 displays applications' completion time under the estimated reservations. It can be seen that around 5% under-estimated errors lead to severe performance degradation (4 to 10 times longer completion time) of all applications during under-provisioned reservations. We found most containers sustain thrashing and hovered by frequent long and useless garbage collections (LUGC) due to these shortages of memory. A portion of containers are encountering out-of-memory errors that ultimately crash. Since these applications rely on containers to cache intermediate results to speedup, it would involve high-overhead recoveries and re-computations. For example, KMeans slows down to 56 minutes when under-estimated 2.7 GB is used from normally near 10-minute completion time under 3.2 GB. These dramatic drops reveal that applications reach the vicinity of optimal reservations. A slight reducation of memory provision would make applications get into the crash zone. To avoid such unbearable under-provisioned allocations, MEER backtracks to last over-estimation of 3.2 GB as a near-optimal reservation for future recurring executions. Meanwhile, the search process terminates with a steady accuracy.

Most batches jobs take 7 to 10 executions to obtain steady optimal reservations while iterative jobs take longer. Graph applications like Triangle Count have relatively simple

operations and memory usages patterns. By obtaining estimations after several profiled footprints, their base demands tend to be stable. The estimations rapidly converge and searching is terminated faster. But its ultimate 8% over-estimated error might be worse than others. Although PageRank has a high error rate at initial estimation, it improves the ultimate accuracy to 91.3% after subsequent 16 recurring executions. The high accuracy of near 96% (2.3% error for WordCount, 2.9% for Terasort) for batches jobs is outstanding. Machine learning jobs achieve around 95% and graph computation jobs reach 92%. Observed from accuracy of initial and ultimate estimations, as well as searching overheads, batches jobs outperform others. The gain mostly comes from their highly predictable execution logics and stable footprint patterns. The fewer number of stages than iterative workloads mitigates their uncertainties of estimations.

Since over-estimated reservations appear at most runs, applications achieve stably near-optimal performance within 3% variations during recursive search. For example, the completion time of KMeans is within 3% differences under reservation of initial over-provisioning (20 GB), 1st estimation (6.4 GB), 2nd estimation (4.2 GB) and ultimately optimal one (3.2 GB).

Consequently, MEER achieves an over 80% accuracy at initial estimations for newly submitted jobs from only two pilot runs. Through recursive search during a few recurring executions, errors drastically drop to within 10%, and reach a steady accuracy over 95% for most workloads in our test cases.

2) Compared with Alternative Solutions: We demonstrate MEER's online recursive search outperforms alternative solutions from both accuracy and searching overheads in this experiment. We ran search for above 225 applications under Elastisizer, MEER and SLAMR. The search per application is executing 15 times and computing the average values. Elastisizer adopts random search and coordinate descent that used in [102,201]. We executed it with different seeds of starting points and step sizes.

Figure 3.8 shows the average, minimum and maximum searching overheads for diverse workloads with distinct inputs under three estimators. We observed Elastisizer needs to simultaneously select a best starting point and step size (large enough) to achieve low search overheads, which is challenging and impractical through a few recurring executions. It required 3 to 6 times more overheads than MEER on the average, and 4 to 7 times on the tail. SLAMR performed searching by repeated insufficient sample profilings. It averagely needed 1.5 to 2 times more overheads than MEER, and 2 to 3 times on the tail. Due to the loose and inaccurate convergence condition, SLAMR always terminates earlier than Elastisizer. Batch jobs (Terasort, WC, Grep) commonly needs less search runs. MEER obviously achieves 3 and 6 times fewer overheads compared to SLAMR and Elastisizer on them. Since every type of workload with distinct size of inputs has unique optimal reservations, they need to start over new searches. The 7 times increased overheads per search means overall tens of thousands of extra executions for diverse applications in an enterprise datacenter, which is an unaffordable cost for online scheduling.

Figure 3.9 displays the ultimate average, minimum and maximum accuracy of above search with different inputs. MEER achieved an error rate of less than 10% for most workloads and within 5% for batches jobs even on the tail. In comparison, Elastisizer averagely reached the accuracy about 80% for most workloads, while the errors were about 30% on the tail. MEER's initial estimations even outperformed Elastisizer. This is because the step size of Elastisizer was always improperly large and ultimately got away from the target. Rapidly selecting proper step size for each specific application, while balancing trade-offs between searching overheads and accuracy over a few runs is non-trivial. SLAMR conservatively adopted containers' peak usages across stages as estimations and caused severe over-estimations. It resulted in average errors of around 18% for batches jobs, and about 30% to 40% for iterative workloads. Despite its search overheads are relatively moderate, the accuracy is unacceptable that even worse than Elastisizer. MEER obviously outperforms alternative solutions. It achieves 4 or 7 times less errors on the average, and 6 or 8 times on

the tail compared to Elastisizer or SLAMR. MEER consistently delivers a stability of low overheads and high accuracy.

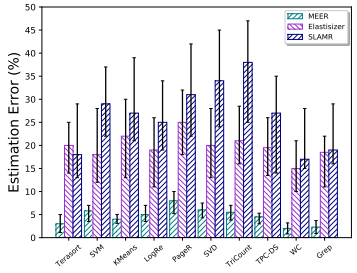


Figure 3.9: Ultimate accuracy of estimations under MEER, Elastisizer and SLAMR.

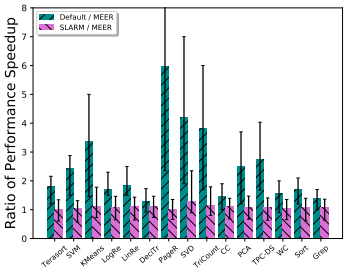
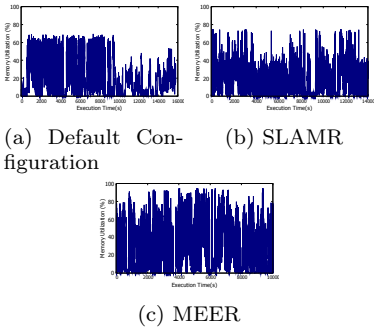


Figure 3.10: Execution time speedup of default YARN to MEER, and SLAMR to MEER per application.



(a) Default Configuration (b) SLAMR

(c) MEER

Figure 3.11: Memory utilization during batches runs under various demands estimators.

Performance on the Batches Running

To evaluate the effectiveness of MEER, we respectively equipped Yarn with MEER and SLAMR, and submitted above workloads in a batch to the system at a time randomly selected between 0 and 1200 seconds. Each application is submitted for 15 times to obtain average durations. Elastisizer is designed for the first generation of Hadoop [14] and its prototype is not applied to Spark workflow. Simply executing it on batches runs of in-memory computing workloads and making comparisons are unfair. The baselines are executions under estimated memory reservations by default Yarn/Spark and SLAMR. We evaluated cluster memory utilization and execution time of applications. We used the runtime ratio of baseline to MEER ($\frac{Baseline}{MEER}$) per application to measure the performance gains of diverse types of workloads by MEER.

1) Performance: Figure 3.10 shows the distribution of runtime speedup under different estimators per application. To show the effectiveness of MEER under various input sizes and varied knees, every bar indicates the average ratio of applications under 15 distinct input sizes per workload. It also includes the minimum and maximum ratios of performance

gains of different inputs. Each ratio is an average of 15-times runs to preclude effects of co-locating interferences.

Compared to the default. Default YARN/Spark estimates containers' memory demands through analysis of task input sizes based on error-prone empiricism. The estimation is highly inaccurate because the compressed and serialized on-disk format of data is always 3 to 6 times larger in memory, which is uncertain and random. Runtime slow down caused by insufficient under-provisioning are ubiquitous in cluster. For most applications, MEER (M) obviously outperforms default reservations (Def) by 2 to 5.9 times. Shuffle-intensive applications consist of numerous iterative stages like KMeans and graph computing workloads are sensitive to memory shortage, and would trigger massive time-consuming full garbage collections and spills during shuffling. When the reservation slightly decreases from optimal 4 GB (M) to insufficient about 3 GB (Def) by 25% under-estimations for large inputs, runtime would increase from 18, 19 and 23 minutes (M) to 1.5, 2.2 and 2.3 hours (Def) for KMeans, SVD++, and TriangleCount respectively. The slow down could be as much as 5, 7 and 6 times. Workloads get into a crash zone from a safe tail region under small under-estimations. For most workloads, the long runtime of applications due to large data inputs degrades more drastically under mis-provisions.

For PageRank with 15 GB data inputs, an inproper small reservation of 3 GB per container causes performance degradation from 18 minutes (6 GB of MEER) to 29.3 hours (Def). Applications are hovered by severe LUGCs and thrashing, with a plenty of container crashes and massive re-computations. It wastes numerous resources of a cluster for an abnormally long time, which slow down the entire batch of runs. Other workloads like KMeans, SVD++ and TriangleCount also have high risks of abnormal long-time executions in some potential configurations. They tend to have large inflection point values as in Figure 3.1, leaving substaintial spaces for insufficient under-estimations. They are mixed with complex transformations like treeAggregate, coalesce and cartesian through complicated data communications between shuffled tasks of various iterations.

Batch jobs like Terasort, WordCount, Sort and Grep have relatively small performance gains that average around 1.8 times. They could be well predicted by default estimator due to their simple data paths between a few stages. Others like Connected Component, Decision Tree and Regressions mostly read and write of data by disk or perform common operations locally. They have similar average gains of about 1.6 times and their runtime decrease steadily with sizable reductions of reservations. From another view point of maximizing resource efficiency, they are more friendly to elasticity and able to execute smoothly under considerably less memory.

Consequently, a slight difference of mis-estimation is as good as a mile. Considering the tremendous performance loss and the probable programs failures in crash zone, applications generally achieve about 2 to 6 times speedup under schedulers that assemble MEER, by thoroughly avoiding under-provisions.

Compared to SLAMR. We have two observations regarding to the performance due to SLAMR. First, there are a few percentage (about 5%) of executions whose performance due to SLAMR even outperform MEER. It is due to the minority of under-estimations in MEER just before termination of the search. In such cases, SLAMR may win. Second, SLAMR consistently delivers a comparable execution time to MEER (within 10% differences). The reason is SLAMR always conservatively adopts over-provisioned reservation based on holistic peak demands across stages, at the cost of huge memory wastes. Thereby, its performance is always close to the optimal one as MEER. SLAMR is not able to accurately forecast memory demands of complex data communications in graph computing workloads. The largest runtime variances are 2.3 times of SVD++ and 1.8 times of Triangle Count.

2) Resource Efficiency: To reveal insights into memory efficiency, Figure 3.11 shows cluster utilization under three estimators during batches runs. The utilization is a ratio of actual usages to cluster capacity. Default reservations and SLAMR always exaggerate the importances of peak demands by significant over-estimations, which are actually an insignificant small portion and have slight impacts on runtime. As shown, there are severe

over-provisioned reservations under default configurations by error-prone empiricism, which suggest a low average utilization of 60%. SLAMR has a even lower one around 40%, indicating it yields more significant wastes of underutilized memory based on peak reservations. These large proportions of spare memory (40% to 60%) could not be used by other waiting applications.

In contrast, MEER improves utilization to an average by 80%, and achieves expected high memory efficiency. The benefits come from tight and just proper sizes of reservations under accurate initial and refined estimations that accomodate just major base demands. Average heap utilizations are also shown to exceed 82% under optimal provisions as shown in Section . Every reservation by MEER maximizes memory utilization while guaranteeing near-optimal application performance.

Summary

In this paper, we present MEER, a system that assists schedulers to accurately and efficiently estimate optimal memory reservations for diverse in-memory computing workloads. We demonstrate a general property of long-lived containers which referred to memory reservation elasticity and the concept of optimal reservation. By leveraging robust profiling, confidence level and probability density analysis for predictions, MEER achieves accurate initial estimations in one step on line. Because of an intrinsic self-decay property of the histogram analysis results, MEER rapidly reaches optimal reservations through a few recursive search steps in future recurring executions. MEER is an effective tool to promptly accumulate resource demand knowledge for schedulers, and an essential component towards future data-driven intelligent scheduling through self-learning. It is also a complementary technique to existing schedulers that leverage knowledge of future resource availability. It improves their effectiveness and strengthen benefits. Overall, the optimal reservation knowledge provided by MEER enable cluster managers to achieve both optimal application performance and maximum cluster memory efficiency.

CHAPTER 4 Prophet: Scheduling Containers with Time-varying Resources Demands on Data-Parallel Computation Frameworks

Introduction

Resource allocation is crucial to data-intensive cluster computation of big data systems. Efficiently scheduling execution instances of data-parallel computing frameworks, such as Spark and Dryad, on a multi-tenant computation platform is critical to applications' performance and systems' utilization. To this end, one has to avoid resource fragmentation and over-allocation, so that both idleness and contention of resources can be minimized. To make effective scheduling decisions, a scheduler has to be informed of and exploit resource demands of individual execution instances, such as short-lived tasks or long-lived executors. The issue becomes particularly challenging when resource demands greatly vary over time within each instance. Prior studies take the convenience of assuming that a scheduling instance is either short lived or of relatively consistent resource demands.

However, when in-memory computing platforms, such as Spark, become increasingly popular, the assumption does not hold. The scheduling instance becomes executor for executing an entire application once it is scheduled. Usually It is not short lived and is of significantly time-varying resource demands. To address the inefficacy of state-of-the-art cluster schedulers, we propose *Prophet*, which takes resource demand variation within each executor into its scheduling decision. To know the varying demands at the time of scheduling, it leverages the fact that execution of a data-parallel application is well pre-defined by its DAG structure and its resource demands at various DAG stages are highly predictable. Equipped with this knowledge, Prophet schedules executors aiming to minimize resource fragmentation and over-allocation. To accommodate unavoidable or unpredicted resource contention as well as resulting performance degradation, Prophet adaptively backs off selected task(s) to remove the contention. We have implemented Prophet in Apache Yarn running Spark and evaluated it on a 16-server cluster. Compared to Yarn's default capacity

and fair scheduler, Prophet reduces makespan by up to 39% and reduces median application completion time by 23%.

Efficiently scheduling execution instances of data-parallel computing frameworks, such as Spark and Dryad, on a multi-tenant computation platform is critical to applications' performance and systems' utilization. To this end, one has to avoid resource fragmentation and over-allocation, so that both idleness and contention of resources can be minimized. To make effective scheduling decisions, a scheduler has to be informed of and exploit resource demands of individual execution instances, such as short-lived tasks or long-lived executors. The issue becomes particularly challenging when resource demands greatly vary over time within each instance. Prior studies take the convenience of assuming that a scheduling instance is either short lived or of relatively consistent resource demands.

Scheduling tasks of multi-resource demands onto servers of given amount of resources (CPU, memory, disk, and network) is often formulated as a multidimensional bin packing problem. As long as the demands are known a priori or can be accurately estimated, the problem has been well addressed [94]. A common technique used for this estimation is to leverage the fact that jobs of an application are recurring and they "repeat hourly (or daily) to do the same computation on newly arriving data." [94]. Therefore, tasks' statistics measured in their prior runs enable effective estimation. Specifically, "since tasks in a phase perform the same computation on different partitions of data, their resource use is statistically similar." [94]. An offline or online profiling of tasks' runs would provide a scheduler with knowledge on tasks' resource demands.

Unfortunately the profiling strategy does not fully address the issue by itself in practice, as the demand measured during a task's run varies (sometimes dramatically). While a multidimensional bin packing problem is NP-hard and has to be solved with heuristics, it is almost impossible to take time-varying demands into consideration of scheduling decisions. A conservative and safe alternative is to use the peak usage of a resource to represent the varying ones to prevent over-allocation. However, this produces risk of resource fragmentation.

While each application can have a large number of tasks and each task has a relatively short execution time, using peak demand may not create large pockets of fragmentation in terms of wasted resource time. However, this becomes a serious issue with in-memory computing frameworks, such as Spark [203] and Storm [176], and can cause significant performance loss.

To achieve high scheduling efficiency, a scheduler has to minimize fragmentation and overallocation of resources [94]. When resources are idle with demands on the resources from tasks to be scheduled, there is resource *fragmentation*. One scenario where this happens is when resources, such as CPU and memory, are pre-allocated into slots where tasks are to be dispatched [27,87]. When aggregate demands from running tasks exceed available resources, *over-allocation* of resources occurs and often leads to interference and serious performance degradation. While a task's CPU and memory demands are often well pre-specified and met by resource pre-reservation, the over-allocation usually happens with network or disk and causes disk seeks or network incast significantly compromising their throughputs.

An in-memory computing framework, such as a Spark application, does not expose its tasks to the platform it runs on, such as YARN or Mesos, for it to directly schedule. Instead, it introduces the concept of executor[5], which is scheduled by the platform's scheduler. Once executors of a framework are scheduled to servers, the framework's scheduler is responsible for scheduling its tasks to the executors. Specifically, the executor is usually a Java virtual machine (JVM) and tasks are threads running on the JVM. Each Spark application has a set of executors scheduled by the platform's scheduler to different servers and their stay alive until all tasks of the application are competed. This two-level scheduling is adopted for two reasons. One is to cache a subset of data in memory to enable in-memory reuse of data across tasks in an executor in a fault-tolerant manner. The other is to significantly reduce overhead of launching tasks, which is critical for in-memory computing. In contrast, in a

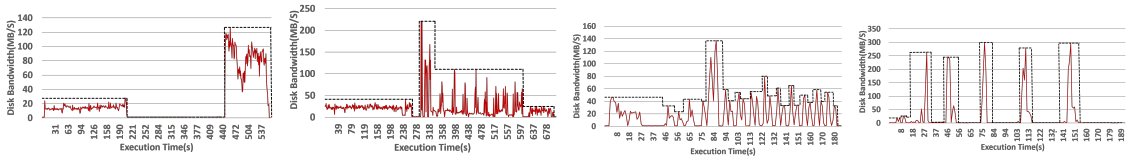
[5]The executor may be named differently. In the YARN environment, it is sometimes called container [181]. In the paper introducing Spark, it is called worker [203], while in the paper describing Mesos [**?**] and Spark Apache's official website [15], it is called executor.

Hadoop application each task runs on a dedicated JVM, which is scheduled by the platform's scheduler.

While there are two levels of scheduling for in-memory computing, the platform's scheduler plays a more performance-critical role by being responsible for resources allocation and sharing between applications. While executors take the place of tasks to become the platform scheduler's scheduling objects, the rationale made by existing schedulers on using peak resource usage to represent an object's varying resource demand is less likely to be valid. An executor runs multiple tasks belonging to different DAG stages and having possibly very different resource demands. Therefore, using the peak demand to represent different demands of a resource during the lifetime of an executor for resource allocation can cause serious resource fragmentation (or wastage), if we assume the resource is allocated according to the peak demand (e.g. in Tetris cluster scheduler [94]).

For a smooth run of tasks in an executor without interference from other executors belonging to other applications, it might be desired to have all four major required resources (CPU, memory, disk, and network) pre-allocated or reserved. Actually users only specify their resource demands on CPU (number of cores) and memory (size of memory) for an executor, which is implemented as a Java virtual machine (JVM). As these demands usually constitute the bottom line of meeting user's requirement on service quality, the requested resources are reserved at the time of executor scheduling. However, how to allocate disk and network to executors or tasks is also critical to applications' performance and system's efficiency, especially considering their highly variable demands.

Our objective is to dynamically adapting the resource configuration for applications of big data systems running on clusters, guarantee the resource allocation for each application match their multi-dimensional resource demand such as CPU,Memory,Network and Disk, while avoiding resource over-allocation and fragmentation.In addition, optimize performance through schedulers, storage, memory resource management aspects for big data system such as Spark and Hadoop in cluster computation.



(a) K-means (b) SVM (c) Pagerank (d) SVD++

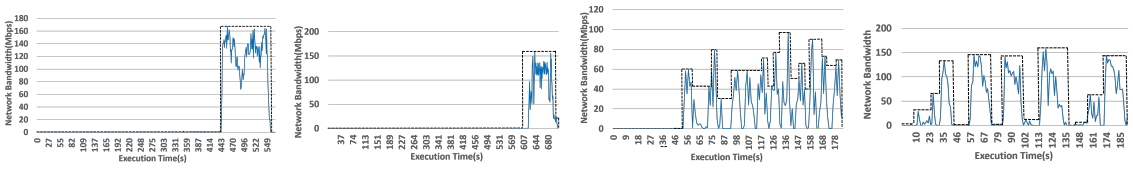
Figure 4.1: Disk bandwidth usages of four Spark benchmarks (K-means, SVM, PageRank, and SVD++). DAG stages are marked with dotted lines.

Motivation

To illustrate the potential efficiency loss, we use four Spark enbenchmarks and corresponding input data generators available in SparkBench [123], a public available Spark specific benchmarking suite, to reveal their executors' resource demand variations. Among the four benchmarks, two (K-means and SVM) represent machine learning workloads, and the other two (PageRank and SVD++) represent graph computation workloads.

- *K-means* is a machine learning workload clustering adataset into K clusters.
- *SVM (Support Vector Machine)*, is a machine learning classifier workload analyzing data and recognized patterns of high dimensional feature spaces while efficiently conducting non-linear classifications.
- *PageRank* is a graph computation workload ranking website pages and estimating their importance.
- *SVD++* is a graph computation collaborative filtering workload improving the quality of recommendation system based on the users' feedbacks.

Figures 4.1 and 4.2 show the disk and network bandwidth demands of the four Spark benchmarks (Spark 1.5.0) on Hadoop Yarn 2.4.0, respectively. Each executor is exclusively run on a server of 24 cores, 32GB of memory, three 7200 RPM disk drives, and 1Gbps NIC. It is obvious that for both disk and network usages the amount of bandwidth requested varies from almost 0 MB/s to around 300MB/s for disk or around 160MB/s for network. Their



(a) K-means (b) SVM (c) Pagerank (d) SVD++

Figure 4.2: Network bandwidth usages of four Spark benchmarks (K-means, SVM, PageRank, and SVD++). DAG stages are marked with dotted lines.

very low resource demands can stay for more than half of some executors' lifetimes, such as for network usages of K-means and SVM, while their peak demands are still very high, such as around 160MB/s. Should the resources be allocated according to peak demands, they would be significantly wasted due to the serous fragmentation. Even worse, starvation may occur on applications with both high peak network and disk demands as servers may not have available resources to meet both peak demands simultaneously (even though such an availability is not necessary). On the other hand, if they were not pre-allocated, executors on the same server may simultaneously experience high demand on the same resource and cause resource over-allocation. This can lead to severe interference (disk seeks or network incast) between the executors, which can sharply degrade applications' performance.

It is necessary to take resource variation of executors into their scheduling decision so that both resource fragmentation and overallocation can be minimized. This is a highly challenging issue considering even scheduling objects with constant resource demands (e.g., using peak demands) can be NP-hard [94].

Fortunately, recent studies on large-scale data-parallel systems have revealed that most applications in production clusters have recurring characteristics, with predictable future resource demands and mostly constant execution time in each DAG stage for given CPU cores and with sufficient memory [42, 57, 83, 94, 110]. To illustrate this, in addition to the aforementioned four benchmarks, we select another six Spark benchmarks. Three of them (LR, TriangleCount, and TeraSort) are from SparkBench [123], and the other three (WordCount, Sort, and Grep) are from BigDataBench [188].

- *Logistic Regression (LR)* is a machine learning classifier benchmark to predict continuous or categorical data.
- *TriangleCount* is a fundamental graph analytics counting the number of triangles in a graph to detect spam or hidden structures in web pages.
- *TeraSort* is a sorting benchmark using map/reduce to sort input data into a total order.
- *WordCount* reads Wikipedia text entries as input, and counts how often words occur.
- *Sort* is a benchmark designed for sorting the words from a Wikipedia dataset.
- *Grep* is a benchmark filtering and finding the specified words from a Wikipedia dataset.

Foe each of ten benchmarks, we supply 9 setups, which are formed by three different CPU core numbers for each executor (one, three, and five) and different different input dataset sizes (small, medium, and large). The dataset sizes for each benchmark and categories are shown in Table 4.6. Each of the setups run five times with different input datasets (of the same size). For each of the five runs in a dedicated cluster of 16 nodes, we collect each stage's start time and peak disk/netowrk bandwidths of an executor and compute their relative standard errors over the five runs. Figure 4.3 plots the errors with CDF (cumulative distribution function) curves. As shown, the relative errors are mostly smaller than 10%. Though dataset has a potential to affect executor's behavior, such as number of iterations to reach a convergence in machine learning applications, the impact is small. More importantly, each stage's start time is very stable (with a 5% or smaller relative standard error),

Because usually the same setup (CPU cores for each executor and input dataset size) remains in use for an application for an extended time period [42,83,110], profiling results on stage start time and peak resource demands of one run is sufficient for an executor scheduler to make an informed decision. When an application constantly changes its setup, we adopt a supported vector machine (SVM) with linear regression technique, and feed results from 25

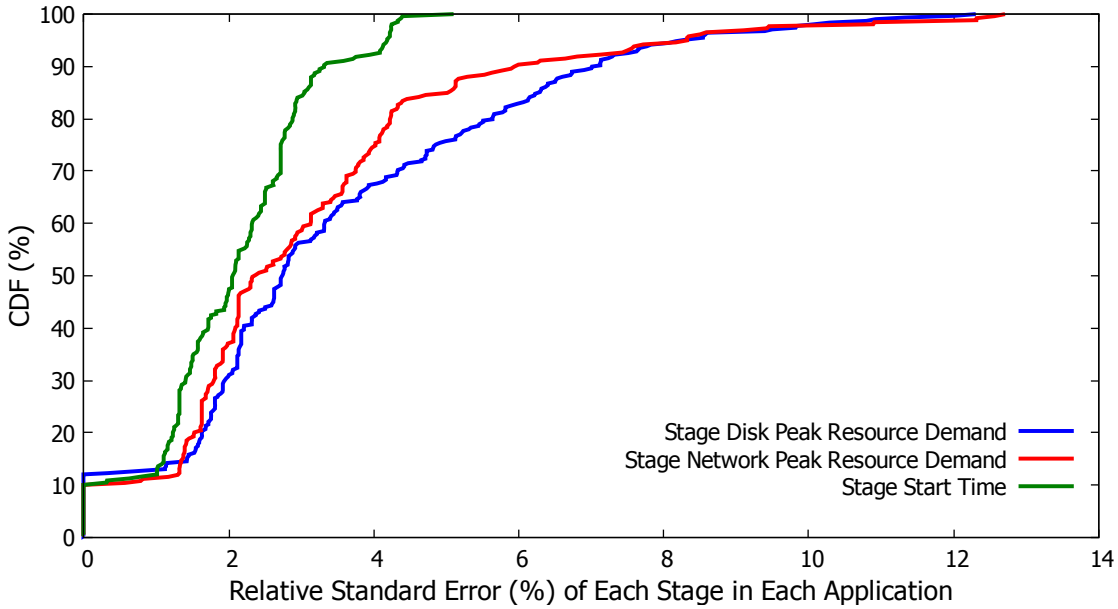


Figure 4.3: Relative standard errors of disk/network bandwidth and stage start time over the five runs of each of 10 benchmarks with different setups on CPU core and input size. Each run uses a different input dataset.

Benchmarks	SVM	KMeans	LR	PageRank	SVD++
Large Input Dataset	38.3G	21.9G	37.1G	4.0G	365.6M
Medium Input Dataset	19.2G	10.9G	18.5G	1.9G	163.3M
Small Input Dataset	9.6G	5.5G	9.3G	933.1M	78.1M
Benchmarks	TriangleCount	Terasort	WordCount	Sort	Grep
Large Input Dataset	364.7M	37.3G	44G	44G	44G
Medium Input Dataset	167.2M	18.6G	22G	22G	22G
Small Input Dataset	86.5M	9.3G	11G	11G	11G

Table 4.6: Three categories of input dataset sizes for each of 10 benchmarks.

profiling runs covering representative setups into the machine to build the prediction model. The model then can take in a new setup about CPU cores and dataset size) and produce its predicted stage start time and peak resource demands. Because changing CPU core count and input size usually does not lead to disruptive change of an executor's behaviors, the model consistently provides high-quality prediction (mostly less than 10% errors).

With the knowledge on an executor's peak disk and network demands at any stage during its lifetime and on each stage's start time (and its duration), a scheduler can estimate future resource availability at any time frame in the near future and make an informed scheduling decision accordingly to minimize resource fragmentation and over-allocation. We design an executor scheduler, named *Prophet*, that selects an executor whose scheduling would result in the smallest amount of fragmentation and over-allocation. To accommodate unavoidable or unpredicted resource contention, Prophet backs off selected task(s) in an executor to adaptively remove the contention.

In summary, We make the following contributions in the paper.

- We identify a performance-critical issue about the executor scheduling on in-memory data parallel computing platforms. We show that without considering resource demand variation within an executor, we can hardly enable an effective scheduling. By showing stability and predicability of resource demands in an executor, we make it possible to take the dynamics on the resource demands into account.

- We design an online executor scheduler, Prophet, that adopts a greedy approach by choosing the currently optimal executors in terms of expected resource fragmentation and over-allocation to dispatch. It also dynamically avoids dramatic performance degradation due to severe resource contention with its task backoff mechanism.

- We have implemented Prophet on YARN and Spark 1.5. to support running Spark and evaluated it on a 16-server cluster. Prophet has minimized resource fragmentation while avoiding over-allocation, simultaneously improving cluster resource utilization,

minimizing application makespan and speeding up application completion time. Compared to Yarn's default capacity and fair scheduler, Prophet reduces the makespan of workloads by 39% and the median job completion time by 23%.

Design of *Prophet*

As an executor scheduler, in addition to its main objective of minimizing resource fragmentation and over-allocation, Prophet has two other objectives. One is fairness across applications, and the other is load balance across servers running applications, In the scheduling, all arriving applications will be placed into a waiting queue. When an application is submitted, its required CPU, memory, and number of executors are specified. When there are applications whose specified resource demands can be met by currently available resources in the cluster, Prophet greedily chooses one that results in minimal fragmentation and over-allocation for dispatching. Then the required number of executors are created on different servers. Note that for load balance across servers in an application's execution, Prophet always creates the required number of executors at the time when the application is scheduled. It does not create executors fewer than the required ones when resources are not sufficient. Otherwise, if executors are allowed to increase, newly created executors will all request data from existing ones and make them become performance bottleneck. For fairness and avoiding starvation, Prophet chooses an application for scheduling from a subset of pending applications that have waited for the longest time (by default 50% of all pending ones). Each application is also assigned a deadline when it arrives at the queue. It will be scheduled immediately when its deadline is passed. The deadline can be assigned according to current average waiting time, such as three times of its average.

Prophet's Scheduling Algorithm

Prophet's scheduling algorithm is designed with assumption that future peak resource demand of an executor, either one that has been scheduled and is running or one that is candidate for scheduling, is known (or can be predicted). By knowing demands of executors

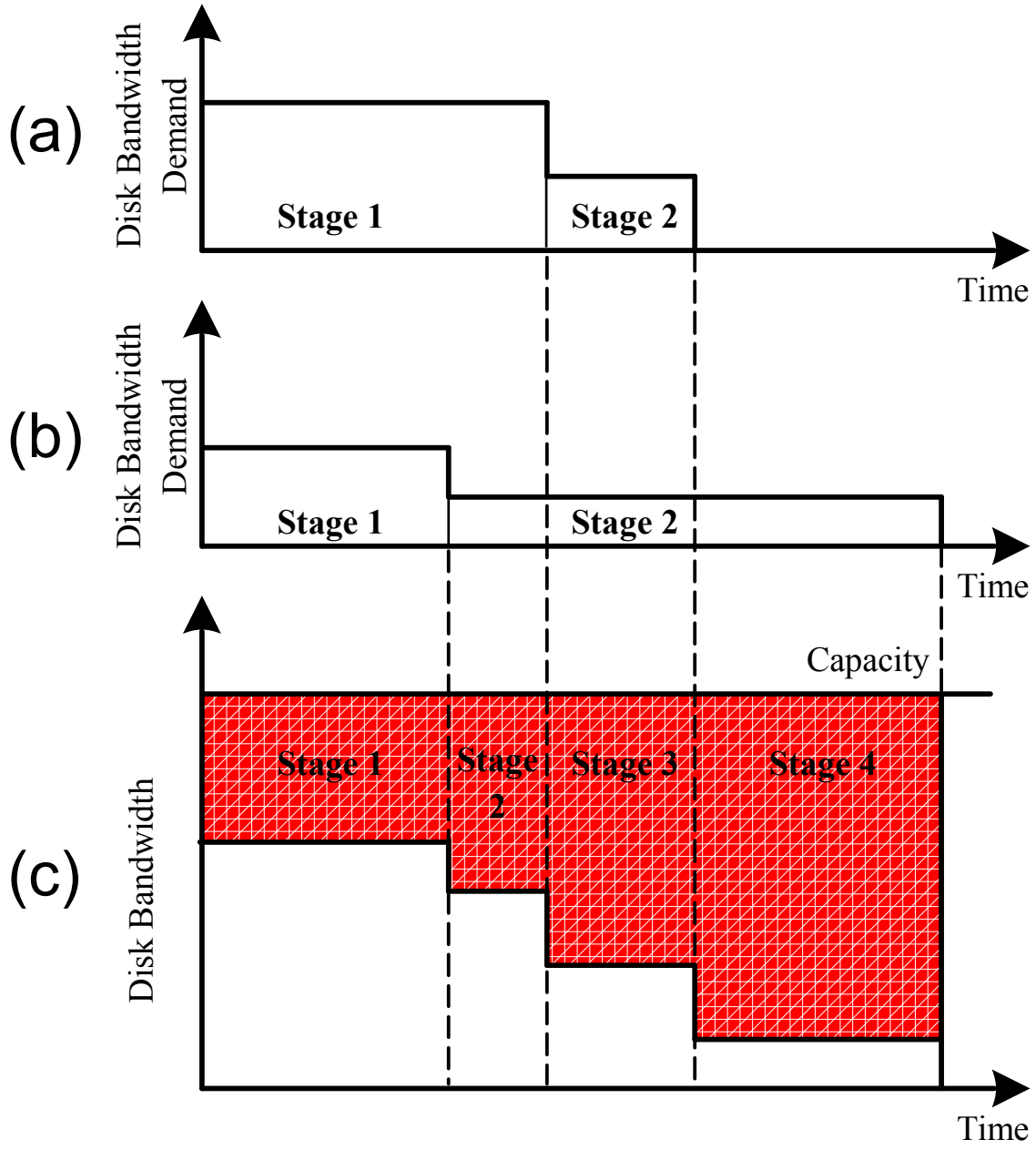


Figure 4.4: Illustration of predicting available disk bandwidth. With known demands on disk bandwidth from executors (see (a) and (b), the shaded area in (c) between their combined demand and the disk's capacity represents the disk bandwidth to be available.

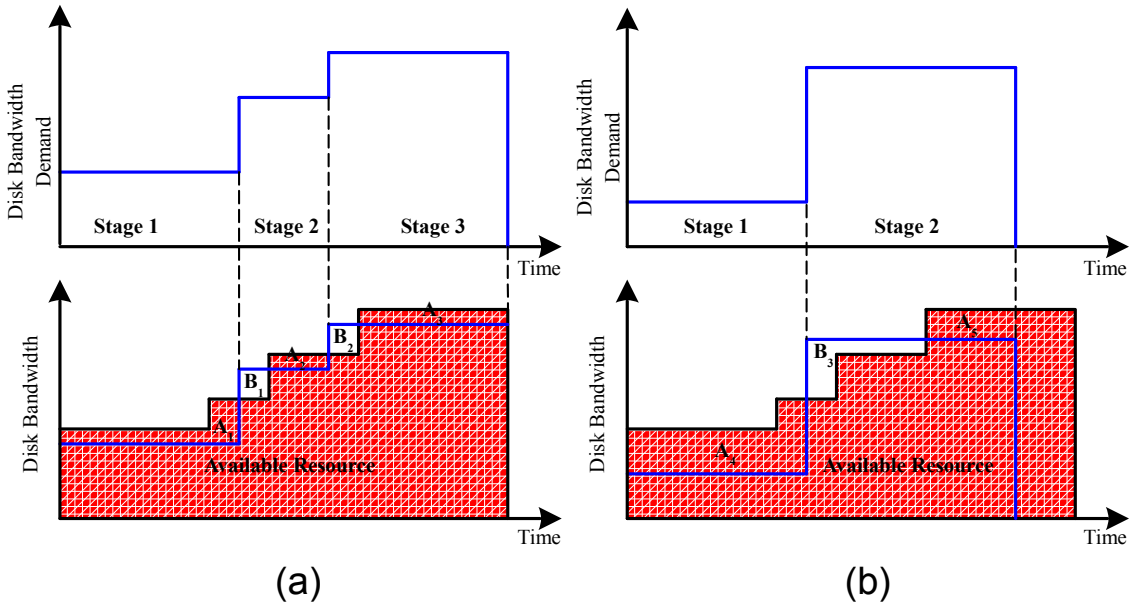


Figure 4.5: Illustration how fragmentation area (FA) and over-allocation area (OA) of disk bandwidth are formed for two executors. For each executor (see (a) or (b)), the up graph shows its demand on disk bandwidth, and the bottom graph shows the demand and the available disk resource (shaded area computed in Figure 4.4) overlap with each other to form FAs, such as A_1, A_2, and A_3, and OAs, such as B_1 and B_2.

currently running at a server, Prophet can compute how much the resource would be still available in the near future. This is illustrated in Figure 4.4 for disk bandwidth of a server with two executors being scheduled on it. In the shown example, each executor has two stages of distinct disk bandwidth demands. However, their combined effect leaves the available resource of four distinct values, or four *resource availability stages*. At this time we have two candidate applications' executors for Prophet to decide which one to schedule, as shown in Figures 4.5(a) and (b), respectively.

If only disk bandwidth is considered, Prophet needs to examine the future fragmentation areas (FAs) and over-allocated areas (OAs) in Figure 4.5. FA or OA refers the area between the two lines for available bandwidth and the demand in the figure. If available bandwidth is larger than the demand, it is FA, such as $A_i (i = 1, 2, ...5)$. Otherwise, it is OA, such as $B_i, (i = 1, 2, 3)$.. A good scheduler should minimize the two areas, FA represents wasted resource and OA represents resource contention and performance degradation. In this example, Prophet will schedule executor in Figure 4.5(a), as it has much smaller aggregate

C_i^r	capacity of Resource r on Server i
$P_k^{r,j}$	Peak demand of Resource r from Executor j at its Stage k
$A_s^{r,i}$	Available Resource r of Server i at resource stage s
D_s^i	Duration of resource stage s
$t_k^{i,start}$,$t_k^{i,end}$	Start and end times of stage k at Executor i
$T_s^{i,start}$,$T_s^{i,end}$	Start and end times of resource stage s at Server i

Table 4.7: Notations in the Prophet's scheduling algorithm

FA/OA area than that in Figure 4.5(b). This example also indicates a scheduler unaware of future resource demands and availability might schedule the executor in Figure 4.5(b) leading to much worse performance.

To formally describe the design of the scheduling algorithm, we introduce a number of notations as shown in Table 4.7. Note that in the notations, quantities about duration and times (D_s^i, $t_k^{i,start}$, $t_k^{i,end}$, $T_s^{i,start}$, and $T_s^{i,end}$) are not defined specifically for certain resource. Instead, they are specified according to change of stages for any resources.

To quantify fragmentation and over-allocation for candidate application's executors, we might simply add FA or OA of an executor's every stage, and consider the sum as the executor's fragmentation score or over-allocation score, or $Fscore$ and $Oscore$ in short, respectively. However, for an executor of many stages, prediction on demands and resource availability at the earlier stages, or those closer to the current time, is usually more accurate than that on later stages, as the latter is more likely to be influenced by unaccounted noises. To this end, we give earlier stages a higher weight. Specifically, if the executor has n stages,

the weight for Stage i ($i = 0, 1, ..., n-1$) is $w_i = 1 - i/n$. Therefore, the two scores can be computed for Resource r as below.

$$Fscore_r = \sum_k \left\{ \sum_s \left[\left(A_s^{r,i} - P_k^{r,j} \right) * D_s^i \right] * w_k \right\}$$
$$\text{for any} P_k^{r,j} < A_s^{r,i}, \text{as long as} \tag{4.1}$$
$$\begin{cases} T_s^{i,start} \geq t_k^{j,start} \\ T_s^{i,end} \leq t_k^{j,end} \end{cases}$$

$$Oscore_r = \sum_k \left\{ \sum_s \left[\left(P_k^{r,j} - A_s^{r,i} \right) * D_s^i \right] * w_k \right\}$$
$$\text{for any} P_k^{r,j} > A_s^{r,i}, \text{as long as} \tag{4.2}$$
$$\begin{cases} T_s^{i,start} \geq t_k^{j,start} \\ T_s^{i,end} \leq t_k^{j,end} \end{cases}$$

In theory, to minimize both fragmentation and over-allocation in the selection of applications for scheduling, we might simply use the sum of the two scores as the metric for the selection. However, resource over-allocation can cause contention among executors and slow down all involved ones. More seriously, the slowdown may lead to more idleness (fragmentation) of other resources. To address the issue, we give $Oscore$ a higher weight when computing the overall score.

$$OverallScore_r = (1 - \eta) Oscore_r + \eta * Fscore_r \tag{4.3}$$

In our prototype, we set η as 0.3 by default, which is experimentally determined to balance the risks of severe performance degradation and wastage of resources. We leave a comprehensive study of this factor as a future work.

While for each resource (disk or network resources) Prophet can compute an overall score, for all resources it obtains a vector of overall scores for an application's executor. To convert the vector into one-dimension quantity for comparison across candidate applications, we use the Euclidean norm of the vector. Accordingly Prophet selects application whose executors have the smallest norm. The scheduling algorithm is described in Algorithm 1.

Algorithm 1 Prophet Scheduling Algorithm

Require: Denote the Available Resource of Server i as: AR_i
Require: Denote the Resource Demand of Executor j as: RD_j
Require: Denote the Overall Score Vector of Executor j as: OSV_j
1: **When Executor** j **of application** p **is added to queue**
2: Offline Predictor predicts its $P_k^{r,j}$
3: **When a hearbeat is received from Server** i
4: **while** there is $AR_i\{cpu, memory\}$ on Server i **do**
5: **for** each Executor j in the queue **do**
6: **if** $RD_j\{cpu, memory\} < AR_i\{cpu, memory\}$ **then**
7: Acquire latest predicted $P_k^{r,j}$, $A_s^{r,i}$, D_s^i
8: Compute OSV_j of Executor j
9: **else**
10: ISV_j = NULL
11: **end if**
12: **end for**
13: Launch Executor j whose norm of OSV_j is minimum and not empty on Server i
14: Update $AR_i\{cpu, memory\}$, $A_s^{r,i}$, D_s^i of Server i
15: **end while**

Ameliorating Contention with Task Backoff

While Prophet attempts to avoid expected over-allocations, there still can be unexpected ones or expected minor ones turn out to be major over-allocation. As we have indicated in Section 1, severe over-allocation leads to intensive interference. For disk and network, such an interference can cause the effective bandwidths to be much lower than their normal peak ones due to reasons such as random access and incast, respectively. When inter-

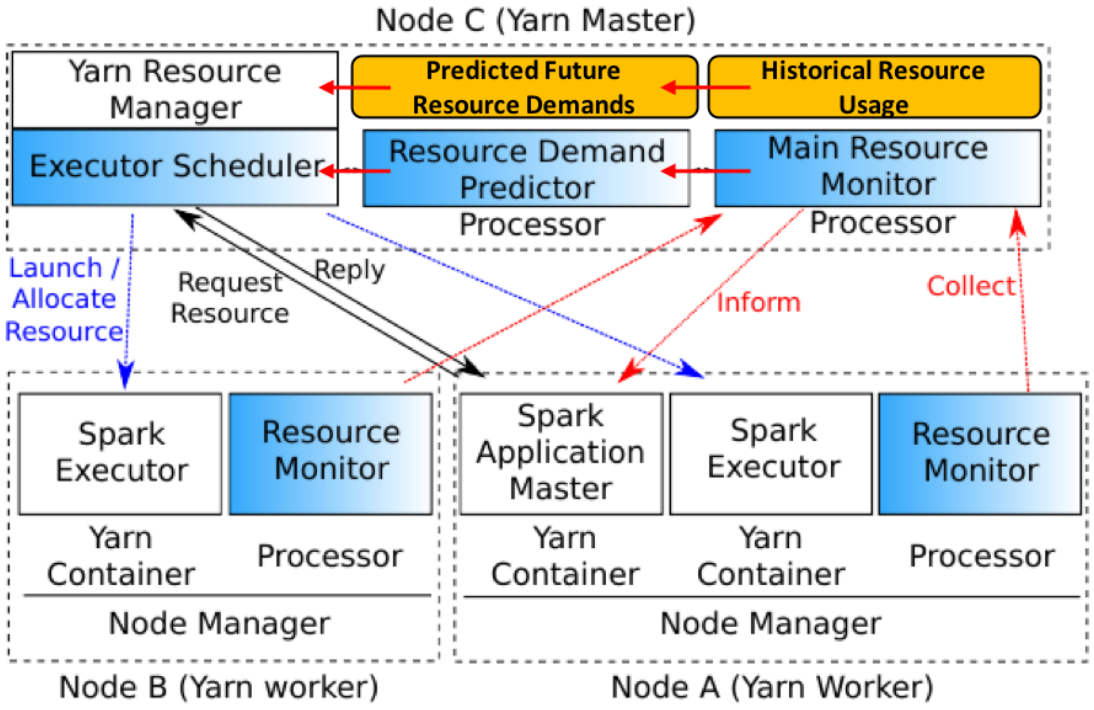


Figure 4.6: The framework of Spark applications running on a Yarn cluster, in which Prophet modules are included (shown as shaded boxes).

ference essentially blocks tasks of an executor from moving forward, the executor's reserved CPU cores and memory are also wasted. To address the issue, Prophet has an emergency handling mechanism built in the Spark's task scheduler. When it is observed that effective disk or network bandwidth is substantially lower than their peak one while it stays busy to serve requests at a server, a serious over-allocation is detected at the server. Prophet will examine the profiled resource demands of each executor on the server and identify ones that are most likely to overuse the contested resource. Then it activates a backoff mechanism by reducing number of tasks dispatched to the executors until the effective bandwidth approaches the peak one or the resource is not busy anymore. Note that the mechanism is enabled only temporarily, usually lasting for only a few task scheduling rounds, as an overaction could compromise utilization of CPU and memory.

Implementation and Evaluation of Prophet

We have implemented Prophet executor scheduler on Hadoop YARN 2.4.0 and task backoff mechanism in Spark 1.5.0. In addition, we implemented a resource usage monitor on each server to detect over-allocation. In this section, we will provide implementation details, system setups for performance evaluation, and evaluation results.

Prophet's Implementation

Figure 4.6 depicts where the Prophet modules are situated in the framework of Spark applications running on a YARN cluster. Yarn's cluster-wide resource manager is responsible for receiving executors' resource request from each Spark application master, and communicating with node manager on at each server to decide if there are sufficient resource to meet the resource request. If yes, corresponding resources will be allocated, and the Spark application master and its executors would be running as containers on servers managed by node manager. On this framework we made a few instrumentations.

- The resource demand predictor runs as a separate process on the Yarn's master node hosting its resource manager. In the background it continuously learn and predict executors' resource demands.

- The executor scheduler is enabled as Yarn's plug-in scheduler. It communicates with the predictor before making its scheduling decisions..

- The task backoff mechanism is implemented in Spark's scheduler, which runs with each Spark application master and communicates with the resource monitor to decide if task backoff should be enabled for an application and if yes, for how long.

- The main resource monitor running as a separate background process on the master node communicates and collects information from those resource monitors running on worker nodes.

These changes are lightweight. They do not increase complexity and scalability of Yarn's scheduling framework. The profiling and prediction workload is run in the background.

Experiment Setup

We deployed our implementation of Prophet in Hadoop Yarn 2.4.0 and Spark 1.5.0 on a 16-server cluster. Each server has 24 cores, 32GB of memory, three 3.5TB 7200 RPM disk drives with a 110MB/s peak bandwidth for each one. It has a 1Gbps NIC and runs Linux 3.16. We use the 10 benchmarks that were described in Section 1. In the same as we ran the benchmarks in Section 1, for each benchmark, we vary its input size as listed in Table 4.6 and its CPU core count (1, 3, or 5). So essentially we have 90 applications to run in the evaluation. Each application is submitted to the system at a time randomly picked between 0 second (experiment start time) and 1200 seconds.

The input dataset of the machine learning and graph computation benchmarks (K-means, SVM, Pagerank, SVD++, LR, and TriangleCount) kept in memory as Spark RDD abstraction to support the later parameter vector calculation, update and broadcast of each iteration.

We compare Prophet to three state-of-the-art Spark scheduling algorithms implemented in Yarn, which are Dominant Resource Fairness(DRF) scheduler [87], the capacity scheduler(CS) [27, 28] and Tetris [94]. The capacity scheduler is designed to achieve fairness on memory allocation based Hadoop's slot-based resource management, while DRF considers fairness for both CPU and memory. In addition to CPU and memory, Tetris considers network and disk bandwidths. It tries to efficiently pack tasks/exectors when resources are sufficient to accommodate their peak demands. Nevertheless, to the best of our knowledge, all the existing schedulers are designed for task-grained scheduling without considering the resource demand variation within scheduling objects.

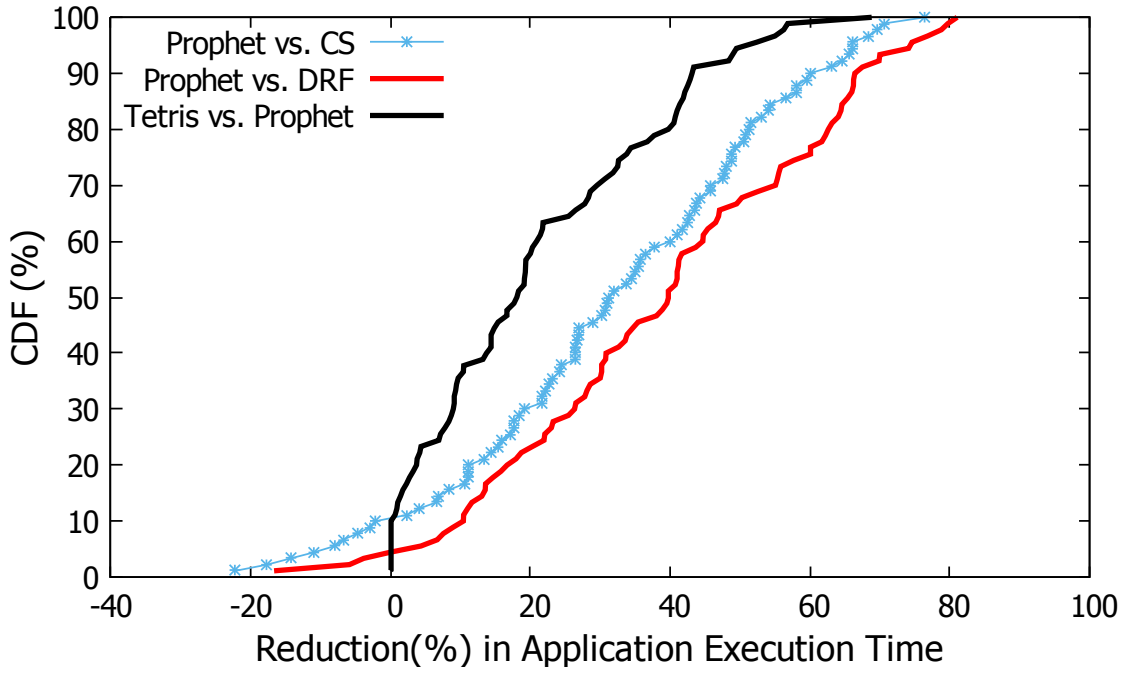


Figure 4.7: CDF curves for reductions of execution times by Scheduler X over Scheduler Y, shown as X vs. Y. X can be Prophet and Tetris, and Y can be CS, DRF, and Prophet.

Experiment Results

Figure 4.7 shows cumulative distribution function (CDF) curves of application's execution time reduction by Prophet over CS, by Prophet over DRF, and by Tetris over Prophet. An application's execution time is measured from the time its executor are scheduled to its completion. For example, the figure shows that there are 50% of applications whose execution times are reduced by 31% or less they are scheduled by Prophet over those by CS, reduced by 40% or less by Prophet over those by DRF, or by 18% or less by Tetris over those by Prophet.

While CS consider only memory and DRF considers only memory and CPU, it is a surprise to see Prophet generally performs better than them in terms of execution time. Prophet uses prediction and task backoff to avoid over-allocation of disk and network bandwidth. In contrast, CS and DRF experience (much) more serious interference between executor at a server, and take longer time to complete. However, it is interesting to have these two observation. First, There are a few percentage of applications whose CS/DRF execution

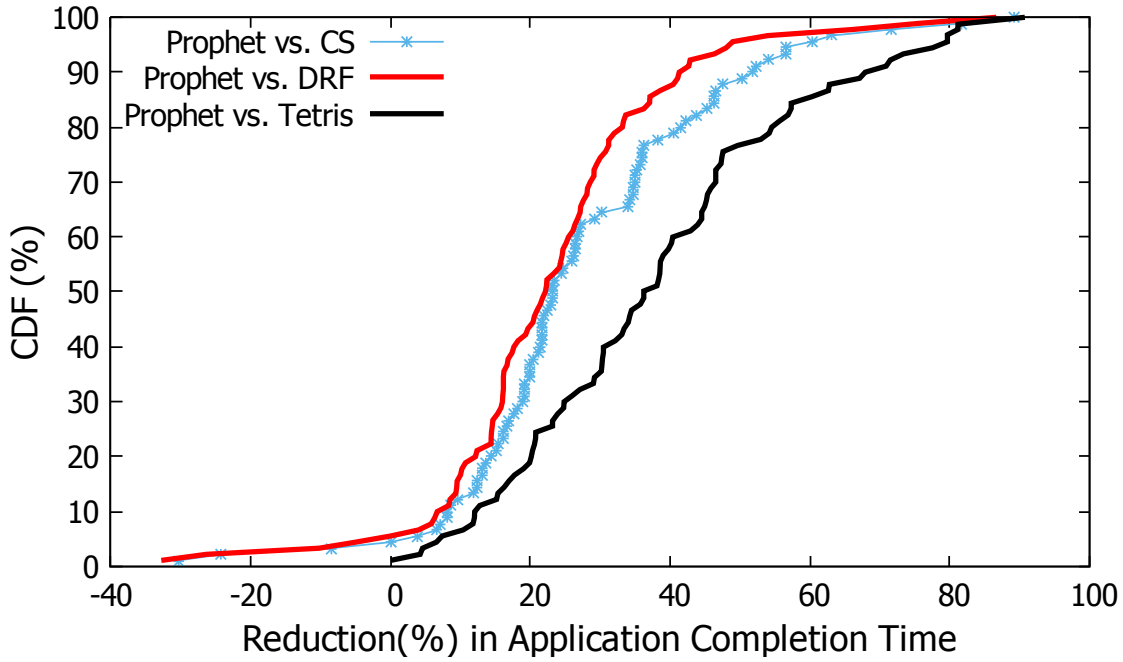


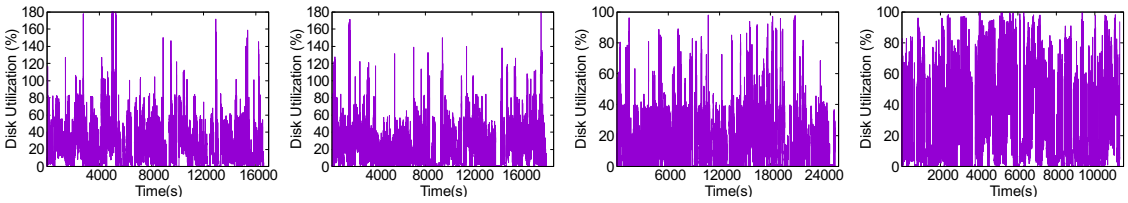
Figure 4.8: CDF curves for reductions of completion times by Scheduler X over Scheduler Y, shown as X vs. Y. X is Prophet, and Y can be CS, DRF, and Prophet.

Scheduler	CS	DRF	Tetris	Prophet	Propeht w/o Backoff
Makespan (s)	16604	18369	25537	11290	15707

Table 4.8: Makespans produced by various schedulers for running the 90 applications.

times are shorter than those of Prophet. This is because Prophet also makes effort to reduce fragmentation, which may increase risk of interference. In such cases, CS and DRF may win. Second, Tetris consistently has a shorter execution time than Prophet. Execution time can only compromised by over-allocation, and not by fragmentation. Tetris uses an executor's peak resource demands for allocation. So it is less likely to have an over-allocation. However, Prophet also needs to consider reducing fragmentation, which does not help with execution time. However, a metric more meaningful to users is completion time, which is measured from the time when the application is submitted to its completion.

Figure 4.8 shows CDF curves of application's completion time reduction by Prophet over CS, by Prophet over DRF, and by Prophet over Tetris. For this metric, Prophet is better than Tetris. For example, there are 50% of applications whose completion times are reduced by 36% or more, and 10% of applications whose times are reduced by 12% or



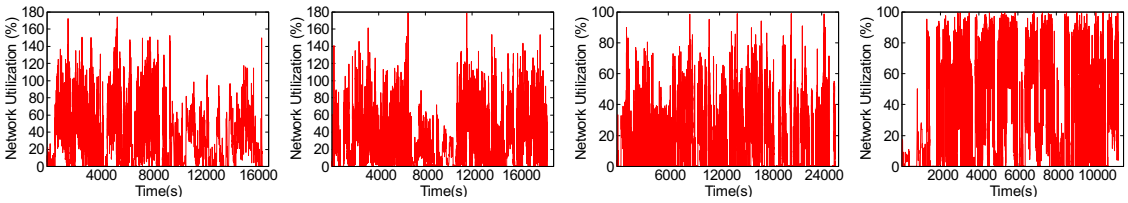
(a) CS (b) DRF (c) Tetris (d) Prophet

Figure 4.9: Disk utilizations during running 90 applications under various schedulers.

more. If we read makespans of the executions under different schedulers listed in Table 4.8, it is clear that Prophet is much better than other schedulers. The makespan measures the total time period used to complete all the 90 applications under a scheduler. It is directly correlated to the system's resource efficiency. Prophet reduces the makespan by 32%, 39%, and 56% compared to CS, DRF, and Tetris, respectively. The reduction over Tetris is the most significant, while Tetris produces the best application execution time.

These results reveal the strength of Prophet, which is aware of varying future resource demands and takes them into scheduling decision. If a scheduler does not have the knowledge, it has two options. One option, that is taken by Tetris, conservatively uses executors' peak demands for scheduling. While this minimizes possibility of over-allocations and helps with the execution, it would leave significant fragmentations, which compromises resource efficiency. Therefore, it is expected to see that Tetris has the worst makespan. The option, that is taken CS and DRF, simply does not consider disk and network demands in the scheduling. So they are more likely to have serious interference than Tetris and Prophet. That is why their execution times are worse. In the meantime, they are less likely to have fragmentations than Teris. That is why their makespans are shorter than Tetris. By explicitly considering varying resource demands, Prophet can address both over-allocation and fragmentation issues.

To reveal insights on how disk and network resource bandwidths are actually consumed, we use their utilizations under the four schedulers in Figures 4.9 and 4.10. The utilization is the ratio between aggregate demands on a resource from all executors at a



(a) CS (b) DRF (c) Tetris (d) Prophet

Figure 4.10: Network utilizations during running 90 applications under various schedulers.

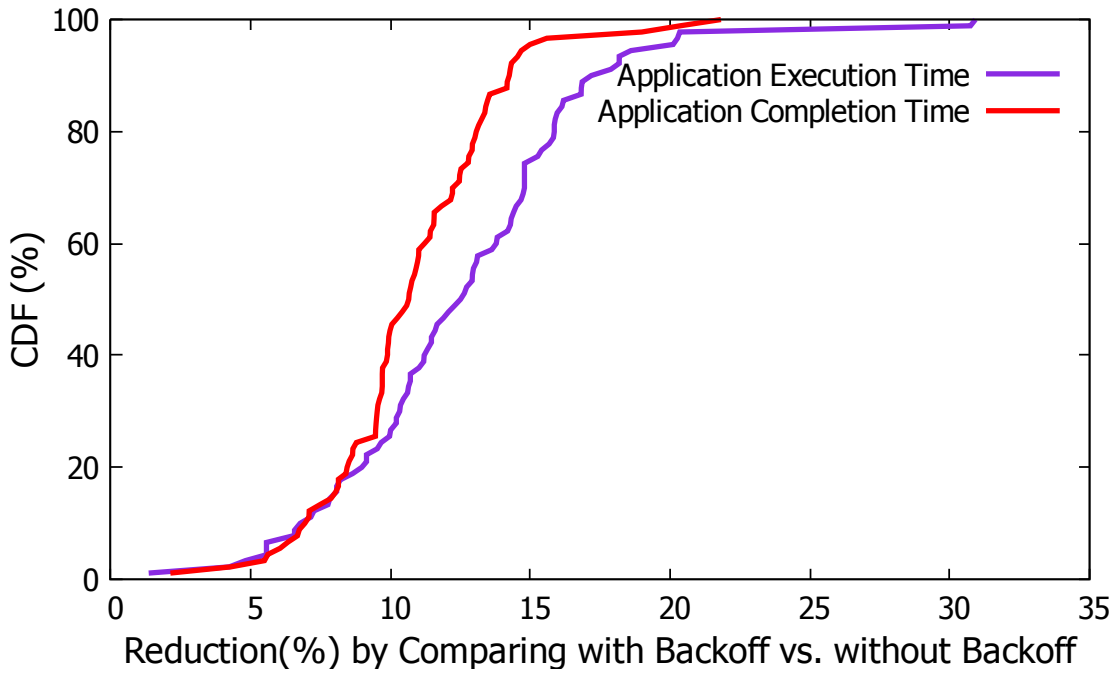


Figure 4.11: CDF curves for reductions of execution and completion times by Prophet over Prophet without task backoff mechanism.

server and the server's capacity of the resource. As shown, for CS and DRF, there are many significant over-allocations, which suggests that much lower effective (disk or network) bandwidth. In contrast, Tetris and Prophet have little over-allocation. However, there are much more high utilization values in Prophet than those in Tetris (for either disk or network) , indicating that Tetris has much more serous fragmentation issue.

While Prophet has two components to achieve its scheduling objectives, we would like to see the contribution made by each of the components (prediction-based scheduling policy and task backoff mechanism). Figure 4.11 shows CDF curves for reductions of execution and completion times by Prophet over Prophet without task backoff mechanism. While the

prediction-based scheduling policy tries to minimize both fragmentation and over-allocation, the task backoff mechanism basically addresses only the over-allocation issue. Application execution time is directly by affected by interference caused by over-allocation. With the backoff mechanism, applications' execution time is more significantly compromised than the completion time. This experiment also reveals that in a shared execution platform, it is necessary to have a backup mechanism to keep the system from unavoidable or unpredictable resource usages.

Summary

We have demonstrated that existing task schedulers are not suitable for scheduling executors with time-varying resource demands on an in-memory data-parallel computing platform, such as Spark. They suffer from serious over-allocation and fragmentation problems and can substantially compromise application performance and system resource utilization. Motivated by observations on recurring resource usage patterns in the platform, we propose a scheduling algorithm, Prophet, to learn and leverage the patterns in the executors' scheduling. In particular, Prophet predicts detailed resource availability at a server and varying demands from executors in the near future, and takes efforts to make the demands best match the available resources. This will help with both the application performance and system efficiency. To be robust, Prophet has a task backoff mechanism to accommodate unexpected over-allocation.

We have implemented Prophet on Yarn and Spark. Extensive experiments with publicly available benchmarks show that Prophet could reduce makespan by up to 39% and median application completion time by 23%, compared to Yarn's default capacity and fair scheduler.

CHAPTER 5 Large-scale Datacenter Co-location Techniques

Introduction

we concluded several critical datacenter technique challenges and root causes in Alibaba that lead to severe inefficiency during past years. We also introduce the feasibility and challenges of an enterprise-wide co-location techniques at large scale to effectively improve datacenter efficiency. It involves in the evolutions of infrastructures in Alibaba during recent 5 years.

Challenges of Datacenter Inefficiency

C1: Over-provision and over-purchases in terms of stringent SLAs and extremely peak traffic bursts. In Alibaba, the dominant user-facing products are online latency-critical (LC) services such as online shopping, advertising, search, financing and online payment of eCommerce. They require stably low latency (mostly within 100 ms as shown in Figure 5.3) under stringent service level agreements (SLAs) (variances within 99%) to prevent abnormally terminated transactions or unacceptable data loss due to services time out. These faults are intolerable for eCommerce workloads and would lead to enormous economic losses [31, 143, 158].

Moreover, there are several famous annual promotion events such as "Double 11" (Nov.11) or "618" (Jun. 18). They bring 10+ times higher traffics, loads and transactions in a few minutes. Figure 5.1 shows the peak transanction per second (TPS) of "Double 11" in recent years. We observed the peak TPS rapidly increases since 2009 and is almost double in 2017. It incredibly reached 325000 and might be double in 2018. The TPS and resource requirements of "Double 11" are hundreds of times higher than day-to-day periods.

Figure 5.2 and Figure 5.3 display the comparisons of total query per second (QPS) and response time (RT) of critical eCommerce services like buy, cart and tradeplatform between "Double 11" and daily peak periods. The average QPS of core components (Http,

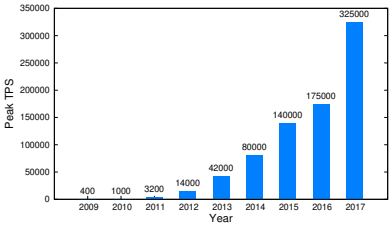


Figure 5.1: Peak TPS of Double 11 in recent years.

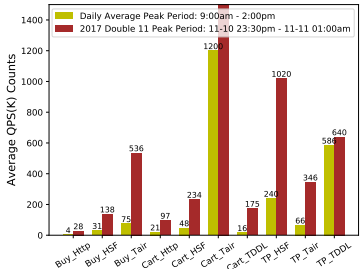


Figure 5.2: QPS comparisons between Double 11 and daily core eCommerce services like Buy, Cart and TradePlatform (TP).

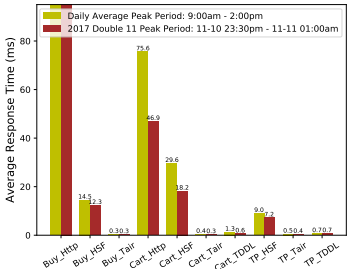


Figure 5.3: RT comparisons between Double 11 and daily core eCommerce services like Buy, Cart and TradePlatform (TP).

HSF, Tair, TDDL[6]) of critical services in "Double 11" are always 5 to 10 times higher than daily ones. It incredibly reached 4600,000 QPS for Tair (memcache) of cart service, 1020,000 QPS for HSF (RPC) and 640,000 QPS for TDDL (database) of tradeplatform service. There are a bunch of capacity re-planning, load balance and auto-scaling techniques designed for "Double 11". They amortize the extreme TPS pressures and make QPS of services fall in an almost bearable scope. The average RT of service components should still be within 95% variances as those in daily periods, which are 0.3ms, 7.2ms and 0.7ms respectively. Strictly guarantee service stability within millseconds-level latency under such extreme load spikes while maintaining high resource efficiency is non-trivial.

In the past years, we planned capacities and resource demands according to peak traffic bursts and load spikes during big events in advance, and wastefully purchased extra massive servers to satisfy an annually rapid growth of peak traffics in "Double 11", leaving tremendous underutilized resources during the rest of the year. We also over-provisioned LC services separately on dedicated clusters to ensure stringent SLAs. It left spare resources of most clusters, whereas others suffered starving during peak periods in the unshared environment. The daily enterprise-wide average CPU utilization was even lower than 10% over

[6]In Alibaba, we develop high-speed industry RPC, memcache and database components. We contributed them to Alibaba Cloud and open source community. They are named HSF [4], Tair [23] and TDDL [24] respectively.

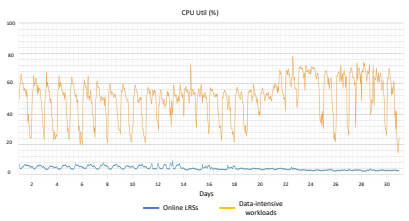


Figure 5.4: Aggragate average CPU utilization (%) of online LRSs (blue dash) and data-intensive workloads (orange dash) of two seperate clusters during one month.

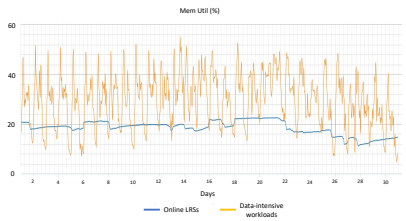


Figure 5.5: Aggragate average memory utilization (%) of both online LRSs (blue dash) and data-intensive workloads (orange dash) of two seperate clusters during one month.

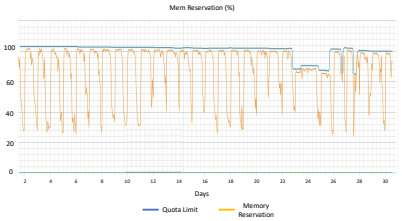


Figure 5.6: The ratio of memory reservation to quota limit (%) of data-intensive workloads of an offline cluster during one month.

past years, which significantly threatens ROI. Unrestrained scaling up with huge unnecessary costs is unaffordable and unsustainable as the rapid growth of business and datacenter scale.

The Motivation and Feasibility of Colocations

Recent studies showed that an effective approach to improve efficiency is to co-locate data-intensive workloads on the same servers of LC services to fully exploit their underutilized resources [57, 74, 75, 131, 140, 141, 185, 200, 207]. Due to above chanllenges and status of Alibaba datacenters, we leverage the benefits of colocation techniques to maximize datacenter efficiency. It includes two scenarios:

1. In day-to-day periods, we co-locate mixed types of workloads in a global shared resource pool. Data-intensive jobs fully utilize the spare resources of servers left by LRSs.

2. During big events, we lend capacity from data-intensive workloads to LRSs in a short spike period (one peak hour), to accommodate extremely peak traffic pulse bursts of LRSs while avoiding extra server purchases of **C1**.

The co-location techniques are naturally feasible due to four reasons as followings:

Complementary resource demands. Figure 5.4 and Figure 5.5 display the aggragate average CPU and memory utilization of two types of clusters during a month. Online LRSs constently operate at a low utilization of 10% CPU and 20% memory, whereas data-intensive jobs at average of 40% and 50% respectively with stable periodic variations. Over

70% of data-intensive jobs are short temporary queries that executed within 3 minutes. Differ from the common assumptions that long-running production jobs take up at least 70% of the cluster [69, 73, 86, 177, 185], data-intensive jobs that regarded as best-effort ones overwhemingly take advantage of most resources.

Virtually, LRSs reserved and hold 90% of available resources in online cluster, and leave 95% of them spare during day-to-day period. It is because schedulers always over-provision LRSs to handle spikes and strictly ensure SLAs while capacity planning is also difficult to be accurate. The amortize, load balance and affinity constraints aggrevate the inefficiency issues of most commercial production datacenters, leading to 6% to 12% utilization.

Accordingly, the stably low usages of LRSs and large proportions of best-effort jobs with high demands are naturally complementary, and could be efficiently co-located on the same servers without overloading in day-to-day period.

Complementary runtime characteristics. Table 5.9 displays the characteristics of two types of workloads. They are fully complementary to each other. By over-committing mechanism, data-intensive workloads could sufficiently utilize the spare resources reserved by LRSs during daily periods. Since they do not require real-time responses and are not sensitive to interferences, they are friendly to co-locations. Additionally, most data-intensive workloads are best-effort ones that are preemptible and could tolerate re-computations. They could efficiently and rapidly return resources and make spaces for LRSs by preemption and reclaimation when LRSs request scaling-up during contentions or traffic bursts periods. With the prerequisite of complementary characteristics and multiplexing of resources, we could guarantee high priority and stability of LRSs without wasteful over-provisioning under colocations.

Complementary diurnal usage patterns. Figure 5.6 exhibits the daily ratio of resource reservations to quota limit of data-intensive applications, which demonstrates obvious periodical diurnal usage patterns. Most daily recurring workloads like analytic

Table 5.9: Workload Characteristics

Characteristics	LRSs	Data-intensive Applications
Type	Production services	Batch jobs mostly
Priority	High and non-degradable	Best-effort and preemptible
Real-time Response	Yes	No
Latency Sensitive	Yes	No
SLA Requirement (Stability)	Strict	Loose
Recomputation	No	Yes
Load Spikes	Unpredictable and frequent	Predictable and few
Day-to-Day	Daytime high, midnight low	Daytime low, midnight peak
Big Events	Short pulse bursts	Degradable
Resource Usages	Daily low, events high	Constantly high

reports that reserve 90% of cluster resources are auto-submitted at 1pm and always finish by 8am as shown in Figure 5.6. On the contrary, peak traffic periods of all LRSs like eCommerce, O2O or digital entertainments always happen on daytime that started from 9am to 11pm, when the usage pressures of data-intensive jobs are slight. The complementary peak periods and diurnal usage patterns of two types of workloads naturally enable the time division multiplexing sharing of resources. It also implicitly mitigates the extent of co-located interferences during daytime.

Enable resources rent during big evets. The pulse bursts sustained only minutes to one hour after mid-night. By co-locate two types of workloads in the unified shared clusters, LRSs could borrow a majority of resources from data-intensive workloads during that short pulse periods. We are capable of temporarily degrading quota of batch jobs, and resume it in time when peak demands of periodical analytic jobs come. The short-term resources rent could make immeasurable savings that avoiding annual tremendous over-purchases, at the tiny costs of negligible impacts on batch makespans.

Accordingly, we could effectively boost datacenter utilization in day-to-day periods, and stand up to extreme pulse bursts during big events without extra purchases by workloads colocations.

Challenges of Co-locations

However, the performance interferences between colocated workloads caused by contentions of shared hardware resources still threaten service stability. Traditional QoS-aware (quality-of-service) scheduling and isolation techniques [57, 74, 75, 131, 140, 141, 185, 200, 207] are not sufficient for eCommerce workloads with strict SLA requirements of millseconds-level latency. We introduce several critical challenges of colocations.

C3: Ineffective traditional colocation techniques due to sensitive latency and long chains. Online services in Alibaba always have a long transaction chain that consists of dozens of cascaded services across diverse middlewares including web services, memcache, RPC and database. Figure 5.7 shows a complete shopping process (these services might not be directly linked.). The fundemental components such as memcache (Tair) or RPC (HSF) of core services like cart and TP are always visited 1200,000 or 240,000 times per second in daily requests as shown in Figure 5.2. The normal RT per request are within 0.4ms and 9ms. The tail latency [70] caused by slight interferences would incur service time out and is not tolerable for eCommerce workloads.

For example, if one slight server load spike occurs, the latency-sensitive and frequent access components like Tair running on it could be blocked by a large number of time out threads. Several cart services on that server would be time out, and the consequence is magnified several orders of magnitude to the block of HSF components for downstream services. It causes significant SLA violations and faults for hundreds of buy and tradeplatform services on other servers that depending on these upstream cart data. It eventually spreads tens of thousands of users and prevents them from creating orders normally or even makes duplicated payments. The prior isolation solutions that resolving web search [57, 74, 75, 122, 131, 140, 141, 148, 150, 180, 185, 207, 208] or social media [90] based industry scenes are not sufficient for long-chain and interference-sensitive eCommerce workloads in Alibaba. Moreover, the resource contentions would be amplificated several times during extremely high QPS of "Double 11". We need to ensure stability and strict SLAs

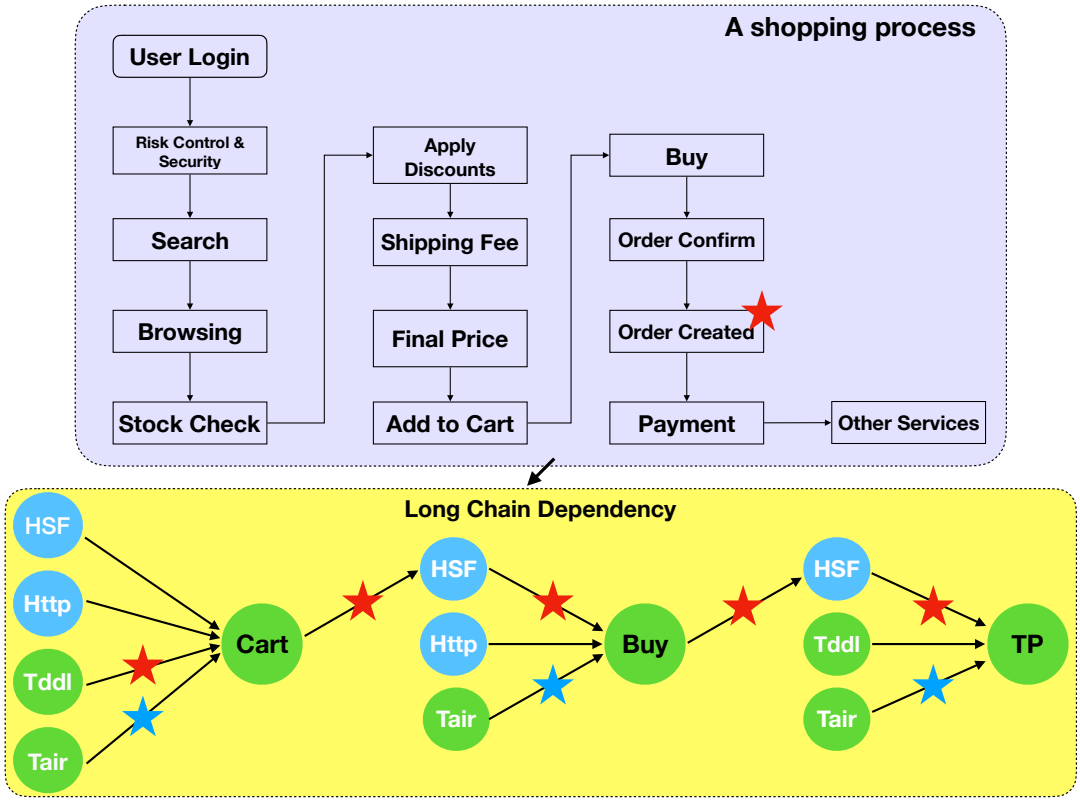


Figure 5.7: A complete shopping process consists of 200+ services. Pentagrams indicate the chain reaction of services time out.

of arbitrary service under such significant interferences in the long chain, which hinders the feasibility of traditional colocation techniques.

C4: Inefficient traditional colocation and resource provision techniques due to the extreme complexity and large scale of diverse workloads. Modern technique ecosystem of Alibaba is far more complicated than traditional stacks of eCommerce services or big data computations [14, 15, 71, 203]. We have a universal set of existing world-wide industry workloads as shown in Figure 5.8. We classify four main categories of workloads as followings:

Latency-critical long-running services (LC LRSs) serve online requests using long-standing (e.g. weeks or months) containers to achieve low latency. Alibaba has one of the largest-sacle LRSs in the world. There are more than 60000+ types of online services (e.g. buy, tp, cart..) ranging from 60+ business departments (e.g. Taobao, Alipay, Map..), which constitutes of dozens of middleware components (e.g. HSF, Tair, TDDL..). They are running inside daily millions of non-stopped containers [29,52] as stateless microservices [149] on hundreds of thousands of servers, and supporting the largest-scale eCommerce business in the world. They have strict SLA requirements of latency and are sensitive to performance interferences.

Data-intensive offline computing (DIOC) applications[7] typically take minutes to a few days to run to completion. There is a broad category of big data warehousing workloads ranging from MapReduce [14, 71], DAG-based processing [107, 203], MPI [97], graph computing [93,119,133,138], interactive ad-hoc query [50,55,168,175,192] and machine learning jobs [40, 61, 111, 144, 157, 203]. They are running inside tens of millions short- or long-lived workers [86, 194, 196], which stay alive until applications complete. They could tolerate moderate performance fluctuations, and focus on throughputs and makespan instead of strict latency.

[7]Datawork is an integrated platform providing data lakes services like ETL, data pipelines and storage to support unified big data warehouse, machine learning and real-time computation engines of MaxCompute [5], PAI [6] and Blink [12]. It has the similar role to Google Cloud Dataflow [44] and Microsoft Naiad [146].

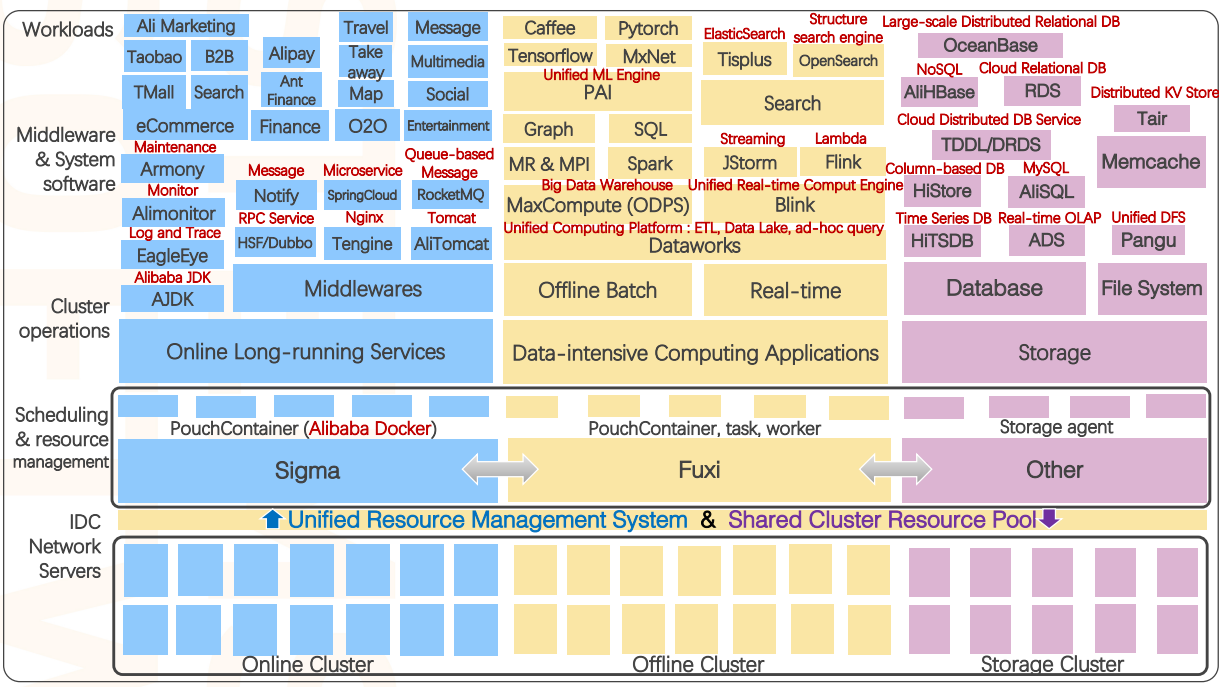


Figure 5.8: The global view of Alibaba technique stack and architecture.

Data-intensive real-time computing (DIRC) jobs provides services based on consuming data in real-time. It includes diverse types including end-to-end [21, 116, 176] or batch streaming processing [13, 204] and search [3, 7] atop of unified lambda [36] engine Blink [12]. They require both low latency and high throughput.

Storage services and system daemon process are agents running in the back-stage. They provide many distinct large-scale massive storage and database services like distributed file system [2], relational [1,22,25], key-value store [23], object storage [9,10] and time series [8] databases. They serve the whole categories of above workloads as seperate agents. The stabilities of these agents are critical to ensure SLAs of other applications.

Various categories of workloads from different systems have totally distinct execution and scheduling workflow, resource usage pattern and SLA objectives. For example, web (http) and communication (HSF) components of online LRSs need substantial stateless CPU computations and low network latency, cache components (Tair) rely on high-speed memory and network bandwidth with large capacities while database services (TDDL) require high

IOPS and low latency. The different combination manners of these components by LRSs as shown in Figure 5.7 aggravate the complexity of resource demands. Even same LRSs belong to different departments have fully unique traffics, loads and resource patterns. Most services (e.g. tp, buy..) from new retail offline stores have several magnitudes lower computation and concurrency requirements than ones of eCommerce.

Moreover, we need to process millions of applications across the entire group per day, 50% of which are non-recurring and random interactive queries. Accordingly, taking diverse workflows of a wide variety of distributed data-intensive computing systems and such large-scale workloads into account, it is impractical to perform QoS-aware scheduling by precise profilings or runtime predictions for arbitrary workload through historical telemetry that commonly adopted in recent works [67, 75, 112, 160]. Other application-oriented methodologies to mitigate interferences [74, 75] under co-locations are ineffective in this scene either.

It brings huge challenges to co-locations. How to select appropriate applications and efficiently co-locate massive scopes of workloads without prior knowledges of their characteristics, while ensuring SLAs and millsecond-level latency for interference-sensitive LRSs during long-chain and peak traffic bursts, is becoming a world-wide challenge. Additionally, it also becomes a barrier for accurate capacity planning and just right allocations due to the difficulty and inaccuracy of resource-to-performance modeling. It always leads to severe resource inefficiency and SLA violations.

In this unique scene, we need some runtime dynamic adjustment and control techniques from scheduler and operating system layer that introduced later. We rely on priority-based elastical management and multiplexing of resources to offset the passable quality of scheduling. We refine over- or under-provisioning by overcommitment, reclaimation and preemption based on actual usages to maximize efficiency. We continously ensure SLAs through dynamic auto-scaling and isolation approaches based on a feed-back control loop.

C5: Unified scheduling challenges due to conflicts of scheduling objectives and workflows for co-located LRSs and data-intensive jobs.

The dominate production services are recurring LRSs that execute for weeks or months with strict SLA requirements of latency. Since these long-lived containers need to be pre-deployed before task executions and seldom migrate once dispatched, LRSs have significant fewer opportunities to be scheduled. They require predictable reservations of specific number of resources (CPU cores, capacities of mem, disk and IO bandwidth..) that always remain unchanged during future long-term executions. They could tolerate expensive scheduling and planning overheads to achieve better placements and global optimization objectives, as well as satisfying complex multi-constraints of affinity and anti-affinity, fairness, fragmentations and load balance [86, 196].

On the contrary, there are tens of millions of daily data-intensive applications. More than 70% of them are short-running non-recurring interactive jobs that execute within 3 minutes. The millsecond-level task duration [57, 73, 113, 122, 153, 170] and high throuputs require schedulers to rapidly make millsecond-level decisions. They naturally have a vast number of allocation opportunities to compensate poor scheduling qualities through dynamic adjust and incrementally re-allocate resources to numerous short-running tasks. They need low scheduling latency and accommodate passable qualities, whose effects sustain only minutes.

The scheduling objectives of diverse workloads are conflictive. Even different data-intensive systems have distinct scheduling workflows (e.g. Tensorflow and MPI: gang-scheduling with asynchronizaed communications, Spark and Graph: DAG-based scheduling under BSP [178] barriers.). How to design a unified resource scheduler to co-locate various workloads, while satisfying conflictive objectives and workflows in a large-scale shared resource pool is extremely challenging.

C6: Inefficient colocations due to heterogeneous I/O and storages devices in two types of clusters. Additionally, data-intensive workloads [14, 15, 71, 203] and LC services were originally designed to run in seperate clusters with heterogeneous storage and network devices. Batch clusters use hard disks to satisfy massive storage and coarse-grained throughputs of I/O, whereas LC ones are leveraging SSD to achieve strict low latency. A

large number of offline workloads depend on network-intensive shuffling operations to perform data communications. Co-locate LC workloads on batch cluster with congested network and slow disk I/O would violate their SLAs. The limited storage capacities of SSD in online cluster could not accommodate data of batch workloads either. The massive data copy needs between two types of clusters during colocations are unaffordable.

Envolvement of Infrastructure in Alibaba

To maximize efficiency and sufficiently utilize spare resources (**C1**), hundreds of thousands of servers across 18 global datacenters of modern Alibaba infrastructure constitute a large-scale shared resource pool as a private cloud coordinated by an unified resource management system, instead of separately dedicated clusters. Online LRSs of all buisness are containerized and running inside Linux containers (LXC) [29, 52, 185] to achieve better isolations and utilization. We developed scheduling system for LRSs named Sigma [39] and container management tool named PouchContainer [134] 9 year ago to manage millions of containers uniformly. We also built Fuxi [210] to schedule diverse data-intensive workloads since 2009.

Solutions of C6. To enable colocations, we started to decouple computation and data storage three years ago. Most applications are re-written to remove dependency between execution logics and data path to support stateless services. All hard disks and SSDs constitute an unified distributed storage cluster [2] as software-defined storage (SDS) [33]. Co-located LC and data-intensive workloads read and write data to remote storage cluster through network I/O, eliminating barriers of heterogeneous devices and data movement. We also upgrade network bandwidth from 10Gbps to 25Gbps and 100Gbps to overcome increased I/O pressures, which is prove to be able to effectively resolve significant network congestion in disaggregate datacenter [85, 100, 121, 125, 171]. Other techniques and designs from hardwares, racks, physical datacenter and various systems also make huge contributions to support colocations.

Solutions of C1-C4. Five years ago, we started to co-locate diverse workloads and leverage elastic resource sharing to maximize efficiency, and take adequate advantages of large-scale resource pool satisfy demands of big events. We relied on priority-based elastic resource manager and isolation techniques from scheduler, architecture and operating system layer such as flexible cgroup management, reclaimation and preemption, LLC isolation, memory bandwidth control, customerized CFS scheduling [18, 154] to ensure stringent SLAs for interferences-sensitive and long-chain services without over-provisioning and getting rid of long-tail latency. We developed a customized and open source version of Linux kernel and OpenJDK named Alikernel [20] and AJDK to support above changes.

Unified Resource Scheduling at Large Scale

The number of data-intensive workloads overwhelmly surpassed LRSs while occupied at least half of resources and machines of global datacenters in recent years. Daily tens of millions of workloads with distinct workflows and objectives are managed by two fully different schedulers Sigma and Fuxi seperately over ten years. Scheduling for online LRSs depend on various constraints [86, 88] and elastic re-planning [57, 112, 170, 185], whereas dispatching for data-intensive workloads rely on heuristics algorithms involving in multi-resource fairness and bin-packing [87, 94–96, 196]. There is a bunch of particular designs and accumulated placement experiences for distinct schedulers in Alibaba like distinctive reservation and admission-control mechanisms, the same as most internet companies. It is impractical and unaffordable to replace them by re-designing an unified one, which needs to be immoderately complicated to satisfy various objectives. It also brings enormous risks of architecture re-design and stabilities degradation of production systems that were running over dozen of years.

Solutions of C4-C5. We designed a hybrid two-level architecture that combines Sigma and Fuxi in a shared-state [170] way. Sigma is designed to be compatible with APIs of kubernetes [51] and they share common design patterns and similar architectures, whereas

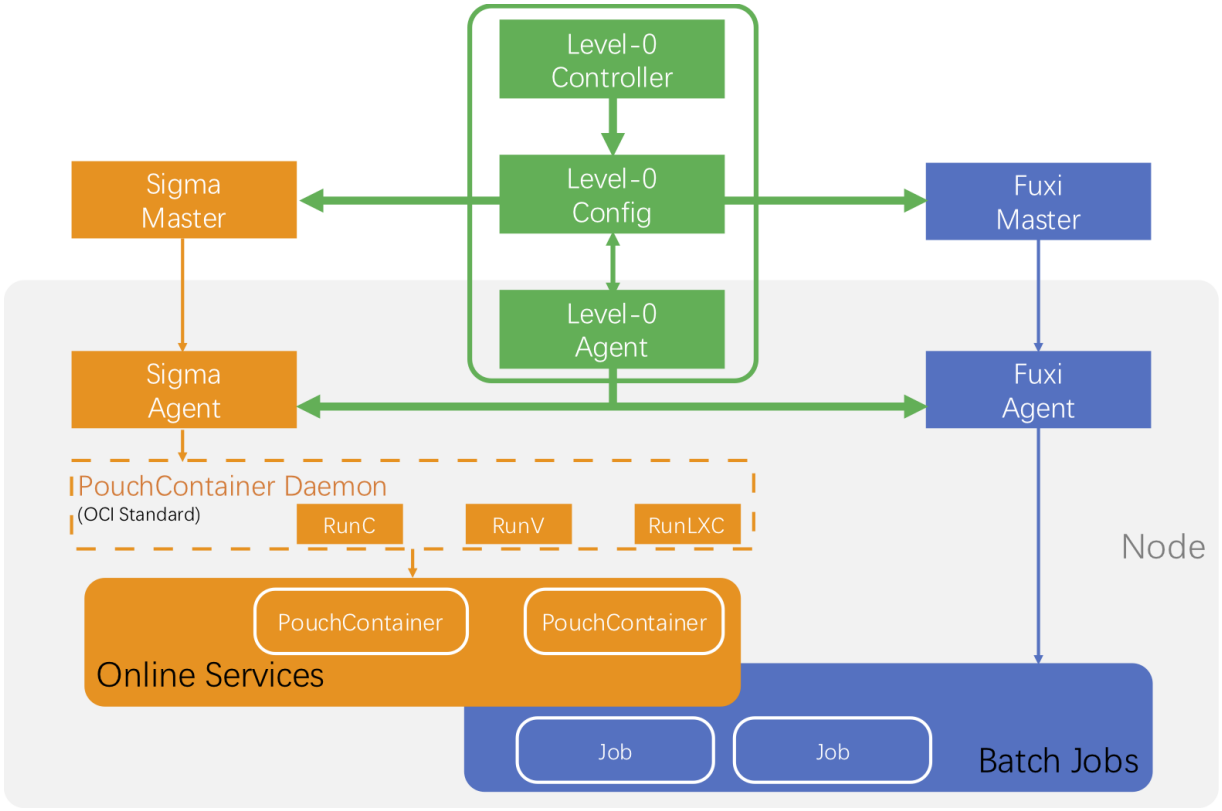


Figure 5.9: Alibaba unified resource management.

Fuxi [210] resembles to Yarn [181] as a two-level scheduler. LRSs and data-intensive workloads are dispatched by original respective algorithms constantly. However, resource allocations are uniformly controlled by a Level0-manager as shown in Figure 5.9. Level0-manager coordinates, synchronizes and notifies the state and usages updates for two schedulers. Besides, Level0-manager also provides abilities of elastic multiplexed, auto-scaling and isolation controls during runtime. This unified management has been adopted by each cluster for three years and effectively supported big events like "Double 11". Through efficient communication and coordination design, it has no obvious drawbacks compared to centralized schedulers like Borg [185].

Elastic Resource Sharing

In this section, we would discuss unified elastic resource management of colocations (Level0-manager). The capacity planning and scheduling, container orchestration, and ar-

chitecture of Sigma and Fuxi are beyond our contents, which could be discussed in the future. We rely on priority-based quota allocations, overcommitment, preemption and reclaimation to fulfill elastic multiplexing of resources and maximize utilization. We also have auto-scaling approach to rapidly make peak shifts during big events.

Priority-based Quota, Overcommitment, Preemption and Reclaimation

Essentially, there are two prominent elastic allocation approaches that have been widely used in large-scale production datacenters: priority-based quota [57,108,170,185] and hierarchical max-min fairness [30,87,94–96,114,181,202,210]. Since data-intensive workloads managed by Fuxi naturally have lower priority than production LRSs of Sigma, Level0-manager adopts the former to fast take back overcommited resources in time and guarantee strict SLAs of LRSs by restrict resource contentions even under pulse bursts of big events. Fuxi adopts the latter to sufficiently share resources between various organizations with identical priorities.

Quota Allocation

Virtually, quota is a medium of admission control to decide which types of workloads to admit for acquiring resources. The vector of actual dynamic available resource (CPU, RAM, disk, I/O..) per machine is divided as quotas by Sigma and Fuxi. Applications are admitted only if quotas of their groups (Sigma or Fuxi) are sufficient enough to fit in their reservation demands. Jobs under insufficient quota group are immediately rejected upon submission to avoid server overload. In the representative datacenter scheduler like Borg [185], priority-based quotas are simply divided as production and non-production ones that non-production quota is never guaranteed in the face of resource contention. However, there is a brunch of critical daily data-intensive analytics jobs that owning high priority managed by Fuxi. The quotas of Sigma and Fuxi are both production ones that need to be strictly guaranteed rather than overselling lower-priority ones of batch jobs [185].

Table 5.10: Priority and QoS Guarantee of Resources and Workloads

Resource Priority	Workload Type
Gold (S1)	sensitive LC LRSs
Silver (S2)	most LRSs, DIRCs, and some DIOCs (normal)
Copper (S3)	most DIOCs (overcommitment)
Resource Priority	**QoS Guarantee**
Gold (S1)	CPU: 100% reserved guarantee, exclusively occupied cores, no overlap with silver, overlap with copper through preemption by CPU share and HT (Hyper Thread) isolation; Memory: most page caches, reclaimation and OOM kill at the end.
Silver (S2)	CPU: specific share proportion guarantee, ms-level scheduling latency, not exclusively occupied, system daemon agents, no overlap with gold, overlap with copper through preemption by CPU share without HT isolation; Memory: medium page caches, reclaimation and OOM kill in the middle.
Copper (S3)	CPU: uncertain share proportion without guarantee, used for peak load shaving, overlap with gold and silver, preemptible anytime with least share proportion; Memory: least page caches, reclaimation and OOM kill at first.

Nevertheless, strict quota allocation incurs inefficient resource sharing since dynamic remaining quotas are calculated based on reserved resources instead of actual usages. The large proportion of reservations are always underutilized due to over-provisioning and capacity mis-estimation [74, 75, 86, 112, 126, 160, 185, 194, 196]. We mainly rely on runtime overcommitment based on usages as a critical supplementary mechanism to sufficiently utilize resources.

Priority-based Overcommitment and Preemption

Persistently preempt resources of data-intensive jobs to make spaces for LRSs would prevent high-priority batch jobs being normally executed, which violating their fairness constraints. To enable effective overcommitment and ensure minimal resource availability for critical workloads, we set fine-grained priority and QoS guarantee for runtime resources as shown in Table 5.10. Applications request types of resources based on their SLA requirements.

We divide every hardware resource by three bands: gold, silver and copper. Different priorities indicate distinct runtime QoS guarantee of allocated resources managed by operating system and cluster manager. It affects hardware management like CPU (e.g. quota and

priority of CFS [18] scheduling and preemption), memory (reclaimation and OOM kill), I/O (IOPS rate of blkio and network bandwidth control) and other controls (hyper thread, LLC cache..). The aggregate gold and silver resources per server are less than its capacity to avoid over-allocation [94–96, 196]. Kernel and schedulers always rapidly reclaim low-priority resources to make space for high-priority jobs during contentions by task preemption, scheduling blacklist and migration. Workloads that employ these resources are named as S1, S2 and S3 respectively.

Only a small portion of critical LRSs request gold resources, which are always exclusively occupied by latency-sensitive services and never shared with other S1 or S2 workloads to guarantee strict SLAs. Most LRSs like buy or cart, DIRCs and portions of important DIOCs belong to silver types. These silver resources are always multiplexed and shared by workloads with same priorities. To prevent preemption cascades [8] and service cascade failures (**C3**), gold and silver resources are never preemptible. Both of them are scarce and be applied within the respective quota limit of Sigma and Fuxi. The quota proportions of online and offline groups are dynamic planned ahead based on historical usages and controlled by level0-manager. Most of the time, available gold and silver resources providing for Sigma online group are sufficient enough.

There are two types of offline resources: normal (silver) and overcommitment (copper). Normal ones are applied within the quota limit of Fuxi offline group, whereas overcommitted ones depend on actual usages that are not related to quota reservation. Most latency tolerant batch jobs are using best-effort copper resources to execute, which are underutilized ones of gold and silver quota reservations. They tolerate lower-quality resources, and could be preempted and reclaimed anytime during contentions when S1 or S2 jobs need. They resume later by efficient rescheduling and recomputations.

Since offline data-intensive workloads always make use of 5× to 8× more resources than online LRSs as shown in Section , the average 60% utilization of Fuxi quota group is

[8] A high-priority task bumped out a slightly lower-priority one, which bumped out another slightly-lower priority task.

quite higher than 5% of Sigma. The prominent overcommited resources come from Sigma group. To avoid quite frequent and wasteful evictions, Level0-manager continuously profiles and estimates how many resources LRSs will use in a near-future time window. We set the rest of Sigma quota as available copper resources and dispatch proper offline workloads whose profiled usages just fit in these free resources to minimize fragmentations and avoid server overloading [196]. We also add a safety margin to usage predictions of LRSs to decide the limit of total overcommited resources, so as to accommodate unexpected load spikes and mis-estimations. The overcommited silver resources of offline quota group are similar and controlled by Fuxi [210].

Overcommitment and preemption are key to improve efficiency. There are more application-oriented fine-grained priorities within each category for delicacy management (system, monitoring, production, batch...). By priority-based rules, we are capable of co-locating diverse workloads under two different schedulers to achieve both high efficiency and stringent SLA guarantee for LC LRSs during long chains (**C1-C3**).

CPU

We introduce detail management of CPU resources in this section. Generally. there are two modes of CPU usages:

CPU set: Workloads are binding to some specific logical cores of CPU. The contentions are easy to locate and control at the expenses of inefficient sharing and load imbalances.

CPU share: Workloads could utilize any spare core within the shared group based on time-slice. It provides flexible and sufficient sharing of CPU, but needing complicated management and isolation techniques to elimanite uncertain contentions or monopolism by inappropriate allocations.

The limited CPU cores per machine are not sufficient enough to simultaneously satisfy reservation demands of S1, S2 and S3 workloads under pure CPU set mode. Additionally, a portion of data-intensive workloads require high-priority guarantees of resources with strong

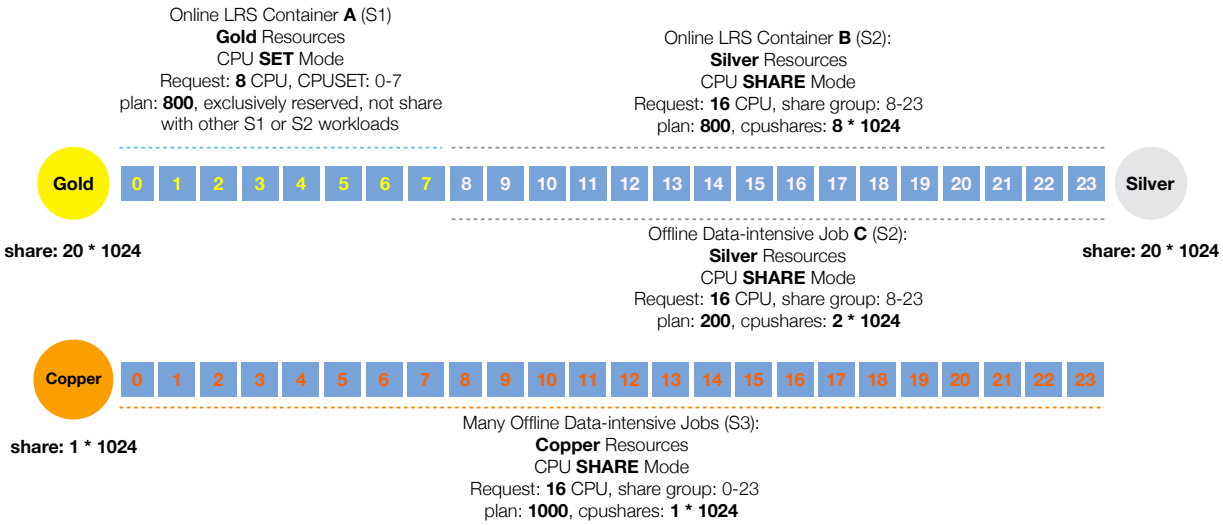


Figure 5.10: CPU elastic management.

isolations. Best-effort S3 jobs are also friendly to CPU share with short-term contentions. Therefore, we adopt a priority-based hybrid mode of both set and share to fully utilize CPU while strictly control contentions.

Discussion and Future

The scheduling and resource management of Sigma are not discussed in this paper. We focus on colocation of LRSs and offline data-intensive workloads in this paper and leave LRSs scheduling introduction to future work. We also characterize dependencies and diverse chains of different LRSs and data-intensive workloads in other papers.

We already publish two public traces [38] to quatitively demonstrate the scanerios and status of Alibaba datacenter. They include resource usages and execution time for both online services and offline jobs, as well as detail system metrics involving interferences such as CPI, cache miss per thousands of instructions, memory access frequency. We would continuesly update traces to reflect the status of Alibaba datacenter.

CHAPTER 6 Imbalance in the Cloud: an Analysis on Alibaba Cluster Trace

Introduction

To improve resource efficiency and design intelligent scheduler for clouds, it is necessary to understand the workload characteristics and machine utilization in large-scale cloud data centers. In this paper, we perform a deep analysis on a newly released trace dataset by Alibaba Group in September 2017, consists of detail statistics of 11089 online service jobs and 12951 batch jobs co-locating on 1300 machines over 12 hours. To the best of our knowledge, this is one of the first work to analyze the Alibaba public trace. Our analysis reveals several important insights about different types of *imbalance* and *resource inefficiency* in the Alibaba cloud. Such imbalances exacerbate the complexity and challenge of cloud resource management, which might incur severe wastes of resources and low cluster utilization. 1) Spatial Imbalance: heterogeneous resource utilizations across machines and workloads. 2) Temporal Imbalance: greatly time-varying resource usages per workload and machine. 3) Imbalanced proportion of multi-dimensional resources (CPU and memory) utilization per workload. 4) Imbalanced multi-resource demands between online service and offline batch jobs. Additionally, the trace demonstrated that Alibaba cluster is operating at extremely low utilizations for online services (less than 10% CPU and 45% memory average utilizations). We believe accomodating such imbalances during resource allocation is critical to improve cluster efficiency, and will motivate the emergence of new resource managers and schedulers.

Cloud datacenters usually comprise thousands of machines, providing highly reliable, efficient and scalable services. Examples of typical cloud services including web search, e-commerce systems, and social networks. With the increasing popularity of cloud and data center computation, users tend to share large hardware platforms. However, effective resource management is very important to guarantee both quality of service and high resource utilization [54] [186] [169] [104] [196] [181] [57] [210].

Recent studies [75] [112] [160] [167] [195] revealed that most cloud facilities and commercial clusters are operating at low utilization. According to the data of Geithner and McKinsey several years ago, the global server utilization seems to be very low, which is only 6% to 12%. Even leveraging virtualization technology, the utilization is still below 17%. It probably incurs low cost-efficiency, energy-proportional and scalability challenges of clouds.

Co-locating online service and offline batch jobs on the same cluster is shown to be an efficient approach to improve cluster utilization in modern cloud data centers [198] [62] [186]. However, the trace demonstrated that Alibaba cluster reserved fix amounts of resources for online services rather than elastical allocations. Under such reservation mechanism, traditional co-locating strategy is ineffective because batch jobs could not leverage reserved idle resources of service jobs. Additionally, contention and interference on shared resources can cause latency spikes that violate the service-level objectives of service jobs. Ensuring quality of service (QoS) for latency-sensitive job is non-trival in such environment.

By understanding the workload characteristics and machine utilization in large-scale cloud data centers, we could provide predictable knowledges to cluster manager. Through planning ahead and performing intelligent scheduling, we could improve resource efficiency and avoid such interferences.

In this paper, we perform a deep analysis on a newly released trace dataset by Alibaba Group in September 2017, covering 1300 servers over 12 hours [38]. Alibaba Cloud is one of the largest public cloud platforms in the world, on which processing millions of tasks acrossing hundreds of data centers everyday. This trace includes runtime statistics of a hybrid cluster, on which online service and offline batch jobs are co-locating. As we know, it is the unique one having hybrid runtime information among all public traces.

To the best of our knowledge, this is one of the first work to analyze the public Alibaba trace. We explored runtime status of the hybrid cluster, and showed several important insights about **imbalanced utilization** and **resource inefficiency** in the cloud.

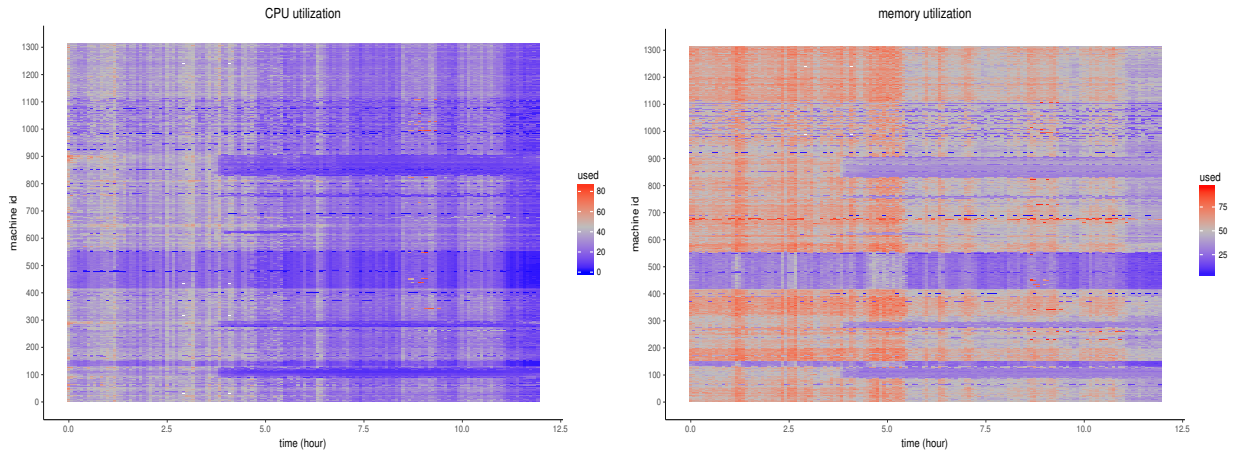


Figure 6.1: The heat maps of CPU and memory utilization of machines in the cluster. The white portion indicates the lack of data in the trace. Red color indicates high utilization while blue color indicates low utilization.

Such imbalances exacerbate the complexity and challenge of cloud resource management. It includes:

- Spatial Imbalance: heterogeneous resource utilization across machines and workloads.
- Temporal Imbalance: greatly time-varying resource usages per workload and machine.
- Imbalanced proportion of multi-dimensional resources (CPU and memory) utilization per workload.
- Imbalanced resource demands and runtime statistics (duration and task number) between online service and offline batch jobs.

Many modern resource managers are designed under the assumption of ideal cluster environment. The commonly occurred imbalance phenomenons in Alibaba trace would lead to significant resource inefficiency and wastes. We believe it is critical to accomodate such imbalances during resource allocation to improve cluster efficiency. They will also motivate the emergences of new resource managers and schedulers.

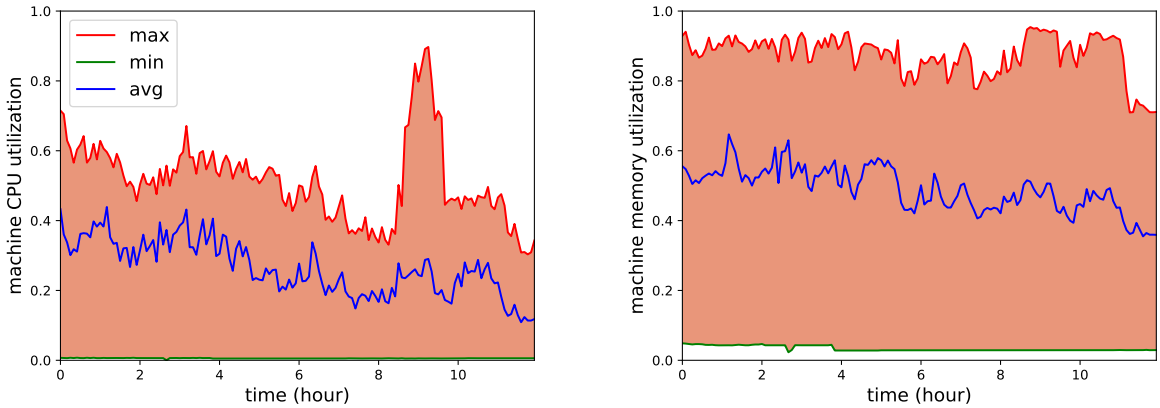


Figure 6.2: The CPU and memory utilization of machines during execution. The red line indicates the maximum utilization of all machines in the cluster, the blue one indicates the average utilization and green one means minimum utilization of all machines.

The Dataset

Alibaba released a new dataset ClusterData201708 in September 2017, which contains a production cluster runtime information during 12 hours period, and includes 1.3k machines that run both *online service* and *offline batch jobs* [38]. The data is motivated to address the low utilization and resource inefficiency challenges of Alibaba cluster when co-locating online services and batch jobs.

There are three types of data in the trace: machine utilization and runtime information of both batch and online service workloads. For confidentiality reasons, portion information in the trace is obfuscated.

Machine utilization is described as two tables: the "machine events" table and the "machine resource utilization" table. Capacities reflect the normalized multi-dimension physical capacity per machine. Each dimension (CPU cores, RAM size) is normalized independently.

Batch workloads are described as two tables: "instance" table and "task" table. The user submits a batch workload in the form of Job (which is not included in the trace). Each job cocnsists of multiple tasks, each forming a DAG according to the data dependency. They

are consisting of multiple instances and executing different computing logics. Instance is the smallest scheduling unit of batch workload. All instances within a task execute exactly the same binary code with identical multi-resource demands, but processing different portions of data.

Online service jobs are described by two tables: "service instance event" and "service instance usage". The trace includes only two types of instance events. One event for creation, and another for finish. Event of creation records the startime of a service instance, and event of remove indicates the finish of an service instance. Each instance is the smallest scheduling unit and running in a lightweight virtual machine of Linux container (LUX). It could also be regarded as a complete service job.

Either intances of batch or service workloads express their resource demands in the form of reservation, which is commonly used in modern resource managers [186] [181] [104] [169] [57] [210]. And their cluster manager of Fuxi [210] leverages admission-control strategy for resource allocation. The combination of above two mechanisms is regarded to be the essential cause of low cluster utilization and resource inefficiency in recent studies [196] [195] [76]. In the following sections, we introduced several imbalanced phenomenons in Alibaba cloud.

Imbalances of Machines

Figure 6.1 plots the resource utilization per machine in the cluster during 12 hours. The trace provided normalized CPU and memory usages infomation per sampling time for each machine. All the data are retrieved from "machine events" and "machine resource utilization" table.

We had an interesting observation that CPU utilizations of portion of machines (id from 400 to 600 and 900 to 1100) are always higher than others while their memory utilizations are relatively lower. And CPU utilizations of most machines are gradually increasing during cluster running while memory utilizations are decreasing. Thus we could always ob-

serve the highest CPU utilization and lowest memory utilization of machines at the end of trace period (from 11 to 12 hour). In contrast, CPU is always idle at the begining (from 0 to 3.5 hour) while memory keeps high load.

It demonstrated that there exists significant **spatial imbalance** (heterogeneous resource utilization across machines) and **temporal imbalance** (time-varying resource usages per machine) of utilization for machines in cloud data center.

From Figure 6.2, we saw more fine-grained information of resource usages per machine. We summaried average, minimum and maximum utilization among 1300 machines at each sampling time. Both the CPU and memory usages are normolized.

The average CPU utilization per machine is within 40% and maximum maintains about 60% along the sampling period. Average memory utilization per machine is within 60% and maximum about 90%. The green line plots utilizations of the machine whose utilization is the minimum among all machines per sampling time. Both CPU and memory utilization of such minimum usages are nearing zero. From hour 8 to 10, the maximum CPU utilization rapidly spikes, reaching over 90%, while the average CPU usages maintain stable. By comparing these huge gaps between minimum, average and maximum usages of machines, we observed tremendous spatial imbalance of utilization in cluster. It demonstrated that cloud data centers need new schedulers to balance the load and avoid hot spot of machine utilization, so as to improve cluster efficiency. Differ from CPU usages, memory usages maintain steady during that period. It also indicates **the proportion of multi-dimensional resources utilization (CPU and memory) of workloads is imbalanced**.

Additionally, we observed severe wastes and **resource inefficiency** of CPU and memory resources in cluster. However, due to relatively low maximum usages of machines, CPU utilization has the opportunity to be greatly improved through comprehensively understanding workloads' resource demands and making proper reservations. Nevertheless, improving memory utilization is challenging since job performance is sentitive to the relatively high maximum usages of machines. Simply decreasing the reservations to improve

cluster memory efficiency might lead to serious performance degradation due to thrashing. Recent study [195] proposed one solution by making better demands estimations. In conclusion, the cloud data center needs new resource managers and schedulers to improve cluster resource efficiecny by avoiding above imbalanced and low utilization.

Imbalances of Workloads

In the trace, workloads are classified into two categories. One is long-term service job, another is short-term batch job. Each service instance belongs to one job, and is running within a Linux container for 12 hours. While each instance of batch jobs belongs to one task, and is running for seconds or minutes. Multiple tasks compose of one batch job. Detail runtime statistics of batch workloads are shown in Table 6.11.

Each job commonly has several tasks, but the maximum one has 156. There are three types of status for batch tasks, including normally terminated, failed and waiting due to preemption. Most tasks are normally terminated, while over 2000 are waiting. The majority of tasks own hundreds of instances, while some has an extremely large number of 64486. The corresponding average durations of instances and tasks are 129 and 192 seconds respectively. The maximum durations are 29558 and 29585 seconds, while both of the minimum durations are less than 1 second. By diving into the task execution infomation, we found the longest task that ran over 8 hours was consisting of several longest instances that were executing at the same time. Thus their maximum durations are similar.

In constrast, each service job consists of only one long-term instance. There are totally 11089 service instances (jobs) running for the whole 12 hours (43200 seconds). It illustrates the **imbalanced numbers and durations of runtime instances for service and batch jobs.** By leveraging such imbalanced knowledges, one could schedule and co-locate batch and service instances in a more efficient way.

Table 6.11: Statistics of Batch Jobs

Status	Number
Failed tasks	1126
Terminated tasks	67013
Waiting tasks	8847
Average instance number per task	152
Maximum instance number per task	64486
Minimum instance number per task	1
Average task number per job	6
Maximum task number per job	156
Minimum task number per job	1
Total instances	15186017
Total tasks	76986
Total jobs	12951
Average instance duration	129 (seconds)
Maximum instance duration	29558 (seconds)
Mimimum instance duration	$\leq$ 1 (seconds)
Average task duration	192 (seconds)
Maximum task duration	29585 (seconds)
Mimimum task duration	$\leq$ 1 (seconds)

To make efficient resource reservations, it is necessary to understand the workload characteristics and demands. We studied the distribution of resource requests, actual usages and corresponding utilization in the following subsections.

Imbalances of Resource Demands

Batch Workloads

For each job, we summaried its average requested and used CPU numbers per task in Figure 6.3. We accumulated the CPU and normalized memory requests of all instances within a task. And accumulated requested resources of all tasks within the same job, then divided by corresponding task numbers to get the average values. However, the scale of the normalized memory size per task would not be from 0 to 1, which we ignored.

We could see most of the batch jobs requested 1 to 100 cores of CPU for each task, while the maximum requested number is more than 1000. In contrast, we observed most jobs used 0.01 to 1 core CPU per task while very few used more than 100 cores. Additionaly, we

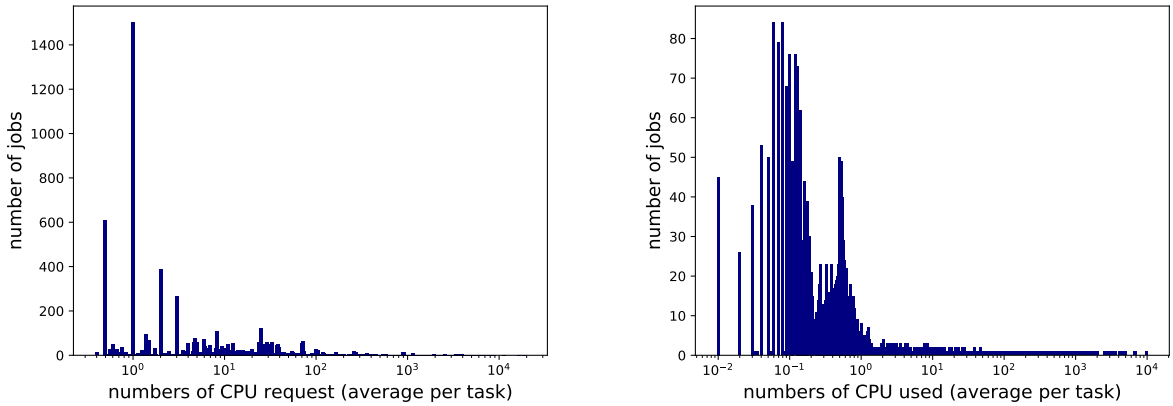


Figure 6.3: Job counts by average CPU request numbers (left) and average CPU used numbers (right) per task. Note the log-scale on the plot's x-axis.

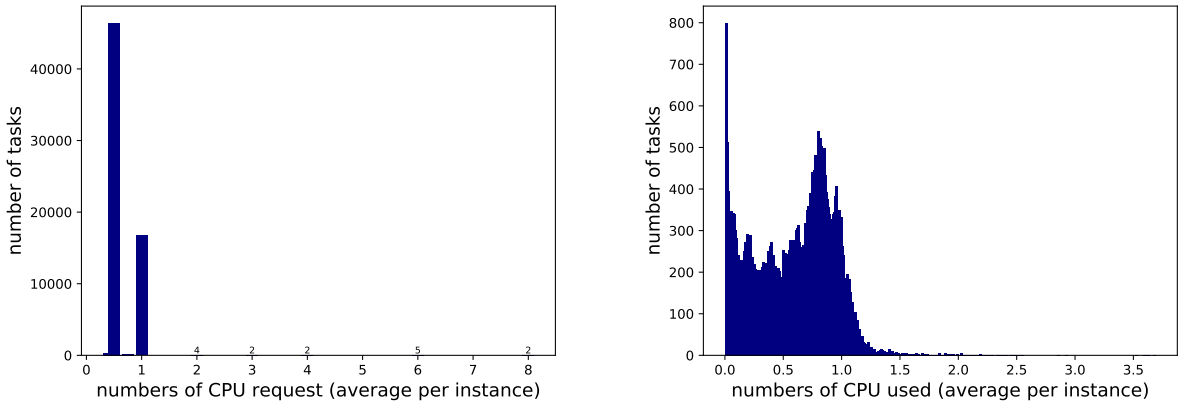


Figure 6.4: Task counts by average CPU request numbers (left) and used numbers per instance (right). Note the log-scale on the plot's x-axis.

observed many jobs are waiting for resources while few jobs occupied overwhelming cluster CPU resources (more than 100 cores per job). Such **spatical imbalance** of CPU usages across workloads probably leads to the bottleneck of cluster throughput, while exacerbating ineffieicny of resources. New scheduling algorithms are essential to accomodate imbalanced loads and demands of workloads.

In Figure 6.4 and Figure 6.5, we summaried the average requested and used CPU numbers per instance for each task, as well as normalized memory sizes. We accumulated

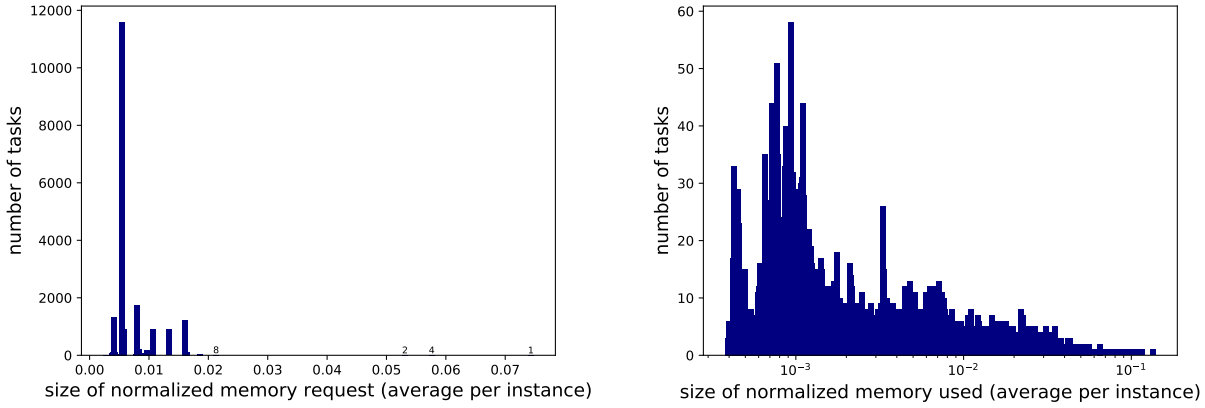


Figure 6.5: Task counts by normalized average memory request sizes (left) and used sizes (right) per instance. Note the log-scale on the plot's x-axis.

all the CPU numbers and normalized memory sizes of instances within a task. And retrieve the average value through dividing the sum by corresponding task's instance numbers.

Most tasks request either 0.5 or 1 core CPU per instance, while few requests 6 or 8 cores (hard to distinguish in figure). The used CPU numbers are mainly between 0.1 and 0.7. Small portion of tasks' average used CPU numbers per intance are between 0.7 and 1.2. As we can see, most tasks' instances are operating at half of the CPU utilization (used to requests). Due to the mechanism of resource reservation, Alibaba cluster are suffering severe **inefficiency** and wastes of resources.

From Figure 6.5, the majority of tasks requested normalized memory sizes between 0.05 and 0.15 per instance. While they commonly used 0.001 to 0.05 sizes. However, since the request and used memory sizes per instance are normalized independently in two seperate tables, it is not accurate to observe memory utilizations by comparing them directly. It's shown that most tasks are consuming only small portions of memory, while few occupied the majorities. It confirms the existences of **spatial imbalance** across workloads, and highlights the motivation to design new allocation mechanisms to handle complexity of scheduling.

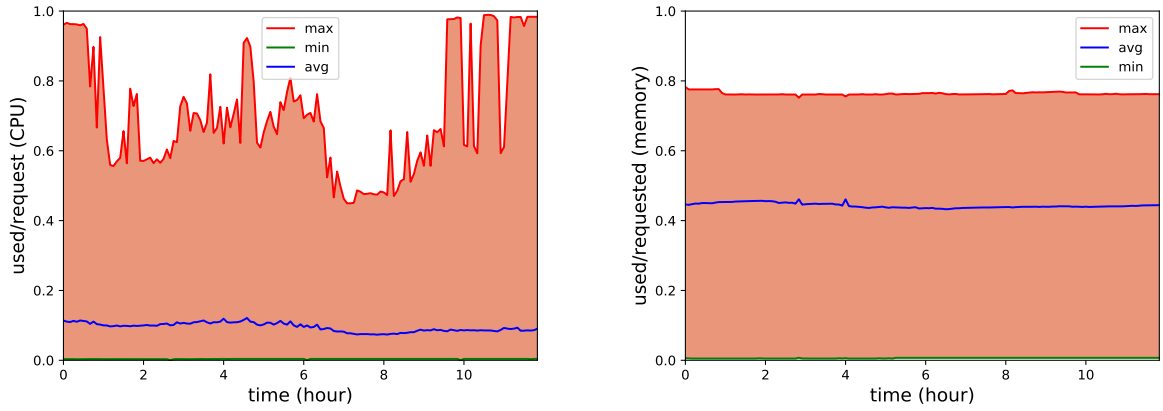


Figure 6.6: The ratio of used CPU and memory to requested per sampling time. The red, blue, green lines indicate the maximum, average and minimum ratio of all service instances respectively.

Service Workloads

Each service instance is running within one Linux container for 12 hours. Figure 6.6 shows the average, maximum and minimum ratio of resource used to requests of all instances at each sampling time. It indicates the average time-varying utilization of CPU and memory per service instance.

Most service instances stably used less than 10% CPU resources they requested during executions. However, there were always some portions of instances consuming 60% to 90% resources (red maximum line), while some used near-to-zero cores (green minimum line). Such **spatial** and **temporal imbalances** across service instances make it knotty to make proper reservations. Balance the trade-offs between performance and resource efficiency would be the principal challenge for cluster managers. The normalized average memory utilization is stably 45%, while maximum keeps 79% and minimum maintains 1%. Unlike resources of CPU, it is shown that there are opportunities to make better reservations to improve memory utilization [195].

Differ from the time-varying average utilizations of all instances in Figure 6.6 (spatial average), Figure 6.7 plots the CDF of instances' average CPU and memory utilizations of 12 hours (temporal average). The traces provides average CPU and memory utilization

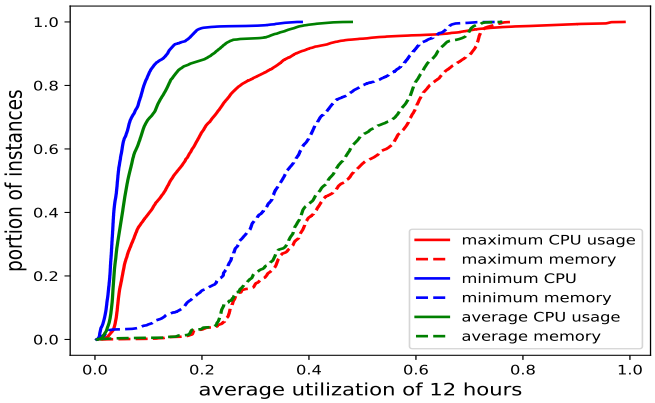


Figure 6.7: CDF of instances' average resource utilizations of 12 hours.

every 5 mintue per instance. We list the maximum and minimum values of 5-minute average utilization during 12 hours per instance, and plot the CDF. Similarly, we sumed all 5-minute utilizations of 12 hours, and dividing it by intervals ($12 * 60 \div 5 = 144$) to achieve the average CDF curve.

There are 50% of instances whose average CPU utilization of 12 hours reached up to 0.05, maximum 0.2 and minimum 0.02. There are even 90% of instances whose maximum CPU utilization of 12 hours only reached up to 0.4, which illustrates extremely huge wastes of reserved CPU numbers. Unlike the idle of most CPU cores, memory utilization is a little bit higher. There were 50% of instances reaching about 0.45, 0.5 and 0.35 respectively. Comparing with Figure 6.6, it identifies **imbalanced proportion between CPU and memory utilization** for service instances. Resource allocation strategy should take such imbalance into account, and design better fair share algorithm of multi-dimensional resources.

Users always tend to over-provision resources to guarantee SLA for latency-sensitive production services. However, such extremely low utilizations would lead to incredible high costs for large-scale cloud data center. Meanwhile, online service jobs reserved and hold resources forever, which might cause imbalanced cluster loads (hot spots) or job starvation due to insufficient resources on constrainted hosts. By considering the results of Section and Table 6.11, batch and online service jobs are shown to have serious **imbalanced instance**

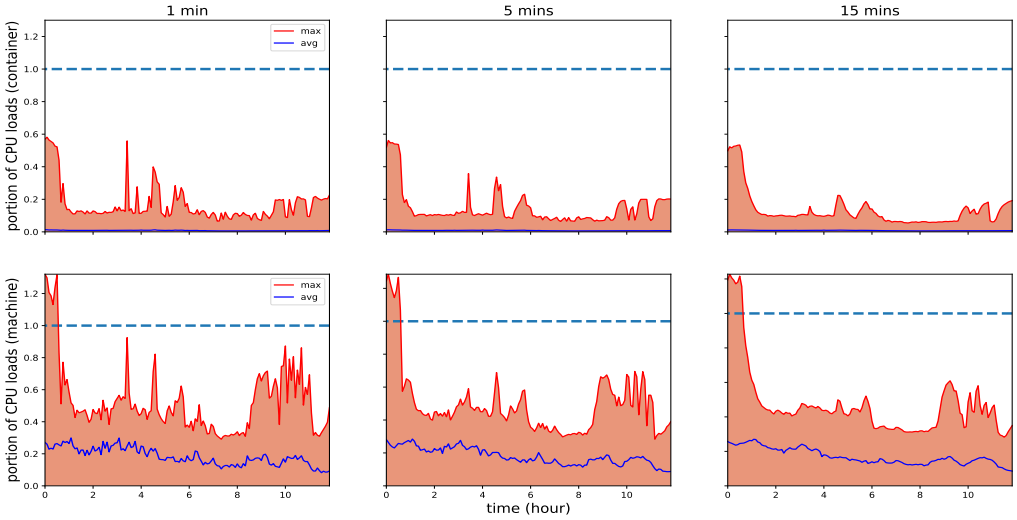


Figure 6.8: Hourly average and maximum CPU loads of all service instances (top) and machines (bottom). The sampling interval is one minute, five minutes, and fifteen minutes from left to right. The dashed line represents the total capacity of each machine.

numbers, resource utilization and duration. Modern schedulers could take above runtime phenomenons of hybrid cluter into account, and adopts sphotistical co-locating strategy to avoid imbalance and maximize resource efficiency.

The top of Figure 6.8 shows the hourly average and maximum CPU loads of all service instances. We observed the maximum CPU loads are below 60% while the average are below 10%. Additionally, CPU loads are about 60% at beginning and drastically drop to 20% one hour later. Afterwards, the maximum loads fluctuate over time while average ones keep stably few. Most instances are idle in cluster. We could see obvious **spatial and temporal imbalances** across service workloads, which increase the complexity of scheduling.

The bottom of Figure 6.8 displays the hourly average and maximum CPU loads of all machines. The fluctuation trends of both maximum and average loads are similar to containers'. However, the gaps between highest and lowest usages are even bigger. At beginning, the machine's maximum CPU loads are even over 1. While the minimum ones are still close to 0. The average CPU loads are about 20%. It confirmed the existenses of **spatial and temporal imbalances** across machines.

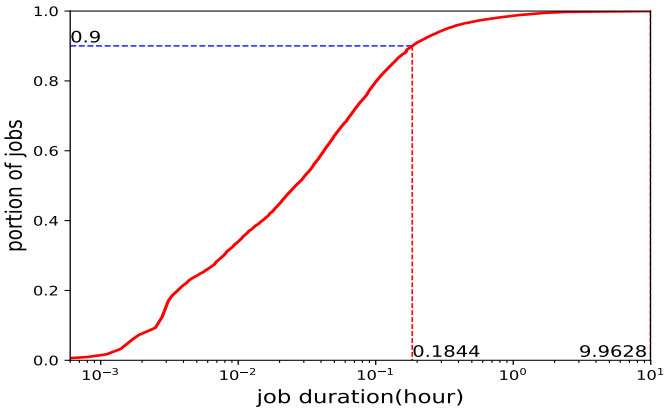


Figure 6.9: CDF of job durations. We only count the terminated jobs.

Imbalances of Batch Job Durations

Figure 6.9 plots the duration distributions of batch jobs. We exploited the difference between durations of earliest created task and latest finished task within the same job, to indicate job running time. 90% of jobs run less than 0.19 hours, while the longest one is running up to 10 hours. In detail, over 12481 jobs run less than half of an hour and over 12705 jobs run less than 1 hour. Short jobs overwhelmingly occupied the cluster. It also identifies the **imbalances of job durations.** One could take these phenomenons into account, and leverage proper scheduling algorithm such as SJF (Shortest Job First) to speedup executions of short jobs, while maximizing cluster makespan.

In addition, the large proportion of short jobs give us opportunities to improve quality of co-locating choices in hybrid cluster. We have more opportunities to select other proper jobs to avoid interferences and contentions between batch and service jobs. A scheduler that adequately exploiting such imbalances could greatly improve cluster efficiency and guarantee SLA for service jobs.

Discussion

Due to the reservation machanism and imbalanced phenomenons in Alibaba cloud data center, co-locating service and batch jobs is ineffective to improve cluster efficiency. In

the future, one could leverage elastical allocations of containers and knowledges of imbalances to greatly improve resource efficiency in hybrid cluster.

In addition, by considering data locality, the imbalance phenomenons would be aggravated during scheduling. How to make proper resource allocation and scheduling decisions to balance the trade-offs between imbalance relief, data locality and SLA (performance) is challenging. It also becomes our future research direction.

Summary

Understand machine characteristics and workload behaviors in large-scale cloud data centers is critical to maximize cluster resource efficiency. In this paper, we performed a deep analysis on a newly released trace dataset by Alibaba Group in September 2017, covering 1300 servers over 12 hours. To the best of our knowledge, this is one of the first work to analyze the Alibaba public trace.

We explored detail runtime characteristics of a hybrid cluster that co-locates both online service and offline batch jobs. And discovered several interesting insights about *imbalance* in the cloud. Such imbalances exacerbate the complexity and challenges of cloud cluster management, incurring severe resource inefficiency. We believe accomodating imbalances of both machines and workloads is critical to cluster efficiency, and will motivate the design and emergences of new resource managers and schedulers.

CHAPTER 7 CONCLUSION AND FUTURE WORK

Conclusions

We concluded the evolution of resource scheduling and datacenter computation engines over decades. We demonstrated the motivation and trend to force us towards intelligent scheduling. We showed the inherent NP-hard online scheduling problems could be transferred to resolvable offline scheduling issue, which has global optimal solution rather than a heuristic algorithm. By obtaining three critical abilities of intelligent scheduling, we are capable of efficiently manage datacenter in modern complicated architecture.

We also illustrated the long-running workloads overwhelminngly occupy most modern commercial datacenters. The characteristics of LRAs or LRSs bring new challenges, and we need rely on contention (QoS)-aware predictive scheduling (Prophet) and optimal reservation estimation (MEER/Prometheus) to perform efficient scheduling, so as to achieve both best datacenter efficiency and optimal application performance for both users and cluster operators.

We summaried the evolutions and development of resource scheduling filed from different perspectives and observations. We compare them with our motivation of intelligent scheduling, to learn the unique observations from them and identify their drawbacks. By abstract and conclude from the whole filed, we think out the trends and directions of future next-generation predictive datacenter resource management and scheduling.

We also introduced a complex enterprise datacenter co-location techniques at large scale of Alibaba Group. It is far more complicated than scheduling, but also involved in evolutions of IDC, physical datacenters, racks and cluster topology, network and storage, server hardwares and local operating system. Modern infrastructure motivates us to manage global datacenters from top to bottom.

We evaluated the efficiency and effectiveness of modern infrastructure of Alibaba equipped with co-location techniques through data analysis of newest public Alibaba datacenter trace. We gained several interesting insights and observations of enterprise global

datacenters at large scale, and motivate us toward next-generation of architectures by considering solutions of these issues.

Improve multi-resource efficiency in modern complicated global datacenter infrastructure is non-trivial, especially when considering emerging long running workloads. It proposed higher level requirements of techniques. It is necessary and general trend to leverage artificial intelligence to explore next-generation intelligent scheduling methodology.

Future Directions

We are developing the new layer of managing local OS efficiently, to act in concert with cluster schedulers to better achieve their objectives. It would also provide abundant knowledges of both profilings and predictive ones, so as to assist schedulers towards intelligent scheduling. It enables the end-to-end guarantee that schedulers could have the ability to fully control workloads since they are submitted by users till the completion, even during runtime to prevent unexpected contentions or interferences.

We are also refining the profilings, predictions and other techniques to achieve better and more accurate results of three necessary and critical abilities (a)(b)(c). We are straightly on the way to intelligent scheduling.

REFERENCES

[1] Alibaba apsaradb for rds. https://www.alibabacloud.com/product/apsaradb-for-rds.

[2] Alibaba cloud: alibaba pangu. https://www.alibabacloud.com/blog/pangu-%E2%80%93-the-high-performance-distributed-file-system-by-alibaba-cloud_594059.

[3] Alibaba cloud elasticsearch. https://www.alibabacloud.com/product/elasticsearch.

[4] Alibaba cloud: Hsf development. https://www.alibabacloud.com/help/doc-detail/63867.html.

[5] Alibaba cloud large-scale data warehousing: Maxcompute. https://www.alibabacloud.com/product/maxcompute.

[6] Alibaba cloud machine learning platform: Pai. https://www.alibabacloud.com/product/machine-learning?spm=a3c0i.11852017.1180869.1.4c2e7ac9cbQ2PJ.

[7] Alibaba cloud opensearch. https://www.alibabacloud.com/help/doc-detail/26108.htm.

[8] Alibaba cloud: Time series database (hitsdb). https://www.alibabacloud.com/product/hitsdb.

[9] Alibaba hbase. https://www.alibabacloud.com/blog/blogbloghow-to-use-hbase-client-to-access-alibaba-cloud-nosql-table-store_351732.

[10] Alibaba object storage service (oss). https://www.alibabacloud.com/product/oss?spm=a3c0i.11736792.1148529.1.35e65ce0V0Ix4T.

[11] Alibaba public trace. https://github.com/alibaba/clusterdata.

[12] Alibaba real-time engine: Blink (alibaba flink and storm). https://flink.apache.org/news/2017/04/04/dynamic-tables.html.

[13] Apache flink. http://flink.apache.org.

[14] Apache hadoop. http://hadoop.apache.org.

[15] Apache spark. http://spark.apache.org.

[16] Apache spark : Monitoring and instrumentation. https://spark.apache.org/docs/2.2.1/monitoring.html.

[17] Apache spark configuration. http://spark.apache.org/docs/2.2.1/configuration.html.

[18] Completely fair scheduler (cfs) in linux. https://en.wikipedia.org/wiki/Completely_Fair_Scheduler.

[19] Gartner says efficient data center design can lead to 300 percent capacity growth in 60 percent less space. http://www.gartner.com/newsroom/id/1472714.

[20] Github: alibaba alikernel. https://github.com/alibaba/alikernel.

[21] Github: alibaba jstorm. https://github.com/alibaba/jstorm.

[22] Github: alibaba oceanbase. https://github.com/alibaba/oceanbase.

[23] Github: Alibaba tair. https://github.com/alibaba/tair.

[24] Github: Alibaba tddl. https://github.com/alibaba/tb_tddl.

[25] Github: Alisql (alibaba mysql). https://github.com/alibaba/AliSQL.

[26] Google public cluster trace. https://github.com/google/cluster-data.

[27] Hadoop mapreduce - capacity scheduler. http://bit.ly/1tGpbDN.

[28] Hadoop mapreduce - fair scheduler. http://bit.ly/1p7sJ1I.

[29] Linux containers (lxc). https://linuxcontainers.org/.

[30] Peloton: UberâĂŹs unified resource scheduler for diverse cluster workloads. https://eng.uber.com/peloton/.

[31] Prime down: Amazon sale day turns into fail day. https://techcrunch.com/2018/07/16/prime-down-amazons-sale-day-turns-into-fail-day/.

[32] Slarm github: spark-memory-postprocess. https://github.com/woggle/spark-memory-postprocess.

[33] Software-defined storage. https://en.wikipedia.org/wiki/Software-defined_storage.

[34] Spark-bench benchmark suite. https://github.com/SparkTC/spark-bench.

[35] Tpc-ds benchmark suit. http://www.tpc.org/tpcds/default.asp.

[36] Wikipedia: Lambda architecture. https://en.wikipedia.org/wiki/Lambda_architecture.

[37] Google trace. *https://github.com/google/cluster-data*, 2011.

[38] Alibaba trace. *https://github.com/alibaba/clusterdata*, 2017.

[39] Github: alibaba open clusterdata. https://github.com/alibaba/clusterdata, 2017.

[40] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, volume 16 of *OSDI '16*, pages 265–283, 2016.

[41] O. A. Abdul-Rahman and K. Aida. Towards understanding the usage behavior of google cloud users: the mice and elephants phenomenon. In *Cloud Computing Technology and Science (CloudCom), 2014 IEEE 6th International Conference on*, pages 272–277. IEEE, 2014.

[42] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou. Re-optimizing data-parallel computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI '12, pages 21–21. USENIX Association, 2012.

[43] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. Vijaykumar. Shufflewatcher: Shuffle-aware scheduling in multi-tenant mapreduce clusters. In *2014 USENIX Annual Technical Conference*, USENIX ATC '14, pages 1–12, 2014.

[44] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernandez-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. volume 8 of *VLDB '15*, pages 1792–1803. VLDB Endowment, 2015.

[45] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '17, pages 469–482, 2017.

[46] Y. Amannejad, D. Krishnamurthy, and B. Far. Detecting performance interference in cloud-based web services. In *Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on*, pages 423–431. IEEE, 2015.

[47] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris. Scarlett: coping with skewed content popularity in mapreduce clusters.

In *Proceedings of the 6th ACM European conference on Computer Systems*, EuroSys '11, pages 287–300. ACM, 2011.

[48] G. Ananthanarayanan, M. C.-C. Hung, X. Ren, I. Stoica, A. Wierman, and M. Yu. Grass: trimming stragglers in approximation analytics. In *11th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '14, 2014.

[49] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI '10, page 24, 2010.

[50] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1383–1394. ACM, 2015.

[51] K. authors. Kubernetes: Production-grade container orchestration. `https://github.com/kubernetes/kubernetes`, 2017. Access data: 2018-08-19.

[52] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of 1999 USENIX Conference on Operating Systems Design and Implementation*, OSDI '99, pages 45–58, 1999.

[53] L. A. Barroso, J. Clidaras, and U. Holzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 8(3):1–154, 2013.

[54] L. A. Barroso, J. Clidaras, and U. Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 8(3):1–154, 2013.

[55] M. Bittorf, T. Bobrovytsky, C. Erickson, M. G. D. Hecht, M. J. I. J. L. Kuff, D. K. A. Leblang, N. L. I. P. H. Robinson, D. R. S. Rus, J. R. D. T. S. Wanderman, and M. M. Yoder. Impala: A modern, open-source sql engine for hadoop. In *Proceedings of the 7th Biennial Conference on Innovative Data Systems Research*, CIDR '15, 2015.

[56] B. Bonlender. State of the data center industry. In *Department of Commerce, State of Washington*, 2018.

[57] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI '14, pages 285–300. USENIX Association, 2014.

[58] X. Bu. Autonomic management and performance optimization for cloud computing services. 2013.

[59] X. Bu, J. Rao, and C.-z. Xu. Interference and locality-aware task scheduling for mapreduce applications in virtual clusters. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, pages 227–238. ACM, 2013.

[60] M. Carvalho, W. Cirne, F. Brasileiro, and J. Wilkes. Long-term slos for reclaimed cloud computing resources. In *5th ACM Symposium on Cloud Computing*, SoCC '14, pages 1–13, 2014.

[61] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. 2015.

[62] W. Chen, J. Rao, and X. Zhou. Preemptive, low latency datacenter scheduling via lightweight virtualization. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 251–263. USENIX Association, 2017.

[63] Y. Chen, A. S. Ganapathi, R. Griffith, and R. H. Katz. Analysis and lessons from a publicly available google cluster trace. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-95*, 94, 2010.

[64] Y. Cheng, Z. Chai, and A. Anwar. Characterizing co-located datacenter workloads: An alibaba case study. In *Proceedings of the 9th Asia-Pacific Workshop on Systems*, APSys '18, page 12. ACM, 2018.

[65] R. C. Chiang and H. H. Huang. Tracon: interference-aware scheduling for data-intensive applications in virtualized environments. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 47. ACM, 2011.

[66] M. . Company. Revolutionizing data center efficiency. In *Uptime Institute Symposium*, 2008.

[67] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 27th Symposium on Operating Systems Principles*, SOSP '17, 2017.

[68] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica. Clipper: A low-latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '17, pages 613–627, 2017.

[69] C. Curino, D. E. Difallah, C. Douglas, S. Krishnan, R. Ramakrishnan, and S. Rao. Reservation-based scheduling: If you're late don't blame us! In *Proceedings of 5th the ACM Symposium on Cloud Computing*, SoCC '14, pages 1–14. ACM, 2014.

[70] J. Dean and L. A. Barroso. The tail at scale. In *Communications of the ACM*, pages 74–80, 2013.

[71] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[72] P. Delgado, D. Didona, F. Dinu, and W. Zwaenepoel. Job-aware scheduling in eagle: Divide and stick to your probes. In *Proceedings of the 7th ACM Symposium on Cloud Computing*, SoCC '16. ACM, 2016.

[73] P. Delgado, F. Dinu, A.-M. Kermarrec, and W. Zwaenepoel. Hawk: Hybrid datacenter scheduling. In *2015 USENIX Annual Technical Conference*, USENIX ATC '15, pages 499–510, 2015.

[74] C. Delimitrou and C. Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 77–88. ACM, 2013.

[75] C. Delimitrou and C. Kozyrakis. Quasar: resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14. ACM, 2014.

[76] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. *ACM SIGPLAN Notices*, 49(4):127–144, 2014.

[77] C. Delimitrou and C. Kozyrakis. Hcloud: Resource-efficient provisioning in shared cloud systems. In *Proceedings of the 21th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 473–488. ACM, 2016.

[78] C. Delimitrou, D. Sanchez, and C. Kozyrakis. Tarcil: Reconciling scheduling speed and quality in large shared clusters. In *Proceedings of the 6th ACM Symposium on Cloud Computing*, SoCC '15, pages 97–110. ACM, 2015.

[79] S. Di, D. Kondo, and F. Cappello. Characterizing cloud applications on a google data center. In *Parallel Processing (ICPP), 2013 42nd International Conference on*, pages 468–473. IEEE, 2013.

[80] S. Di, D. Kondo, and W. Cirne. Characterization and comparison of cloud versus grid workloads. In *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*, pages 230–238. IEEE, 2012.

[81] K. Elmeleegy, C. Olston, and B. Reed. Spongefiles: Mitigating data skew in mapreduce using distributed memory. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, SIGMOD '14, pages 551–562. ACM, 2014.

[82] L. Fang, K. Nguyen, G. Xu, B. Demsky, and S. Lu. Interruptible tasks: Treating memory pressure as interrupts for highly scalable data-parallel programs. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 394–409. ACM, 2015.

[83] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM European conference on Computer Systems*, EuroSys '12, pages 99–112. ACM, 2012.

[84] R. Gandhi, D. Xie, and Y. C. Hu. Pikachu: How to rebalance load in optimizing mapreduce on heterogeneous clusters. In *USENIX Annual Technical Conference*, USENIX '13, pages 61–66, 2013.

[85] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker. Network requirements for resource disaggregation. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '16, pages 249–264, 2016.

[86] P. Garefalakis, K. Karanasos, P. Pietzuch, A. Suresh, and S. Rao. Medea: Scheduling of long running applications in shared production clusters. In *Proceedings of the 13th European Conference on Computer Systems*, Eurosys '18, 2018.

[87] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI*, volume 11, pages 24–24, 2011.

[88] A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Choosy: Max-min fair sharing for datacenter jobs with constraints. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 365–378. ACM, 2013.

[89] L. Gidra, G. Thomas, J. Sopena, M. Shapiro, and N. Nguyen. Numagic: a garbage collector for big data on big numa machines. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, 2015.

[90] A. Goder, A. Spiridonov, and Y. Wang. Bistro: Scheduling data-parallel jobs against live production systems. In *USENIX Annual Technical Conference*, USENIX ATC '15, pages 459–471, 2015.

[91] I. Gog, J. Giceva, M. Schwarzkopf, K. Vaswani, D. Vytiniotis, G. Ramalingan, M. Costa, D. Murray, S. Hand, and M. Isard. Broom: Sweeping out garbage collection from big data systems. In *HotOS*, HotOS '15, 2015.

[92] I. Gog, M. Schwarzkopf, A. Gleave, R. N. Watson, and S. Hand. Firmament: Fast, centralized cluster scheduling at scale. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '16, 2016.

[93] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *11th USENIX*

Symposium on Operating Systems Design and Implementation, volume 14 of *OSDI '14*, pages 599–613, 2014.

[94] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. In *Proceedings of the 2014 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '14, pages 455–466. ACM, 2014.

[95] R. Grandl, M. Chowdhury, A. Akella, and G. Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *Proceedings of 12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '16, page 65. USENIX Association, 2016.

[96] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni. Graphene: Packing and dependency-aware scheduling for data-parallel clusters. In *Proceedings of 12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '16, pages 81–97. USENIX Association, 2016.

[97] W. D. Gropp, W. Gropp, E. Lusk, A. Skjellum, and A. D. F. E. E. Lusk. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999.

[98] Y. Guo, J. Rao, D. Cheng, and X. Zhou. ishuffle: Improving hadoop performance with shuffle-on-write. volume 28, pages 1649–1662. IEEE, 2017.

[99] J. Han, S. Jeon, Y.-r. Choi, and J. Huh. Interference management for distributed parallel applications in consolidated clusters. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 443–456. ACM, 2016.

[100] S. Han, N. Egi, A. Panda, S. Ratnasamy, G. Shi, and S. Shenker. Network support for resource disaggregation in next-generation datacenters. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, HotNet '13. ACM, 2013.

[101] H. Herodotou, F. Dong, and S. Babu. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SoCC '11, page 18. ACM, 2011.

[102] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A self-tuning system for big data analytics. In *Proceedings of the 5th biennial Conference on Innovative Data Systems Research*, volume 11 of *CIDR '11*, pages 261–272, 2011.

[103] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI '11, pages 295–308. USENIX Association, 2011.

[104] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

[105] B. Huang, M. Boehm, Y. Tian, B. Reinwald, S. Tatikonda, and F. R. Reiss. Resource elasticity for large-scale machine learning. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 137–152. ACM, 2015.

[106] C. Iorgulescu, F. Dinu, A. Raza, W. U. Hassan, and W. Zwaenepoel. Don't cry over spilled records: Memory elasticity of data-parallel applications and its application to cluster scheduling. In *2017 USENIX Annual Technical Conference*, USENIX ATC '17, 2017.

[107] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM*

SIGOPS/EuroSys European Conference on Computer Systems, EuroSys '07, pages 59–72, 2007.

[108] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP'09, pages 261–276. ACM, 2009.

[109] V. Jalaparti, H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Bridging the tenant-provider gap in cloud services. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, page 10. ACM, 2012.

[110] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar. Network-aware scheduling for data-parallel jobs: Plan when you can. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 407–420. ACM, 2015.

[111] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

[112] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, Í. Goiri, S. Krishnan, J. Kulkarni, and S. Rao. Morpheus: Towards automated slos for enterprise clusters. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '16, page 117. USENIX Association, 2016.

[113] K. Karanasos, S. Rao, C. Curino, C. Douglas, K. Chaliparambil, G. M. Fumarola, S. Heddaya, R. Ramakrishnan, and S. Sakalanaga. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *2015 USENIX Annual Technical Conference*, USENIX ATC '15, pages 485–497, 2015.

[114] I. A. Kash, G. OâĂŹShea, and S. Volos. Dc-drf: Adaptive multi-resource sharing at public cloud scale. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, pages 374–385, 2018.

[115] Y. Koh, R. Knauerhase, P. Brett, M. Bowman, Z. Wen, and C. Pu. An analysis of performance interference effects in virtual environments. In *Performance Analysis of Systems & Software, 2007. ISPASS 2007. IEEE International Symposium on*, pages 200–209. IEEE, 2007.

[116] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 239–250. ACM, 2015.

[117] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skew-stant parallel processing of feature-extracting scientific user-defined functions. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 75–86. ACM, 2010.

[118] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skewtune: mitigating skew in mapreduce applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 25–36. ACM, 2012.

[119] A. Kyrola, G. E. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In *10th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '12, 2012.

[120] P. Lama and X. Zhou. Aroma: Automated resource allocation and configuration of mapreduce environment in the cloud. In *Proceedings of the 9th international conference on Autonomic computing*, ICAC '12, pages 63–72. ACM, 2012.

[121] G. Lauterbach and A. R. Rao. Dis-aggregated and distributed data-center architecture using a direct interconnect fabric, 2012. US Patent 8,140,719.

[122] J. Leverich and C. Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems*, Eurosys '14. ACM, 2014.

[123] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura. Sparkbench: A comprehensive benchmarking suite for in memory data analytic platform spark. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, CF '15, pages 53:1–53:8. ACM, 2015.

[124] M. Li, J. Tan, Y. Wang, L. Zhang, and V. Salapura. Sparkbench: a spark benchmarking suite characterizing large-scale in-memory data analytics. pages 2575–2589, 2017.

[125] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *Proceedings of the 36th annual international symposium on Computer architecture*. ACM, 2009.

[126] L. Liu and H. Xu. Elasecutor: Elastic executor scheduling in data analytics systems. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, pages 107–120, 2018.

[127] Q. Liu and Z. Yu. The elasticity and plasticity in semi-containerized co-locating cloud workload: a view from alibaba trace. In *Proceedings of the 9th ACM Symposium on Cloud Computing*, SoCC '18. ACM, 2018.

[128] Q. Liu and Z. Yu. The elasticity and plasticity in semi-containerized co-locating cloud workload: a view from alibaba trace. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, pages 347–360. ACM, 2018.

[129] Z. Liu and S. Cho. Characterizing machines and workloads on a google cluster. In *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, pages 397–403. IEEE, 2012.

[130] D. Lo, L. Cheng, R. Govindaraju, and C. Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. In *Proceeding of the International Symposium on Computer Architecuture*, volume 42 of *ISCA '14*, pages 301–312, 2014.

[131] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 450–462, 2015.

[132] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 450–462. ACM, 2015.

[133] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, pages 716–727, 2012.

[134] A. G. H. Ltd. Pouchcontainer: Alibaba's open-source container runtime. `https://github.com/alibaba/pouch`, 2018. Access data: 2018-08-19.

[135] C. Lu, K. Ye, G. Xu, C.-Z. Xu, and T. Bai. Imbalance in the cloud: an analysis on alibaba cluster trace. In *Proceedings of the IEEE International Conference on Big Data*, pages 2884–2892, 2017.

[136] L. Lu, X. Shi, Y. Zhou, X. Zhang, H. Jin, C. Pei, L. He, and Y. Geng. Lifetime-based memory management for distributed data processing systems. In *Proceedings of the 42nd International Conference on Very Large Data Bases*, VLDB '16, pages 936–947. VLDB Endowment, 2016.

[137] M. Maas, K. Asanovic, T. Harris, and J. Kubiatowicz. Taurus: A holistic language runtime system for coordinating distributed managed-language applications. In *Proceedings of the 21th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, 2016.

[138] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 135–146. ACM, 2010.

[139] H. Mao, M. Alizadeh, I. Menache, and S. Kandula. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pages 50–56. ACM, 2016.

[140] 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 *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '11, pages 248–259. ACM, 2011.

[141] J. Mars, L. Tang, K. Skadron, M. L. Soffa, and R. Hundt. Increasing utilization in modern warehouse-scale computers using bubble-up. In *Proceedings of the 45th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '12, pages 88–99, 2012.

[142] J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. Contention aware execution: online contention detection and response. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 257–265. ACM, 2010.

[143] C. C. C. C. K. S. Y. X. P. D. J.-P. S. J. L. L. S. L. Maurice Gagnaire, Felipe Diaz. Downtime statistics of current cloud solutions. pages 716–727, 2012.

[144] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, et al. Mllib: Machine learning in apache spark. volume 17, pages 1235–1241. JMLR. org, 2016.

[145] A. K. Mishra, J. L. Hellerstein, W. Cirne, and C. R. Das. Towards characterizing cloud backend workloads: insights from google compute clusters. *ACM SIGMETRICS Performance Evaluation Review*, 37(4):34–41, 2010.

[146] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455. ACM, 2013.

[147] R. Nathuji, A. Kansal, and A. Ghaffarkhah. Q-clouds: managing performance interference effects for qos-aware clouds. In *Proceedings of the 5th European conference on Computer systems*, pages 237–250. ACM, 2010.

[148] R. Nathuji, A. Kansal, and A. Ghaffarkhah. Q-clouds: managing performance interference effects for qos-aware clouds. In *Proceedings of the 5th European conference on Computer systems*, Eurosys '10, pages 237–250. ACM, 2010.

[149] S. Newman. *Building microservices: designing fine-grained systems*. " O'Reilly Media, Inc.", 2015.

[150] K. Nguyen, L. Fang, G. H. Xu, B. Demsky, S. Lu, S. Alamian, and O. Mutlu. Yak: A high-performance big-data-friendly garbage collector. In *12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '16, pages 349–365, 2016.

[151] K. Nguyen, K. Wang, Y. Bu, L. Fang, J. Hu, and G. Xu. Facade: A compiler and runtime for (almost) object-bounded big data applications. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 675–690. ACM, 2015.

[152] D. Novakovic, N. Vasic, S. Novakovic, D. Kostic, and R. Bianchini. Deepdive: Transparently identifying and managing performance interference in virtualized environments. In *Proceedings of the USENIX Annual Technical Conference*, pages 219–230. USENIX Association, 2013.

[153] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 69–84. ACM, 2013.

[154] C. S. Pabla. Completely fair scheduler. *Linux Journal*, 2009.

[155] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant. Automated control of multiple virtualized resources. In *Proceedings of the 4th ACM European conference on Computer systems*, Eurosys'09.

[156] P. Padala, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, and K. Salem. Adaptive control of virtualized resources in utility computing environments. In *Proceedings of the 2nd ACM European conference on Computer systems*, Eurosys '07, pages 289–302. ACM, 2007.

[157] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. 2017.

[158] D. A. Patterson et al. A simple way to estimate the cost of downtime. In *USENIX LISA*, volume 2 of *LISA '02*, pages 185–188, 2002.

[159] M. Poess, T. Rabl, H. A. Jacobsen, M. Poess, T. Rabl, H. A. Jacobsen, M. Poess, T. Rabl, and H. A. Jacobsen. Analysis of tpc-ds: the first standard benchmark for sql-based big data systems. In *Proceedings of the 8th ACM Symposium on Cloud Computing*, SoCC '17, pages 573–585, 2017.

[160] K. Rajan, D. Kakadia, C. Curino, and S. Krishnan. Perforator: eloquent performance models for resource optimization. In *Proceedings of the 7th ACM Symposium on Cloud Computing*, SoCC '16, pages 415–427. ACM, 2016.

[161] S. R. Ramakrishnan, G. Swart, and A. Urmanov. Balancing reducer skew in mapreduce workloads using progressive sampling. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, page 16. ACM, 2012.

[162] J. Rao. Autonomic management of virtualized resources in cloud computing. 2011.

[163] J. Rasley, K. Karanasos, S. Kandula, R. Fonseca, M. Vojnovic, and S. Rao. Efficient queue management for cluster scheduling. In *Proceedings of the Eleventh European Conference on Computer Systems*, Eurosys '16, page 36. ACM, 2016.

[164] C. Reiss. Understanding memory configurations for in-memory analytics. Ph.D Dissertation at University of California, Berkeley (2016), 2016.

[165] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, page 7. ACM, 2012.

[166] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 7. ACM, 2012.

[167] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Towards understanding heterogeneous clouds at scale: Google trace analysis. *Intel Science and Technology Center for Cloud Computing, Tech. Rep*, page 84, 2012.

[168] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino. Apache tez: A unifying framework for modeling and building data processing applications. In *Proceedings of the 2015 ACM SIGMOD international conference on Management of Data*, SIGMOD '15, pages 1357–1369. ACM, 2015.

[169] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 351–364. ACM, 2013.

[170] M. Schwarzkopf, A. Konwinski, M. AbdElMalek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM european conference on Computer Systems*, Eurosys '13, pages 351–364, 2013.

[171] Y. Shan, Y. Chen, Y. Huang, S. Hallymysore, and Y. Zhang. Lego: A distributed, decomposed os for resource disaggregation. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI '18, 2018.

[172] B. Sharma, V. Chudnovsky, J. L. Hellerstein, R. Rifaat, and C. R. Das. Modeling and synthesizing task placement constraints in google compute clusters. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 3. ACM, 2011.

[173] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SoCC '11, page 5. ACM, 2011.

[174] J. Tan, A. Chin, Z. Z. Hu, Y. Hu, S. Meng, X. Meng, and L. Zhang. Dynmr: Dynamic mapreduce with reducetask interleaving and maptask backfilling. In *Proceedings of the Ninth European Conference on Computer Systems*, Eurosys '14, page 2. ACM, 2014.

[175] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proceedings of the VLDB Endowment*, 2(2):1626–1629, 2009.

[176] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. Storm@twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 147–156. ACM, 2014.

[177] A. Tumanov, T. Zhu, J. W. Park, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger. Tetrisched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the Eleventh European Conference on Computer Systems*, Eurosys '16, page 35. ACM, 2016.

[178] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[179] A. Vasan, A. Sivasubramaniam, V. Shimpi, T. Sivabalan, and R. Subbiah. Worth their watts? an empirical study of datacenter servers. In *Proceedings of the 16th IEEE International Symposium on High Performance Computer Architecture*, HPCA '10, pages 1–10, 2010.

[180] N. Vasic, D. Novakovic, S. Miucin, D. Kostic, and R. Bianchini. Dejavu: accelerating resource allocation in virtualized environments. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, pages 423–436, 2012.

[181] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SoCC '13, pages 5:1–5:16. ACM, 2013.

[182] S. Venkataraman, A. Panda, G. Ananthanarayanan, M. J. Franklin, and I. Stoica. The power of choice in data-aware cluster scheduling. In *11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '14, pages 301–316, 2014.

[183] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *Proceedings of the 13th*

USENIX Symposium on Networked Systems Design and Implementation, NSDI '16, pages 363–378. USENIX Association, 2016.

[184] A. Verma, L. Cherkasova, and R. H. Campbell. Aria: automatic resource inference and allocation for mapreduce environments. In *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ICAC '11, pages 235–244. ACM, 2011.

[185] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, Eurosys '15, page 18. ACM, 2015.

[186] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, page 18. ACM, 2015.

[187] J. Wang and M. Balazinska. Elastic memory management for cloud data analytics. In *2017 USENIX Annual Technical Conference*, USENIX ATC '17, pages 745–758. USENIX Association, 2017.

[188] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, et al. Bigdatabench: A big data benchmark suite from internet services. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture*, HPCA '2014, pages 488–499. IEEE, 2014.

[189] Y. Wang, J. Tan, W. Yu, L. Zhang, X. Meng, and X. Li. Preemptive reducetask scheduling for fair and fast job completion. In *Proceedings of the 10th IEEE International Conference on Autonomic Computing*, ICAC '13, pages 279–289, 2013.

[190] A. Wieder, P. Bhatotia, A. Post, and R. Rodrigues. Orchestrating the deployment of computations in the cloud with conductor. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '12, 2012.

[191] T. Wood, P. J. Shenoy, A. Venkataramani, and M. S. Yousif. Black-box and gray-box strategies for virtual machine migration. In *NSDI*, NSDI'07.

[192] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: Sql and rich analytics at scale. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of data*, SIGMOD '13, pages 13–24. ACM, 2013.

[193] F. Xu, F. Liu, and H. Jin. Heterogeneity and interference-aware virtual machine provisioning for predictable performance in the cloud. *Computers, IEEE Transactions on*, 2015.

[194] G. Xu and C.-Z. Xu. Prometheus: online estimation of optimal memory demands for workers in in-memory distributed computation. In *Proceedings of the 8th ACM Symposium on Cloud Computing*, SoCC '17. ACM, 2017.

[195] G. Xu and C.-Z. Xu. Prometheus: Online estimation of optimal memory demands for workers in in-memory distributed computation. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 655. ACM, 2017.

[196] G. Xu, C.-Z. Xu, and S. Jiang. Prophet: Scheduling executors with time-varying resource demands on data-parallel computation frameworks. In *Proceedings of the 13th IEEE International Conference on Autonomic Computing*, ICAC '16, pages 45–54. IEEE, 2016.

[197] N. J. Yadwadkar, G. Ananthanarayanan, and R. Katz. Wrangler: Predictable and faster jobs using fewer resources. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '14, pages 1–14. ACM, 2014.

[198] Y. Yan, Y. Gao, Y. Chen, Z. Guo, B. Chen, and T. Moscibroda. Tr-spark: Transient computing for big data analytics. In *Proceedings of the 7th ACM Symposium on Cloud Computing*, SoCC '16, pages 484–496, 2016.

[199] Y. Yan, Y. Gao, Y. Chen, Z. Guo, B. Chen, and T. Moscibroda. Tr-spark: Transient computing for data analytics. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 484–496. ACM, 2016.

[200] H. Yang, A. Breslow, J. Mars, and L. Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 607–618. ACM, 2013.

[201] T. Ye and S. Kalyanaraman. A recursive random search algorithm for large-scale network parameter configuration. volume 31, pages 196–205. ACM, 2003.

[202] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*, Eurosys'10, pages 265–278. ACM, 2010.

[203] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. 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*, NSDI '12. USENIX Association, 2012.

[204] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438. ACM, 2013.

[205] J. Zhang, H. Zhou, R. Chen, X. Fan, Z. Guo, H. Lin, J. Y. Li, W. Lin, J. Zhou, and L. Zhou. Optimizing data shuffling in data-parallel computation by understanding user-defined functions. In *9th USENIX Symposium on Networked Systems Design and Implementation*, volume 12 of *NSDI '12*, pages 22–22, 2012.

[206] Q. Zhang, J. L. Hellerstein, and R. Boutaba. Characterizing task usage shapes in googleï£¡ï£¡s compute clusters. In *Large Scale Distributed Systems and Middleware Workshop (LADIS'11)*, 2011.

[207] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. Cpi 2: Cpu performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, Eurosys '13, pages 379–391. ACM, 2013.

[208] Y. Zhang, M. A. Laurenzano, J. Mars, and L. Tang. Smite: Precise qos prediction on real-system smt processors to improve utilization in warehouse scale computers. In *Proceedings of the 47th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '14, pages 406–418, 2014.

[209] Z. Zhang, L. Cherkasova, A. Verma, and B. T. Loo. Automated profiling and resource management of pig programs for meeting service level objectives. In *Proceedings of the 9th international conference on Autonomic computing*, ICAC '12, pages 53–62. ACM, 2012.

[210] Z. Zhang, C. Li, Y. Tao, R. Yang, H. Tang, and J. Xu. Fuxi: a fault-tolerant resource management and job scheduling system at internet scale. In *Proceedings of the VLDB Endowment*, VLDB '14, pages 1393–1404. VLDB Endowment, 2014.

[211] Y. Zhu, J. Liu, M. Guo, Y. Bao, W. Ma, Z. Liu, K. Song, and Y. Yang. Bestconfig: Tapping the performance potential of systems via automatic configuration tuning. In *Proceedings of the 8th ACM Symposium on Cloud Computing*, SoCC '17, pages 338–350, 2017.

ABSTRACT

DATA-DRIVEN INTELLIGENT SCHEDULING FOR LONG RUNNING WORKLOADS IN LARGE-SCALE DATACENTERS

by

GUOYAO XU

May 2019

Advisor: Dr. Cheng-Zhong Xu

Major: Computer Engineering

Degree: Doctor of Philosophy

Cloud computing is becoming a fundamental facility of society today. Large-scale public or private cloud datacenters spreading millions of servers, as a warehouse-scale computer, are supporting most business of Fortune-500 companies and serving billions of users around the world. Unfortunately, modern industry-wide average datacenter utilization is as low as 6% to 12%. Low utilization not only negatively impacts operational and capital components of cost efficiency, but also becomes the scaling bottleneck due to the limits of electricity delivered by nearby utility. It is critical and challenge to improve multi-resource efficiency for global datacenters.

Additionally, with the great commercial success of diverse big data analytics services, enterprise datacenters are evolving to host heterogeneous computation workloads including online web services, batch processing, machine learning, streaming computing, interactive query and graph computation on shared clusters. Most of them are long-running workloads that leverage long-lived containers to execute tasks.

We surveyed datacenter resource scheduling works over last 15 years. Most previous works are designed to maximize the cluster efficiency for short-lived tasks in batch processing system like Hadoop. They are not suitable for modern long-running workloads of Microservices, Spark, Flink, Pregel, Storm or Tensorflow like systems. It is urgent to develop new

effective scheduling and resource allocation approaches to improve efficiency in large-scale enterprise datacenters.

In the dissertation, we are the first of works to define, specify and identify the problems, challenges and scenarios of scheduling and resource management for diverse long-running workloads in modern datacenter. They rely on predictive scheduling techniques to perform reservation, auto-scaling, migration or rescheduling. It forces us to pursue and explore more intelligent scheduling techniques by adequate predictive knowledges. We innovatively identify what is intelligent scheduling, what abilities are necessary towards intelligent scheduling, how to leverage intelligent scheduling to transfer NP-hard online scheduling problems to resolvable offline scheduling issues.

We designed and implemented an intelligent cloud datacenter scheduler, which automatically performs resource-to-performance modeling, predictive optimal reservation estimation, QoS (interference)-aware predictive scheduling to maximize resource efficiency of multi-dimensions (CPU, Memory, Network, Disk I/O), and strictly guarantee service level agreements (SLA) for long-running workloads.

Finally, we introduced a large-scale co-location techniques of executing long-running and other workloads on the shared global datacenter infrastructure of Alibaba Group. It effectively improves cluster utilization from 10% to averagely 50%. It is far more complicated beyond scheduling that involves technique evolutions of IDC, network, physical datacenter topology, storage, server hardwares, operating systems and containerization. We demonstrate its effectiveness by analysis of newest Alibaba public cluster trace in 2017. We are the first of works to reveal the global view of scenarios, challenges and status in Alibaba large-scale global datacenters by data demonstration, including big promotion events like "Double 11".

Data-driven intelligent scheduling methodologies and effective infrastructure co-location techniques are critical and necessary to pursue maximized multi-resource efficiency in mod-

ern large-scale datacenter, especially for long-running workloads.

AUTOBIOGRAPHICAL STATEMENT

Guoyao Xu is a Ph.D. candidate of Department of Electrical and Computer Engineering at Wayne State University. He received the B.S. degree in Electronic Information Engineering from XiDian University, Xi'an, China, in 2012. He received the Master degree in Computer Science from Wayne State University, Detroit, United States, in 2013.

His research interests include intelligent scheduling and resource management in large-scale datacenter, resource scheduling and performance optimization of distributed big data systems, cloud computing, distributed machine learning systems, and machine learning, green computing and energy efficiency. He has published one journal paper in Sustainable Computing: Informatics and Systems, and submitted two to IEEE Transactions on Parallel and Distributed Systems. He has published 7 papers in proceedings of leading international conferences, and 3 submitted papers under review.

He has made several invited presentations in Alibaba Group, Harvard University, 2017 ACM SoCC conference, ACM SIGOPS 2017 ChinaSys conference, 2015 and 2016 Big Data & Business Analytics Symposium in Detroit. He has been a graduate research assistant and associate for eight consecutive years. He has also been awarded as one of the top 50 teams of 2016 TEEC CUP UCAHP North American Startup Contest and NSF Innovation Corps (I-Corps) certification as an entrepreneur lead.