Reliable Computing Service in Massive-scale Systems through Rapid Low-cost Failover

Renyu Yang, *Member IEEE*, Yang Zhang, Peter Garraghan, Yihui Feng, Jin Ouyang, Jie Xu, *Member IEEE*, Zhuo Zhang, Chao Li

Abstract—Large-scale distributed systems deployed as Cloud datacenters are capable of provisioning service to consumers with diverse business requirements. Providers face pressure to provision uninterrupted reliable services while reducing operational costs due to significant software and hardware failures. A widely adopted means to achieve such a goal is using redundant system components to implement user-transparent failover, yet its effectiveness must be balanced carefully without incurring heavy overhead when deployed – an important practical consideration for complex large-scale systems. Failover techniques developed for Cloud systems often suffer serious limitations, including mandatory restart leading to poor cost-effectiveness, as well as solely focusing on crash failures, omitting other important types, such as timing failures and simultaneous failures. This paper addresses these limitations by presenting a new approach to user-transparent failover for massive-scale systems. The approach uses soft-state inference to achieve rapid failure recovery and avoid unnecessary restart, with minimal system resource overhead. It also copes with different failures, including correlated and simultaneous events. The proposed approach was implemented, deployed and evaluated within Fuxi system, the underlying resource management system used within Alibaba Cloud. Results demonstrate that our approach tolerates complex failure scenarios while incurring at worst 228.5 microsecond instance overhead with 1.71% additional CPU usage.

Index Terms—Cloud Computing, Resource Management, Reliability, Services, Failover.

1 Introduction

LARGE-scale distributed systems deployed as Cloud datacenters are composed of thousands of heterogeneous server nodes capable of provisioning a wide variety of services to consumers with diverse business requirements [1]. Datacenters continue to see increased uptake, represented by a 69% increase of Cloud datacenter traffic to over 5.3 Zettabytes, with over 118 million workloads deployed in 2014 [2]. The core aspect of such system operation is the resource manager, responsible for assigning available resources to application requests. Modern resource managers including those in Mesos [3], YARN [4] and Fuxi [5] are used to provision and schedule tens of millions of services daily and are critical for commercial, personal and research pursuits.

Dependability is a key concern for Cloud resource managers due to increasingly common failures which are now the norm rather than the exception caused by the enlarged system scale and complexity [6] [7] [8], different workload characteristics, and plethora of faults types that can activate. Failures within a resource manager have the potential to cause significant economic consequences to Cloud providers due to loss of service to consumers [9], and could affect services provisioned to millions globally in the event of catastrophic failures. In this context, fault tolerance is often an effective means in enhancing the reliability of Cloud systems. It can be implemented through a variety of techniques, including diversity of data and design, recovery blocks [10] and checkpointing [11]. Within the context of Cloud resource managers, such techniques are required to effectively scale to thousands of servers, with acceptable overhead and impact to system performance. The latter is achieved through user-transparent failover, which is defined as a technique to recover a service without noticeable changes to the provisioned service perceived by consumers [12].

In practice, many resource managers for large-scale distributed systems follow a simple but costly approach that terminates and restarts all the tasks and services related to the failed component [3] [4] [13] even if some of them do not exhibit faulty behaviour. However, this approach results in extra resource costs, including additional requests for resource and the re-computation of tasks [5]. It is particularly inefficient for long-running services, especially when they are nearing completion. Yarn 2.6.0 has recently developed a work-preserving mechanism for certain critical components, e.g. the resource manager and the node manager [14] [15]. However, there is no support for application-level restarts that could handle more general and sophisticated failure scenarios. In fact, failure coverage is an important measure of the effectiveness of any fault-tolerant technique [16]. Moreover, a large body of failover schemes in Cloud systems focus primarily on crash-stop failures [16], without specifically dealing with timing failures – another important consideration for failover for increasingly popular time-sensitive services including video streaming, gaming, and real-time analytics. Approaches that assume no failure

- *Renyu Yang is with Beihang University, Beijing, China (Email: yangry@act.buaa.edu.cn).*
- *Peter Garraghan and Jie Xu are with University of Leeds (Email: {p.m.garraghan,j.xu}@leeds.ac.uk).*
- *Yang Zhang, Yihui Feng, Jin Ouyang, Zhuo Zhang, and Chao Li are with Alibaba Cloud Inc. (Email: {xingbao.zy, yihui.feng,jin.oyj, zhuo.zhang, chao.li}@alibaba-inc.com)*
- *Peter Garraghan is the corresponding author.*

Manuscript received Jun 10, 2015; revised Dec 31, 2015; accepted Mar 3, 2016.

correlation among faulty components are obviously inappropriate for addressing simultaneous component failures. Given the frequency of simultaneous component failures in modern datacenters, as reported in [17] [18], there is an urgent need for innovative methods specifically designed for handling them. Finally, although many fault-tolerant approaches are proposed by academic researchers, they are often limited to small-scale experimentations [19] [20] [21] and simulation [22] [23], resulting in difficulties in validating their effectiveness and understanding their operational constraints and intricacies at scale.

This paper describes a novel autonomous fault-tolerant technique for user-transparent failover in massive-scale Cloud systems. Our approach is composed by a hybrid checkpoint recovery technique that uses a combination of light-weight hard-state and soft-state inference. Specifically, failed components can leverage inferred states from interacting components for recovery. This approach allows for demonstrably low-cost checkpoint recovery of components and services, capable of tolerating simultaneous failures of system components, including both crash-stop and timing failures. Through soft-state inference and resource reservation, we can achieve minimized running worker eviction and maximized fault coverage under different failure combinations. The experimental results show that only microsecond- level additional overhead is imposed during failover for task execution time with a minor increment to system CPU utilization. Component recovery adds at worst minute-level overhead under both single and simultaneous component failure scenarios. Our approach has been integrated into Fuxi system [5]; a two-level resource management and job scheduling system used in Alibaba Cloud Inc. The technique is deployed within a production Cloud datacenter comprising of over 5,000 servers and 42 millions submitted tasks, and is currently used by Alibaba to handle Internet-scale workloads. The main contributions of the paper can be summarized as follow:

- **Design of soft-state inference for component recovery:** We present the philosophy and architecture of a novel approach for component failure recovery that collects and exploits states collected from neighbouring components instead of solely relying on hard-state periodically collected from dedicated backup systems.
- **Comprehensive solutions to various failure scenarios in modern large-scale systems:** Our work investigates various failure scenarios in detail which are typical in large-scale distributed systems, as well as provides solutions to practical failover for both single and simultaneous component failures, such as crash-stop and timing failures, in order to increase the failure coverage of the whole system.
- **Reduced worker eviction based on state inference and resource reservation:** The failure in a master or an agent does not result in automatically its fault-free workers to be evicted. We use state-inference to identify late-timing or inaccessible agents, reducing the overhead of checkpointing, so as to reserve more assigned resources for running workers.

The rest of this paper is organized as follows: Section 2 firstly introduces the challenges and motivation; Section 3 outlines our design philosophies based on the proposed soft-state inference; Section 4 details the proposed component failover mechanism and system implementation; Section 5 presents the system evaluation; Section 6 discusses related work; Section 7 discusses the conclusions of this work and future research directions.

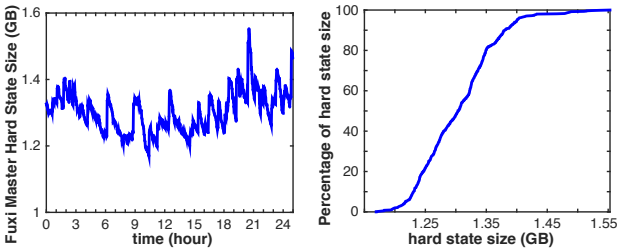
2 Motivation

An important decision when constructing fault-tolerant systems is to define the fault and failure coverage required to provide correct service. This allows for deployed techniques to achieve the maximum effectiveness of fault tolerance, given the developmental and operational resources available. Internet-scale systems are typically composed by hundreds of thousands to millions of heterogeneous and interacting components (e.g. the resource manager, service framework, and computational applications), leading to the manifestation of different types of faults. Faults may occur simultaneously and in any aspect of system operations ranging from application to hardware, and may have a wide variety of causes, including insufficient memory (OOM), overweight system utilization, performance interference, network congestion, server faults (e.g. disk, middleware software), and applications crash or hanging etc [8].

Crash-stop and timing failures are quite predominant in Cloud datacenters [16], resulting in the loss of dependability for provisioned services, as well as noticeable performance degradation. Failures in the critical components, e.g. the Cloud resource manager, significantly affect the reliability of all the services running on the infrastructure. In recent years such events have been reported to cost Cloud providers within the realm of millions of dollars [24]. It is imperative to identify susceptible components and deploy fault-tolerant techniques in order to mitigate the negative impacts.

Apart from the simple re-start of tasks, there are some established methods for software fault tolerance such as recovery blocks [10], N-version Programming (NVP) [25], N-self checking programming [26]. Replication is also widely-used, where identical copies of a component are deployed within the system. A replica will become the primary component in the event of component failure through hot, warm, or cold standby [27] [28]. However, it is practically infeasible to apply a replication-based method to each component within a system composed of millions of components and jobs (each composed of thousands of executing tasks). This is due to the significant resource cost incurred, an issue which is further debilitating in periods of compute resource scarcity within the system.

Checkpoint-based state recovery is another established failover approach that saves runtime states into a disk permanently, often known as *hard state*. Although it imposes relatively low recovery overhead, typically a few seconds, the total overhead due to checkpointing is often extremely high, which includes the costs of disk space, communications, and checkpointing operations. This approach is not suitable for latency-sensitive services and jobs. Figure 1 presents the total state size recorded within a real production cluster over 24 hours of operations. It is observable that at most 1.7 GB states (1.36 GB on average) are stored periodically for future recovery. With the cost of network transmissions as well as R/W locks and unlocks, reading or writing each checkpoint



(a) Hardstate Checkpoint Size (b) Hardstate Size CDF

Fig. 1. Snapshot state size for hard state checkpointing over 24 hours in a production cluster

takes on average 681ms. In particular, all checkpoints have to be updated fully each time to avoid incorrect state restore and execution, leading to a large number of frequent data transmissions and writes to the permanent store. In fact, although the recovery time from a snapshot is 3.89s on average, the resource manager cannot handle any request during the process of performing snapshots due to state locking. This is unacceptable in practice because modern large-scale distributed applications typically have a requirement for their requests to be handled within milliseconds [5]. When a system operates in the absence of failures, this approach is particularly inefficient and infeasible.

In order to address the issues we have discussed so far, we present a novel approach in this paper that uses soft-state inference to reduce the overhead of checkpointing, i.e. saving only the minimum amount of hard states into a disk. This is particularly desirable for large-scale distributed systems. Component recovery takes time but our solution offers reasonably rapid recovery with the options of customisable operations of state inference when needed. Our approach also addresses various failure scenarios in a large-scale system, with the aim to increase its failure coverage and overall dependability.

3 Reliability Design for Massive Scale

3.1 Massive-scale System Case: *Fuxi* System

Before presenting our proposed failover approach, it is important to understand the types of components within a large-scale distributed system. To facilitate this, we provide a high-level discussion of the Fuxi system - a resource management and job execution system deployed within Alibaba. Fuxi follows a master-slave architecture composed by three components: the central resource manager (Fuxi Master), the node agent manager (FAgent) per node and application manager (Application Master or AppMaster) per application.

Similar to terminologies in Yarn [4], an application is defined as a single job in the classical sense of Map-Reduce, DAG (Directed Acyclic Graph) or Spark job etc. The AppMaster is responsible for each application-specific execution logic. Different computation frameworks can be implemented on top of Fuxi system and such applications are submitted into the cluster via a client (see step 1 in Figure 2). In Fuxi, *FuxiJob* (Job for short) is a typical application and each job is composed of running tasks which constitute a DAG. Each node within a DAG represents a task and the edge depicts the pipeline and the data shuffle between two consecutive tasks. A *task* is divided into parallel instances where each individual task *instance* (the basic logic unit of computation) is executed within an isolated container *worker*. Tasks are under the control of the AppMaster, which is responsible for application specific task division, resource request and subsequent task execution.

Fuxi Master is responsible for negotiation between the resource requests from AppMasters, and the available resources within the infrastructure. Due to the dynamicity of workload requirements and resource utilization levels within the infrastructure, Fuxi has been implemented to collect resource request and free resource reclaim information incrementally in order to perform fair and efficient resource allocation. In the occurrence of a specific resource being no longer required by an AppMaster due to the completion of the current execution phase, Fuxi Master is responsible for revoking and rescheduling these resources to another application in a timely manner. After resource requests proposed by different AppMaster are decided, the scheduling responses containing assigned resources will be delivered to each AppMaster. AppMaster can use these resources to determine the concrete computation plan with the specified guaranteed resources. FAgent will then launch requisite computation processes according to this plan.

The Fuxi Agent (FAgent) is an agent that runs on each individual node, and serves two purposes: a) collect information and status of both available and occupied resources periodically and report them to the Fuxi Master for further resource scheduling decisions; b) ensure that application processes are executed normally within the nodes. The latter is made possible through the aid of the process launch, monitor, and isolation etc. It is noteworthy that for the security control, FAgent launches a worker *only when* it receives the same amount of granted resource from Fuxi Master and the execution plan from AppMaster.

Figure 2 illustrates how the components above interact with each other: (1) **Application Submission**: A consumer first submits a request to the Fuxi Master to launch an application (i.e. a MapReduce job) via a job description containing data, detailing the application type, package location and application-specific information; (2) **AppMaster Launching**: Fuxi Master locates an FAgent with sufficient resources for the AppMaster to launch on, and requests the FAgent to start communication with the AppMaster. Once this communication is established, the AppMaster retrieves the application description and determines the resources requires for the different stages of job execution, including appropriate parallelism, the level of granularity required for allocation and preferred locality of workers etc; (3) **Resource Request and Response**: The AppMaster transmits the requirement to Fuxi Master incrementally and waits for resources to be granted or revoked; Meanwhile, Fuxi Master will also notify corresponding FAgents with the granted resource changes. (4) **Execution Plan**: Once the resources have been assigned to the AppMaster, it sends its concrete work plan to the corresponding FAgents. The execution plan contains the required information in order to launch a specific process such as the binary package location, limits of resource usage and parameters for starting; (5) **Execution**: Once FAgent has received the work plan and corresponding permitted resource from Fuxi Master, it starts the application workers. FAgent monitors the status of workers and will

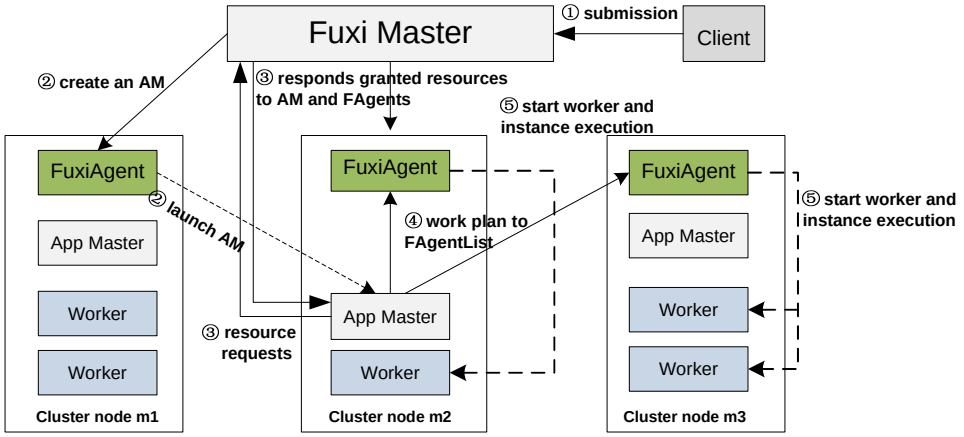


Fig. 2. An overview of Fuxi components and the life-cycle of a job submission, resource request, request handling, and worker execution.

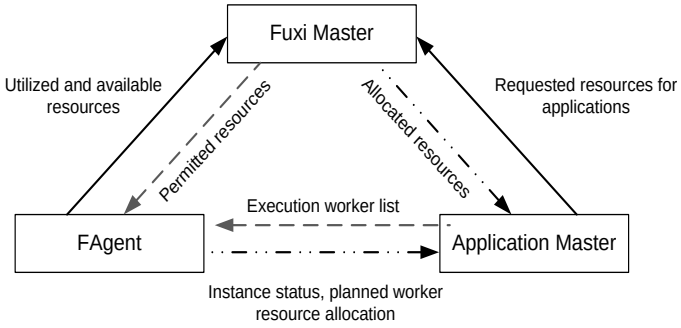


Fig. 3. Soft-state inference applied to Fuxi system components.

restart them in the event of crash failures. Once the worker has finished execution or is no longer needed, the AppMaster makes an incremental request to return resources. More details about Fuxi system (operational details and scheduling performance evaluations) can be found in [5].

It is worth emphasizing that this described architecture is applicable to most two-level schedulers. The Fuxi Master, FAgent, AppMaster in our architecture can be implemented as RM, NM and AM in Yarn, and also can be fulfilled as Mesos Master, Mesos Slave, and Framework Scheduler respectively. Therefore, the proposed techniques for component failover in this paper can be directly applicable within other systems.

3.2 Soft State Inference: Basic Concepts

In order to tackle the failover problem at scale, we leverage a *hybrid* approach for state recovery in order to achieve low-cost component failover. This includes using minimized *hard state* such as meta-data and information persistently stored within a node locally, distributed file system or distributed coordination service [29] [30]. Furthermore, we present for the first time the concept of recovering failed component state via soft state inference where *soft-state* is defined as state data that contains run-time component status, not persistently or deliberately stored for purposes of periodical checkpointing. The fundamental concept of soft-state inference (see Figure 3), is that failed components within the system are capable of recovering their states by collecting data which is currently being used by other components within the system. In essence, we leverage the distributed memory to store each component states which can constitute the overall system states.

Consequently, this mechanism is able to reduce the checkpointing overhead (discussed in Section 2) with minimal additional overhead from collecting distributed soft states. For example, when Fuxi Master fails, it is able to recover the allocated resources from Fuxi Agents whose main function is to launch workers according to granted resource while monitoring the resource status within a physical node. Another aspect of resource soft states is waiting requests, which can be obtained from running AppMasters. Recovery of other components is similar to this process and the arrow in Figure 3 represents the direction of main required states collection of each component.

The inference is also embodied by information deduction from multiple state provenance. Given the complexity of interactions among components that each contains its local information, there is overlapping information held by different components. In practice, each component contains a fraction of the holistic soft states. As a result, the soft state inference has to be conducted using redundant information. Therefore, we assign each component type with a **[C1:] Degree of Confidence (DoC)**, making it possible to identify and infer the required state dependent on the specific failure scenario. In our design, we regulate the prioritized DoC order: *FuxiMaster>FAgent>Worker>AppMaster*. Specifically, Fuxi Master owns the highest priority and confidence level because it is the dominator to match requests with system resources, holding the global information of both resource and running workloads. On the other end of the spectrum, we assume that information sent by AppMaster has a higher probability of less accurate information due to incorrect or malicious configuration. Consider an example of asynchronous messaging among running components in a distributed environment with discordance of message arrivals. For instance, at the timepoint when Fuxi Master begins to recover, the assigned resource amount from the AppMaster is likely to be inequal to the granted resource amount provisioned by a FAgent if FAgent has not yet received a specific resource granting message on time. To avoid issues incurred by inconsistent information during state inference, we preferably adopt FAgent information, as

worker initialization and execution are directly controlled by FAgent. These assumptions are widely utilized in all inference-based fault-tolerant techniques.

3.3 Design Considerations for Effective Failover

To maximize service reliability whilst minimizing detrimental effects to service performance within large-scale systems, we extend Fuxi with Fuxi-FT; containing several fault-tolerant techniques in conjunction with inference to illustrate a feasible design and implementation towards reliable service execution for effective computing systems at scale.

(1) Minimized Worker Eviction: From our experience in massive-scale systems, resource overhead since eviction, and re-computation of non-faulty workers produces a substantial amount of waste as studied in [18] [31]. In addition, long-services are disproportionally affected due to restarting worker execution, leading to suffered QoS. Such behavior results in increased strain on the resource manager, which has to handle more requests and reschedule workers onto nodes, causing reduced component performance as well as increased failure probability. As a result, the first consideration and objective of our approach is **[C2:] reduced worker eviction by resource reservation**. This is achieved through loose-coupling master or agent behavior from its respective workers during execution. Specifically, this entails that failure occurrence of a master or agent does not result in its non-faulty workers to be automatically evicted. For example, to tolerate timing failures, Fuxi Master will attempt to preserve the assigned resource for running workers as if timing-out FAgents or AppMasters are still executing rather than directly evicting and re-scheduling them. In this manner, such faults will have minimal interference with consumers perceived reliability.

(2) Customized Recovery Time Cost with Degraded Service Level: As mentioned previously, the additional overhead cost is mainly dependent on the collection and required boundary of state information completeness. Incomplete information might appear due to timing-out components unable to contribute their states in time. The collection time also closely depends on cluster scale, application number, and application-specified configurations. For instance, increased application number signifies a larger amount of states to collect and the requisite time. On one hand, longer waiting time can potentially lead to the mitigation of soft states incompletion, but resulting in extra end-to-end recovery time. On the other hand, insufficient collection time leads to incomplete states and subsequent degraded service level (such as job extended running time due to worker eviction). Thus, it is necessary for cluster administrators to estimate and customize the information aggregation time, **[C3:] allowing possible tradeoffs between recovery cost and various levels of degraded service**.

(3) Comprehensive Coverage of Faults and Handling: Components within the Cloud resource manager are likely to experience different types of faults ranging from crash-stop to late timing failure, as well as have different underlying root causes. As multiple components tend to fail simultaneously and also exhibit correlation, we intend to achieve **[C4:] maximized fault coverage from both faults mode and fault handling coverage respectively**. Firstly, our approach must be designed to tolerate such occurrences due to the significant detrimental impact to service reliability and performance. This is achieved through expressing simultaneous component failures as unique failover scenarios compared to single components. Secondly, we divide our approach into separate subsections with the precise means to do so dependent on a given failure scenario. For example, within the context of Fuxi, the Fuxi Master is a more critical component compared to that of AppMaster, requiring different techniques and precise engineering practices compared to other components when performing state recovery.

TABLE 1
Basic state terms when performing component failover.

Terms	Descriptions
ResCap	The capacity or amount of resources that has been admitted to a FAgent and can be used to launch tasks.
FMFullAsgnedResToAM	Resources that have been assigned to corresponding AppMasters.
FMFullResCapToFA	All ResCaps to a specified FAgent
AMFullReqResFromFM	Resources that the application still requests and waits for allocation from Fuxi Master.
AMFullAsgnedResFromFM	Assigned resource to a specific AppMaster that it uses to execute tasks.
FAgentFullCapFromFM	Permitted resource to a specific FAgent that it uses to launch tasks.

4 Component Failover Mechanism

4.1 Single Component Failover Mechanism

In this section we discuss in detail the failover process for single component failure scenario.

4.1.1 *Fuxi Master Failover*

In the scenario of a failure within the central resource manager Fuxi Master, a three-replica scheme is deployed composed by a primary component and non-active replicas on warm standby (i.e. the replica is deployed and connected to the system, yet does not actively mirror operations performed by the Fuxi Master). The primary and replica Fuxi Master are typically deployed on separate physical nodes. In the event of failure within the primary Fuxi Master, the standby master will take over all its functionalities.

In addition, Fuxi Master loads hard state and performs soft-state inference. Hard state includes the job description, task topology, resource requirements and the location and ID of applications (currently executing within the system). Soft-state is inferred from information reported by the FAgents and AppMaster as shown in Figure 3. For FAgents, this includes information pertaining to current resource state of physical nodes, specifically the current granted resources (*FAgentFullCapFromFM*) to run tasks and the available resources within nodes. For the AppMaster, this includes the remaining executing worker number and total resources still required (*AMFullReqResFromFM*) in order to complete application execution. With this information, Fuxi Master is able to restore the application list and remaining instances that require execution. It is noteworthy that Fuxi Master does not need to recover the already assigned

resource bitmap from AppMasters by *AMFullAsgnedResFromFM*, due to the equivalent information provisioned by *FAgentFullCapFromFM* which has higher DoC ***[C1]***.

In summarization, the whole failover consists of two phases – (1) information aggregation: Fuxi Master collects the necessary information from both hard and soft state; (2) recovery phase: it repopulates the global resource allocation and utilized resources based on aggregated states. According to ***[C3]***, we configure the time as 60 seconds for cluster of thousands of machines according to our practical experiences.

4.1.2 FAgent Failover

The FAgent has two different counterplans for performing failover in the event of crash-stop and late timing failures respectively. For crash-stop failures, the FAgent collects soft-state from monitoring data of currently executing instances in real-time and then requests a full worker list from each corresponding AppMaster. This is particularly effective as the FAgent itself does not need to contain any persistent data pertaining to the current node and worker status. FAgent then collects soft-state of permitted resources (*FMFullResCapToFA*) from Fuxi Master before finally completing failover. Additionally, the granted resource is restored from the local checkpoint when Fuxi Master could not provide the information due to a simultaneous failover (see Section 4.2). Due to higher probability of a local checkpointing containing invalid state, it it preferable restore state using information contained within the Fuxi Master if possible.

When performing failover for late timing failures that can cause large amounts non-faulty worker eviction, Fuxi Master will hold the allocated resources for all workers within the node where FAgent fails for a definitive period of time. Meanwhile, the Fuxi Master will send a list of failed FAgents to affected AppMasters, allowing them to reserve the necessary resources until the FAgent recovers.

4.1.3 AppMaster Failover

In the event of crash-stop failures, the AppMaster will restart on the same node, or will be rescheduled to another node in the event of a hardware fault. It is worth noting that failure of the AppMaster does not cause its related workers to terminate (which is advantageous for long running workers that are nearing completion). The AppMaster will recover the hard states from a snapshot including task execution and internal data structures from the distributed file system, enabling reconstruction of the task topology and data shuffle and pipeline between tasks. Events such as task instance terminations or failures recorded by application logs are also leveraged as hard state. During its recovery, soft-state is collected: Fuxi Master will concurrently synchronize the allocated resources (*FMFullAsgnedResToAM*) to the failed AppMaster whilst FAgents report current workers' status, as well as all worker computation plans sent from the application before. If a worker has been completed within the component recovery time period, the AppMaster will identify the worker from the record logs and return these resources to Fuxi Master.

For late timing failures, the AppMaster will be marked as timed out by the Fuxi Master after a configured period of time, resulting in rescheduling onto a new physical node based on *FMFullAsgnedResToAM* information it holds. In practice, the timeout upper bound could be set as infinite, indicating that the Fuxi Master will wait until the AppMaster has recovered. All relevant terms used in the above approaches are summarized in Table 1.

TABLE 2
Algorithm Variables

Variables in FuxiMaster	Formalized Description
Total FAgent number	n
Total AM number	m
FAgents set with states	$A = \{A_1, A_2 \ldots, A_n\}$
Granted resource to FAgents	$Grt = \begin{bmatrix} Grt_{11} & \ldots & Grt_{1m} \\ \vdots & \ddots & \vdots \\ Grt_{n1} & \ldots & Grt_{nm} \end{bmatrix}$
Resource on i^th agent to j^th app	Grt_{ij}
All Resource to j^th app	Grt_{*j}
AppMasters set with states	$AM = \{AM_1, \ldots, AM_m\}$
AppMaster Requested res in FM	$Req = [Req_1, \ldots, Req_m]^T$
ReqRes on agents of i^th app	$Req_i = [r_1, \ldots, r_n]$
AppMaster Assigned res in FM	$Asgn = [Asgn_1, \ldots, Asgn_m]^T$
AssignedRes on agents of i^th app	$Asgn_i = [a_1, \ldots, a_n]$
ScheduleUnit (SU) of i^th app	$SU_i = (Asgn_i, Req_i, Config)$

Algorithm 1 : Faulty FAgent Recovery, Inference-based Detection and Resource Reservation

1: Rebooting FAgents set A' and $A' \subset A$
2: **for** each A'_i in A' **do**
3: $A_i \leftarrow A'_i \cup (FAFullCapFromFM$ + checkpoint $\Delta c)$
4: **end for**
5: $FuxiMaster$ starts failover()
6: **for** each AM_i in AM **do**
7: $Asgn_i \leftarrow updateAssignedRes(AM_i)$
8: $Req_i \leftarrow updateRequestedRes(AM_i)$
9: $SU_i \leftarrow SU_i \cup (Asgn_i, Req_i, Config)$
10: **end for**
11: $D = Asgn^T - Grt$
12: Timeing-out FAgent set $A_t \leftarrow \varnothing$
13: **for** each D_{*j} in D **do**
14: **if** $||D_{*j}||_0 == 0$ **then**
15: $A_t \leftarrow A_j$
16: **end if**
17: **end for**
18: $FuxiMaster$ reserves resource on A_t
19: $FuxiMaster$ sends the list A_t to each AM_i in AM

4.2 Multiple Simultaneous Component Failover

With the increased complexity and scale of computing environments, it will undoubtedly lead to more timing failures and simultaneous fault conditions among multiple components. Therefore, we design and implement an exception handling mechanism to cope with these failures. We focus on explaining the failure combinations in which Fuxi Master fails, as the non-faulty Fuxi Master could be easily used to facilitate the recovery of any other faulty components rapidly and these can be deduced to single failover cases discussed in Section 4.1.

4.2.1 Fuxi Master and FAgent Failures

If both Fuxi Master and FAgent fail and cannot recover in a timely manner, the AppMaster will suppose all resources be revoked, thereby evicting all running instances which violates ***[C2]***. Specifically, FAgent failure can be categorized into crash failure or a late timing failure. In this context, FAgent is unable to perform failover due to missing key information from the Fuxi Master, while Fuxi Master is also unable to

collect state provided by the FAgent. To solve this problem, we perform a light snapshot of *FAgentFullCapFromFM*. The incremental checkpointing(Δc) is performed once there is a state change within the permitted resources. Consequently, all FAgents are capable of recovering prior to the reboot of Fuxi Master's recovery phase.

As shown in Table 2 and Algorithm 1, once Fuxi Master commences failover, it first differentiates between inaccessible/timed-out agents and normal agents by aggregating *AMFullReqResFromFM* from Applications Masters and *FAgentFullCapFromFM* from non-faulty FAgents. This process identifies FAgents which do not recover before completing Fuxi Master failover (line 10-16). We use this state-inference to identify late-timing FAgents instead of frequent checkpoint. This is due to heartbeats and statuses of thousands of agents often change dynamically, resulting in large amount of updating overheads when conducting snapshots. Afterwards, Fuxi Master will reserve the corresponding resources for the running workers on the machine experiencing faulty behavior in accordance to ***[C2]*** until the FAgents reconnect to the cluster within a configured time period. Meanwhile, a message containing a timeout agent list is sent to all non-faulty AppMasters. The AppMaster then reserves and retains the resource as if the agents are non-faulty. If FAgents are unable to reconnect before a predefined time threshold, the AppMaster can choose to reclaim the resource from workers by evicting them. In reality, different AppMasters have different policies. For example, a long-running service master tends to wait for the faulty agents to reconnect, attempting to keep workers running as long as possible. However, the AppMaster for short jobs might tend to reclaim the resource and evict workers more rapidly. If FAgent recovers prior to the time threshold, it sends a heartbeat message to the Fuxi Master to report the resource availability.

Algorithm 2 : Faulty AM Resource Recovery and Inference-based Resource Reservation in Multi-Component Failures

1: restarted AM set: $AM' \subset AM$
2: **for** each AM'_i in AM' **do**
3: $AM_i \leftarrow$ its running workers
4: **end for**
5: $metaAM \leftarrow$ AM lightweight checkpoint
6: $TimeoutAM \leftarrow metaAM - AM$
7: **for** each AMt'_i in $TimeoutAM$ **do**
8: $FuxiMaster$ create a $MockSU$ for resource reservation
9: $MockSU \leftarrow synthesize(Grt^T_{*i}, DefaultReq)$
10: $SU_i \leftarrow SU_i \cup MockSU$
11: **end for**
12: $FuxiMaster$ starts failover() and collects AM runtime info
13: re-connected AM set: $AM'' \subset AM$
14: **for** each AM''_j in AM'' **do**
15: $FuxiMaster$ removes $MockSU$ and update states with AM''_j
16: $Asgn_i \leftarrow updateAssignedRes(AM''_j)$
17: $Req_i \leftarrow updateRequestedRes(AM''_j)$
18: $SU_i \leftarrow SU_i \cup (Asgn_i, Req_i, Config)$
19: **end for**

4.2.2 Fuxi Master and AppMaster Failures

In general, AppMaster failover is much more light weight and simplistic compared to the Fuxi Master. As a result, it is typically possible to complete AppMaster failover prior to the Fuxi Master commencing failover. As there is no information provided from the Fuxi Master, the AppMaster instead uses the status of related workers according to ***[C1]*** in order to restore assigned resource state (see Algorithm 2 line 2-4). After the collection phase, Fuxi Master will compare the collected application information with the checkpoint file recording the metadata of applications before rebooting. Fuxi Master is thus able to identify applications that have not reported back and then creates synthesized Schedule Units (MockSU) according to the corresponding granted resource amount from FAgents (line 8). Schedule Unit(SU) is defined as a data tuple $(Asgn_i, Req_i, config)$ inside i^{th} AppMaster representing requesting and assigned resource, and other metadata such as priority, quota group etc. SU synthesis could be done based on our pre-condition that the resources granted to running workers on different FAgents can act as surrogates to the amount of allocated resources. Additionally, the MockSU is used to reserve the allocated resource and can not be preempted by design.

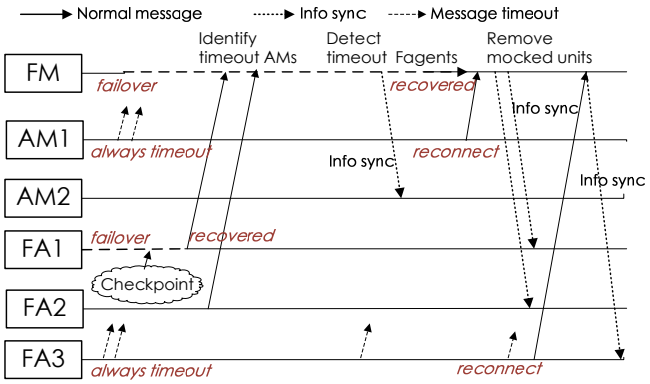


Fig. 4. Failover workflow for Fuxi Master crash, AppMaster timing failure, and FAgent crash or timing failure. In this example, FAgent 1 (FA1) crashes while FA3 experiences a timeout during Fuxi Master failover.

Consequently, these SUs assist with synthesizing the original states of the scheduler, minimizing the occurrence of worker eviction due to application late-time failure caused by timeout during Fuxi Master failover. Afterwards, if the AppMaster reconnects to the system, the Fuxi Master will substitute the MockSU with the real run-time values. If not, Fuxi Master will remove all pertaining SUs and reclaim allocated resources.

4.2.3 Full Component Failure Combinations

It is possible for all types of components to experience failure within a failover period. In order to achieve full fault coverage ***[C4]***, we characterize this type of faiure scenario as: Fuxi Master crashes and reboots, with FAgent failing (crash or timing failure) during which (1) AppMaster crashes; or (2) AppMaster timing-out.

For the first case, AppMaster has to recover its states from collecting meta-data of executing workers for recovering its assigned resources. The reason is because the faulty Fuxi Master cannot handle any request and not all FAgents can provide precise information due to a proportion of components exhibit faulty behavior. According to ***[C1]***, AppMaster can still restore most of the assigned resource and computation intermediate results scattered across machines within the cluster from running workers instead. If Fuxi Master is non-faulty, it has the highest DoC state and does not need to rely on the workers to fetch the assigned

TABLE 3
Soft-state information required for each failure scenario. AR = After Reconnect, FC = From Checkpoint

No.	Failure Combinations (StateMachineNode)	FMFullAsgnedRes	FMFullCap	FAFullCap	AMFullResReq	AMFullAsgnedRes
1	FM Crash (S1)	-	-	√	√	-
2	FAgent Crash (S3)	-	√	-	-	-
3	FAgent Timing (S3)	-	√	-	-	-
4	AM Crash (S2)	√	-	-	-	-
5	AM Timing (S2)	√	-	-	-	-
6	AMCrash/Timing+FAgentCrash/Timing (S2S3)	√	√	-	-	-
7	FM Crash+AM Crash (S4)	-	-	√	√	-
8	FM Crash+AM Timing (S4)	-	-	√	AR	-
9	FM Crash+FAgent Crash (S5)	-	-	-	√	√
10	FM Crash+FAgent Timing (S5)	-	-	-	√	√
11	FM Crash+AM Timing+FAgent Timing (S6)	-	-	AR	AR	-
12	FM Crash+AM Timing+FAgent Crash (S6)	-	-	√ (FC)	AR	-
13	FM Crash+AM Crash+FAgent Timing (S6)	-	-	-	√	√
14	FM Crash+AM Crash+FAgent Crash (S6)	-	-	√ (FC)	√	-

resource information. In reality, it is much more costly compared to communicating with a non-faulty Fuxi Master due to the additional network cost when communicating to potentially thousands and tens of thousands of workers scattered across the cluster. After faulty AppMasters' recovery has completed, the details described in Algorithm 1 could be conducted. It is noteworthy that we do not rely on the information from AppMaster to recover the agents as assigned states could be distorted by invasive applications.

Furthermore, we describe the other scenario where AppMaster is experiencing timing failure as depicted in the failover workflow Figure 4. Firstly, Algorithm 2 is performed to deal with faulty AppMasters. Apparently, there is a high dependency with the number of the active FAgents during these resource occupations as the synthesis is conducted primarily based on FAgents' *FAFullCapFromFM*. Therefore, Fuxi Master will attempt to restore the granted resource to i^{th} application (Grt_{*i}) as much as possible. For example, in Figure 4 the granted resource on FA3 will be temporarily missing as no granted resource information can be provided by the timing-out FAgents. Despite this, workers are able to run as plan without any impact. At the same time, steps described in Algorithm 1 will be conducted to preserve the scheduling resources on faulty FAgents.

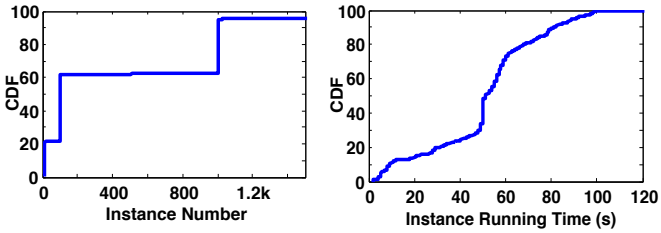
Table 3 summarizes the necessary message and required soft-state during the synchronization under different exception scenarios. To capture and characterize all types of simultaneous failures, we leverage the Finite-State Machine (FSM) to describe each failure type and the relationship among different types. In particular, each combination corresponds to a state in FSM. For the description clarification, we merge analogous states together and the state transitions are illustrated in Figure 5.

4.2.4 System Implementation Using FSM

To apply the proposed mechanism into a production resource manager, we have developed a fault-tolerant management layer (Fuxi-FT) within Fuxi System and modified the corresponding module to support the proposed failover approach and component interaction message protocol. Components collect and propagate required information collaboratively and suitably by message delivery and response to perform inference-based resource reservation and rapid component failover.

In Fuxi-FT, *EventSensor* can identify which type of event currently is active, based on the combined information of heartbeat, system status, health checker etc. The incoming event will be registered into a global bulletin board and instantly enqueued into *EventDispatcher*, waiting for fault handling. In fact, the dispatcher is responsible for identifying the fault type of the state, and routing the fault handling request to relevant handlers. In Fuxi-FT, each handler is implemented and applied in hot-plugging manner. The techniques discussed above are used to handle different scenarios accordingly. For example, once the ceasing of periodical FAgent heartbeat is detected, *FAgentTimingFailure* event will be generated. The dispatcher will trigger the corresponding FAgent recovery process (discussed in Section 4.2.1) upon receiving the event. After the failover process completes, the current state will be transit into the subsequent state according to the FSM depicted in Figure 5.

The proposed approach can be easily integrated into other two-level scheduler such as Mesos and Yarn system etc. In reality, there are no fundamental difference between basic concepts, architectural design etc., resulting in a direct mapping relationship among them. For example, Fuxi Master and FAgent correspond to the RM and the NM in Yarn. Therefore, soft states could be interchanged and propagated by suitably piggy-backing on the RM and NM heartbeats. The RM could also be changed adaptively to satisfy the AM transparent failover functionalities.



(a) Instance number per job (b) Instance running time per job

Fig. 6. Workload profiling from one of Alibaba production systems

5 System Evaluation

5.1 Evaluation Setup

To determine the effectiveness of our proposed failover technique described in Section 4, it is necessary to evaluate its impact on task execution performance and resource overhead in the presence of failures. To investigate this, we conducted a number of experiments submitting jobs onto

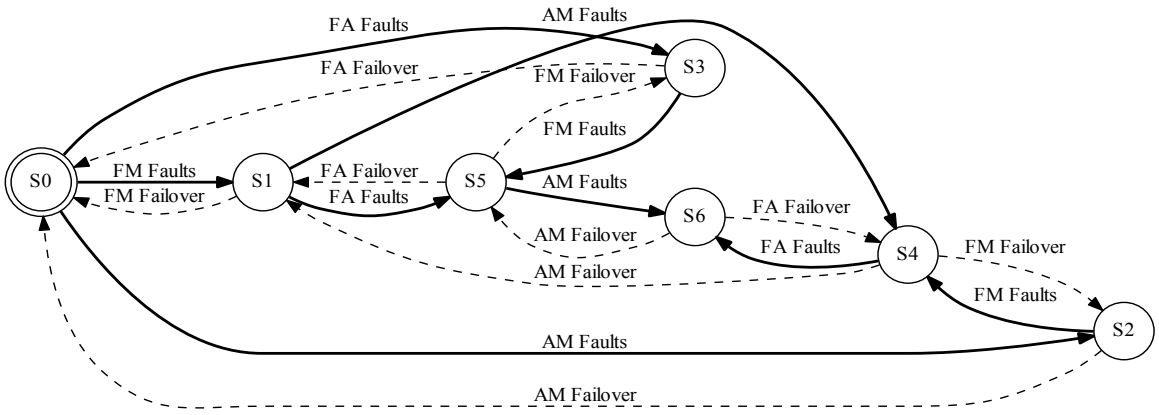


Fig. 5. Fuxi-FT Finite-state machine (FSM): Nodes represent simplified system failure states. S0 is the normal system state and S6 describes a system state where multiple and simultaneous failures occur. Each edge represents a failure event in solid line or a failover process in dashed line.

a large-scale cluster under a number of different failure scenarios and deployed failover techniques.

Before presenting experiments, we conduct workload profiling from a production system which consists of over 2,400 machines. According to Figure 6, roughly 20%, 45% and 35% of all jobs contains 10, 100 and 1,000 instances, respectively. In addition, the average running time is approximately 50 seconds. Thus, we submit these small, medium and large jobs according to the above proportion with each instance executing for 50 seconds and requesting 0.5 CPU cores and 1 GB memory. 4,000 jobs were submitted onto a cluster comprising 300 nodes, while keeping the number of active jobs approximately 2,000 (i.e. Once a job has completed execution, an additional job is submitted onto a node). Each physical node contains 82GB memory, and 23 cores for CPU, and each AppMaster process consumes 100MB memory. Additionally, the types of fault-injection selected for the experiments are crash faults of software and interrupt faults (transient time-out). Each of these experiments are separated into two categories for single component and multiple component failure scenarios. For the former, for every fault injection, either the Fuxi Master, or 5% of the total AppMasters, or 5% of FAgents will be stochastically selected to fail. For the latter, the Fuxi Master and 5% of other components(AppMasters or FAgents) will experience simultaneous failure. In reality, as studied in [17], failures occur much less frequently than specified injection interval, and also affect less component that the specified component failure proportion. Each experiment case also included a different deployment for failover for both Fuxi Master(FM) and AppMaster(AM) as shown in Table 4. They were divided into no failover technique deployed (NoFO), FM failover with no AM failover (FMFO), AM failover with no FM failover (AMFO) and finally our approach which allows all components to failover (Fuxi–FT).

Faults were injected every 300s by sending a light-weight command to terminate/interrupt the execution of specific components. Each experiment observation period is defined as the completion of all submitted jobs into the cluster, and the following metrics of interest were collected for study: (1) Job Performance: impact on job execution, job end to end span-time, instance eviction etc; (2) Component Recovery Time: time required to perform failover; (3) Cluster Resource Utilization: entire cluster utilization and scheduling performance.

TABLE 4
Experiment cases for failover effectiveness

No Faults (Baseline)	Our approach(Fuxi-FT)
Fault Injection (single component & concurrent multiple component faults)	Our approach(Fuxi-FT)
	FM failover; no AM failover (FMFO)
	no FM failover, AM failover (AMFO)
	no FM failover; no AM failover (NoFO)

5.2 Fuxi-FT Component Failover Evaluation

5.2.1 Job Execution Results

Table 5 shows the number of jobs submitted and the total number of unique instances for each experiment. It is observable that all jobs are able to successfully complete and that 2,115 and 2,101 instances (approximately 0.13% of all tasks) are re-scheduled under single component and simultaneous failure injection, respectively. Furthermore, we can observe a minimal increment in the instance execution overhead, increasing by 7.79 microseconds (6.4%) on average between the baseline and the single fault-injection job execution. In contrast, the overhead of instance execution for simultaneous fault-injection is substantially larger, increasing by 228.5 microseconds (200.5%).

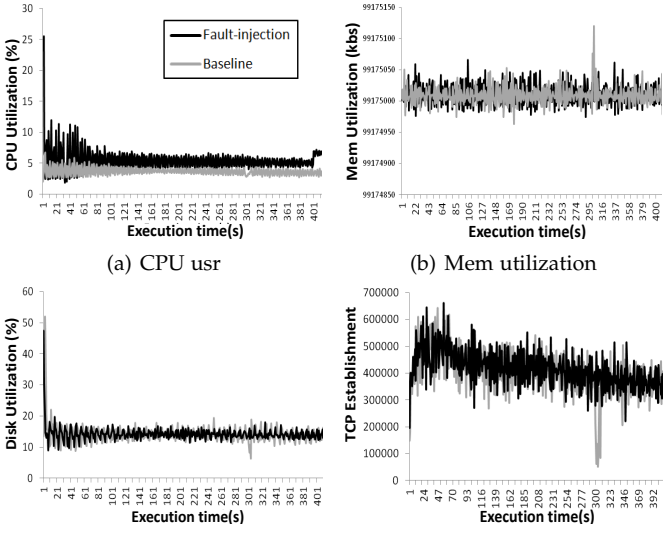
The overhead within the baseline experiment of 113.97 microseconds is due to process initialization and network communication establishment with FAgent and corresponding AppMasters, while the enlarged overhead in the fault-injection experiments is due to the failed components requiring to perform state recovery, occupying more handling and scheduling time for routine instance management. In addition, updating instance status can be delayed due to the extra CPU cycles that are required to finish the holistic computation. The phenomenon is aggravated when simultaneous faults are injected. Despite this, it is worth noting that this additional overhead is substantially smaller compared to an instance's execution time.

5.2.2 Cluster Utilization

We have also studied the system overhead in terms of resource utilization at the cluster level as shown in Figure 7 and Table 6. We observe that there is a lack of statistical significance between resource utilization of memory, disk

TABLE 5
Statistical properties of job and task execution for Fuxi-FT under different failure scenarios.

	No Faults (baseline)				Fuxi-FT with single component fault injection				Fuxi-FT with simultaneous multiple component fault injection			
	Job Category				Job Category				Job Category			
	Small	Medium	Large	Total	Small	Medium	Large	Total	Small	Medium	Large	Total
Submitted	807	1789	1404	4000	812	1806	1382	4000	803	1807	1390	4000
Completed	807	1789	1404	4000	812	1806	1382	4000	803	1807	1390	4000
Failed	0	0	0	0	0	0	0	0	0	0	0	0
	Instance Category				Instance Category				Instance Category			
Submitted	8070	178900	1404000	1590970	8120	180600	1382000	1570720	8030	180700	1390000	1578730
Completed	8070	178900	1404000	1590970	8120	180600	1382000	1570720	8030	180700	1390000	1578730
Rescheduled	0	0	0	0	10	258	1847	2115	17	148	1936	2101
Resched Ratio	0	0	0	0	0.12%	0.14%	0.13%	0.13%	0.21%	0.08%	0.14%	0.13%
Overhead (us)	119.91	112.96	113.42	113.97	121.53	121.72	121.58	121.76	341.49	341.93	341.44	342.37



(a) CPU usr (b) Mem utilization

(c) Disk utilization (d) Network utilization

Fig. 7. Average cluster resource utilization of no fault and fault-injection with Fuxi-FT. (a) CPU utilization, (b) memory utilization, (c) disk utilization, (d) network utilization.

utilization and TCP Establishment between the two experiments demonstrated statistically at less than 1% difference, and presented visually in Figure 7(b-d). This is because when component failover commences, the transmitted soft-state amount to required component is approximately equivalent to routine message communication amongst distributed components. In fact, these components send heartbeats to each other, and synchronize the full knowledge and information periodically in typical operation. The periodic synchronization is the fundamental safeguard for the proposed failover mechanism. Moreover, we observe that in the presence of faults, the cluster utilization of CPU increases by 1.71% representing an average disparity of 46.56% between the baseline and fault-injection experiments. This is because the failover mechanisms are required to perform computation to collect and transmit states in order to perform state inference recovery. We also observe from Table 6 that the request handling performance is not negatively impacted by our proposed approach and the average scheduling performance can be maintained during the soft-state inference. This is a significant improvement compared to the pure hard-state recovery we discussed in Section 2.

TABLE 6
Statistical properties of cluster resource utilization and scheduling performance to handle resource requests.

	Baseline		Fault-Injection		Difference
Resource	Avg	St. Dev	Avg	St. Dev	Avg Diff
CPU(%)	3.67	0.59	5.38	2.15	46.56%
Mem(KB)	99168	15.18	99179	16.70	0.01%
Disk(%)	14.07	3.03	14.20	2.30	0.96%
TCP est(times)	401747	83453.04	411846	70529.68	2.51%
ReqHandle(ms)	0.884	0.56	0.886	0.571	0.23%

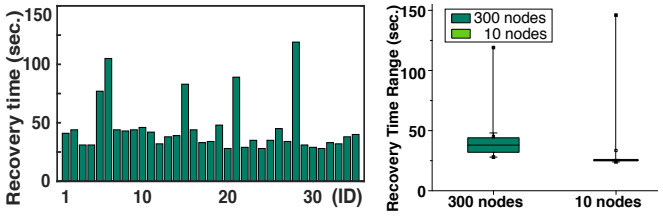
5.2.3 Component Recovery Time

As illustrated in Table 7, Fuxi components exhibit different recovery times. In single fault injection experiments, Fuxi Master and FAgent recover on average within 66.9 seconds and 5.6 seconds, respectively, as well as exhibit stable recovery times indicated by a Coefficient of Variation(CoV) smaller than 0.1. FAgent recovery time is relatively short, as it requires minimal soft-state collection from other components within the cluster as described in Section 4.1, and upon recovery will collect monitoring data from the physical node it is executing. On the other hand, the recovery time of Fuxi Master is considerably longer, which is due to its pre-configuration within the cluster. In our experiment, Fuxi Master is configured with a waiting period of 60 seconds, with best efforts for all FAgents and AppMasters to report their status and collated data, indicating that the active Fuxi Master is capable of failover within 7 seconds. This result indicates that the recovery time could be reduced considerably if alterations are made to the waiting time for information aggregation phase, and the detailed customized configurations are disucussed in Section 3.3. Furthermore, we also observe that simultaneous fault injection scenarios share a similar recovery time with single fault behavior, at 68.81 seconds and 5.71 seconds on average for Fuxi Master and FAgent respectively. It demonstrates that our approach will not incur extra overhead while providing transparent fault handling in terms of component recovery time. Compared to the pure hard-state recovery, soft state inference does take time. Nevertheless, the recovery could still complete within seconds-level with the same failover effects, thereby being soundly accepted.

AppMaster recovery time exhibits the most diversity. For example, Figure 8(a) depicts the average recovery time just under 46.2 seconds with a standard deviation of 27.9 seconds in single component injection scenarios. The reason for this diversity in recovery time is due to the distribution of workers across physical nodes and the incurred

TABLE 7
Component recovery time under different failure scenarios and the comparison between Fuxi-FT and Yarn

Fuxi Component	Single Fault		Concurrent Faults	
	avg(s)	stddev(s)	avg(s)	stddev(s)
Fuxi Master	66.94	1.08	68.81	2.4
FAgent	5.57	0.50	5.73	0.48
AppMaster	46.63	27.90	43.43	17.6
Yarn Component	**Single Fault**		**RM+NM Concurrent Faults**	
	avg(s)	stddev(s)	avg(s)	stddev(s)
RM	57.13	35.32	61.78	39.67
NM	5.59	0.43	5.88	0.77
AppMaster	×	×	×	×



(a) AppMaster recovery time (b) Recovery comparison

Fig. 8. (a) AppMaster recovery time under single component fault injection experiments; (b) Recovery time comparison using box-plot with 25th,75th percentile, min, max

additional complexity in synchronizing those worker data. This behavior is exemplified when comparing AppMaster recovery time within a smaller cluster consisting of 10 nodes and 400 jobs as shown in Figure 8(b), resulting in an average recovery time of 33.4 seconds (19.7% increase) under fault-injection. Such a result indicates that the locality of workers within physical nodes contributes to the extra recovery time.

5.3 Recovery Effects Using Yarn

To make a more comprehensive comparison, we conduct similar experiments using Yarn with same cluster and same job submission configurations. The experimental results show that all jobs will fail if AppMaster and other components fail at the same time. In fact, Yarn does not support the case that AppMaster failovers during ResourceManager(RM) and NodeManager(NM) fail simultaneously. The recovery effects are shown in Table 7. The NM recovery time in single fault injection is 5.59 seconds on average and will have a slight increase to 5.88 seconds if RM fails simultaneously, reflecting similar recovery times compared to FAgent. It is worth mentioning that we find drastic fluctuation in recovery performance for RM between 21.4 seconds to 91.6 seconds. This huge variation is attributed to not using a configurable wait-time to collect states at the begining of RM recovery. In some scenarios where an active AM states fail to re-sync with RM, RM has to wait for a relatively long time, resulting in a very slow component recovery. This design is also strongly related to the submitted job number. The more jobs submitted, the higher probability of recovery extension will be. In fact, we regard the Yarn's strategy as a more common mechanism that does not support AM failover, and will have a more detailed comparisons in next subsection.

5.4 Comparison among Different Failover Mechanisms

In this section, we inject simultaneous faults under four system failover techniques described in Table 4 in order to study and compare job executions, end-to-end job completion time (JCT) under simultaneous fault injection scenario.

As shown in Table 8, Fuxi-FT can guarantee completion of all jobs under both single and simultaneous fault injections. During job executions, all task instances can obtain the requested resource and finally finish computation despite a small proportion of re-scheduled instances. Moreover, there is merely a negligible change of rescheduled instances between the single and multiple injection experiments. This is because in our approach, Fuxi Master attempts to retain already assigned resources by preserving temporarily resources from failed components. A number of AppMasters with a pre-defined upper-bound for timeout will hold respective resources without evicting any running workers. Beyond the threshold, AppMasters will reclaim the resources to Fuxi Master, and then request for new resources re-scheduling when facing with timing failures mentioned in Section 4. Such results demonstrate that all jobs will complete execution using our proposed approach in Fuxi-FT. In reality, faults which cause 5% of components to fail occur much more infrequently than every 300 seconds as observed in [17], as specified in the fault-injection experiment. The conducted experiments demonstrate that even under aggressive fault-injection, it is still possible to provision reliable services to consumers.

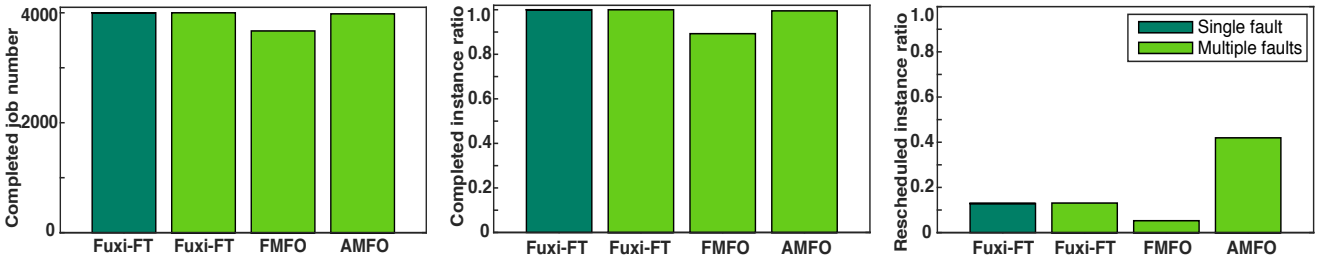
Additionally, it is also observable that if AppMaster is unable to automatically failover, a large number of running jobs fail, causing a significant amount of computation waste and extension to job completion time holistically within the system. In fact, 8.33% jobs eventually fail while only 92.16% task instances finish as plan. Although the rescheduled instance ratio appears to be reduced due to many direct interruptions after job failure, the large proportion of killed job is a severe threat to reliable services provisioning.

We also conduct experiments to switch off the capability for the central scheduler (Fuxi Master) for fault recovery but allow AppMasters to failover independently. In this scenario, the crash-stop failure on Fuxi Master will lead to eviction of all running non-faulty workers. Consequently, all running task instances have to be resubmitted and wait for rescheduling. The AppMaster will also be rescheduled to another location after the Fuxi Master revives. As illustrated from Table 8, the rescheduled instance number are approximately 3x greater times than Fuxi-FT. Indeed, over 4,000 extra rescheduled instances are just incurred by the absence of Fuxi Master effective failover. All comparable results are demonstrated in Figure 9 in terms of completed job number, instance completion ratio and the corresponding rescheduled instance proportion discussed above.

The job completion time is shown in Figure 10. The end-to-end time starts from the launch of the AppMaster and consists of waiting time for the requested resource being satisfied and the completion of parallel task executions. Due to the total requested resource from all jobs surpass the overall cluster capacity, many jobs have to remain within the waiting queue. From the CDF in Figure 10, roughly 40% job completion times are no greater than 300 seconds and different scenarios share a very similar phenomenon despite some marginal discrepancies. This is due to concurrent submitted jobs can obtain the required resource in this time period according to the adopted scheduling policies (e.g.,

TABLE 8
Statistical properties of job and task execution within failover approaches for different failure scenarios

	Single Component Fault	Simultaneous Component Fault			
	Fuxi-FT	Fuxi-FT	FMFO	AMFO	NoFO
	Job Category				
Submitted	4000	4000	4000	4000	4000
Completed	4000	4000	3667	3978	0
Failed(%)	0	0	333(8.33%)	22(0.55%)	4000(100%)
	Instance Category				
Submitted	1570720	1578730	1575400	1589170	1572450
Completed(in submitted %)	1570720(100%)	1578730(100%)	1451830(92.16%)	1580490(99.45%)	0
Rescheduled(in Completed%)	2115(0.135%)	2121(0.134%)	757(0.052%)	6678(0.423%)	1572450(100%)



(a) Completed job number (b) Completed instance ratio (c) Rescheduled instance ratio (unit: %)

Fig. 9. Under the multiple concurrent component fault injections, (a) completed job number comparison (b) completed instance ratio within the submitted instances. (c) rescheduled instance ratio within the completed instances.

fair scheduling). Other jobs have to wait for resources until the resource is reclaimed from running tasks.

In general, the job execution time in AMFO is greater than FMFO and the proposed Fuxi-FT as a large number of task evictions and re-schedulings occur after the Fuxi Master crash-stop failure. All these computations have to be restarted from the very beginning, resulting in serious execution delays. As for jobs in FMFO, the completion time distribution of approximately 80% jobs almost coincides with Fuxi-FT jobs. However, waiting jobs could obtain the resource much faster than Fuxi-FT jobs. The reason for this observation is due to the large amount of resource revoked from the failed jobs and these resources could be instantly utilized by the waiting instances, thereby shortening the holistic completion time.

Without autonomous failure recovery techniques, no jobs and its tasks can tolerate and recover from component failure. As shown in Table 8, any running worker will be killed upon any master failure. Even more debilitating, the worker which runs the master will also be killed, resulting in all instance eventual failures. In general, these catastrophes are very atypical as the adopted fault injection scenario is so harsh that few probabilities exist in real-life systems. However, increasingly complicated scenarios and urgent problems in Internet-scale systems substantially inspire both engineering and academic aspects to impel the rapid improvement of fault-tolerance techniques.

6 Related Work

Fault-tolerance in resource management has been studied for many years within different distributed systems and could be coarsely divided into categories: forward-based recovery and backward-based recovery [16], both of which adopt the redundancy to mask residual design faults of software programs. Forward recovery produces correct results through continuation of normal processing and a classical approach is the N-version programming scheme (NVP) [25]

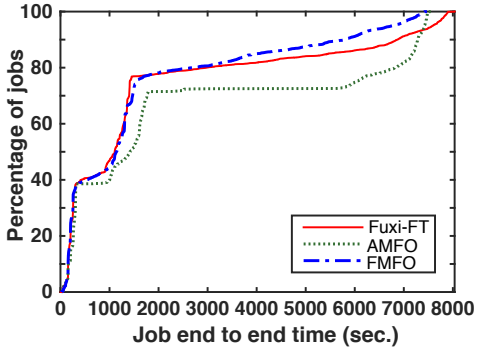


Fig. 10. CDF of job completion time under different failover techniques.

which is actually a n-modular redundancy. However, the main limitation is that it is highly application dependent. In comparison, backward recovery make the process restarted according to the last saved states upon failures. In fact, in order to achieve this, checkpointing involves occasionally saving the state of a process in stable storage during normal execution. This can thus reduce the amount of lost work. Checkpointing and recovery is mainly intended to tolerate transient hardware failures, where the application is restarted upon repair of a hardware unit after failure and is also used as a backward error recovery technique for handling intermittent software faults.

More specifically, these dynamic redundancy approaches such as Recovery Blocks (RB) [10], checkpointing [32], snapshot had been widely-used in the failure recovery and high availability assurance scenarios in HPC [33], web services [34] and Grid [35] and VM-based virtual computing environments [36]. However, as identified in [37], the requirements of scale, heterogeneous workload characteristics and fault-tolerance are substantially different. For example, the periodical multi-level checkpointing and rolling-back techniques [38] are suitable for long-running MPI tasks but cannot be properly applied in short tasks or time-sensitive

tasks. In fact, the resource requests and allocations of these tasks are determined in advance and will not change during its life-cycle. The number of tasks is also not large compared with available resources, making sufficient resources to conduct redundant checkpointing. VM-based snapshot [36] [39] [40] focuses on the de-duplication and recovery for the large amount of memory and disk states in runtime. In general, above systems achieve effective resource scheduling and management by large backlogs of pending work – an assumption which cannot be adhere to the on-demand access required for Cloud computing. Considering the large cost for millions of running tasks, it is infeasible to conduct them in Internet-scale systems.

Furthermore, we mainly focus on the most relevant approaches within the context of large-scale distributed systems. There have been a number of studies that analyze Cloud datacenters tracelogs [41] [42] [43] [44] exemplifying the degree of inherit heterogeneity of workload and server architectures within the system environment. Furthermore, [17] demonstrates the frequency and manifestation of failures, while [45] studies task re-computation waste per server and [31] quantifies produced energy waste.

YARN [4] adopts a request-based resource allocation scheme. Similar to Fuxi, Yarn has the similar component: Resource Manager(RM), Application Master(AM) and Node Manager(NM). RM only recently becomes fault-tolerant since Hadoop 2.6 [14] [15]. In particular, the works could be preserved when the RM and NM restarts. RM repopulates running states by taking advantage of the container statuses sent from all NMs and pending requests resent from AMs. As for the NM, the NM currently stores all necessary states to a local state-store as it processes container-management requests. When the NM restarts, it recovers by loading state for various subsystems. However, even though the continuation of containers without restart, the current protocol in RM does not support AM failover. In fact, although the NM reports real-time failure statuses of AMs to RM, RM revokes the pertaining resources without reserving the allocated resources to affected AMs. Moreover, containers on timing-out NM which comes in late during the RM restart will also be killed. Due to these limitations, only RM and NM crash simultaneous failure could be covered by Yarn. In comparison, this paper systematically describes comprehensive solutions and achieve a full coverage of simultaneous failure scenarios of all components, and they could be applied into any massive-scale resource management system.

Mesos [3] provides an offer-based mechanism in which the primary master is responsible for assigning resources to each computing framework. Of relevance to this work, while the mechanism deploys a warm-standby failover mechanism, the architecture does not include fault-tolerant schemes that can be applied to applications running within system nodes, leaving such responsibility to individual consumers. Our approach provides a full-components failover solution for all sorts of fault combinations.

Borg [37] is the cluster management system within Google. The central Borgmaster gathers resource-availability information from nodes, accepts applications resource requests, and matches one to the other. Borg uses techniques such as replication and persistent state storing to deal with faults including task preemption and reallocation, machine failure or shutdown etc. In order to improve the availability, Borg primarily focuses on how to cope with machine-level hardware breakdowns. However, the information on failure treatment is extremely limited, with only brief examples of practical experience described. Additionally, the scarcity of any experimental evaluation make it difficult for researcher to analyze and compare with their methods. Borg might take occasional (full state) checkpoints when appropriate. By contrast, we present an approach for rapidly low-cost failover, in which light-weighted checkpoints ensure much lower overheads of hard state storing (than full state saving) and soft-state inference facilitates the state re-construction. We also describe how to avoid running workers suffering from simultaneous faults combinations with a comprehensive faults coverage. In terms of recovery efficiency, Borg briefly states the failing-over takes about 10s, but can takes up to a minute in a big cluster. In our paper, our approach is able to complete a recovery (using re-constructed full state) within seconds but less than a minute. Fuxi-FT can also perform much quicker failover (using incomplete state) when a degraded service level is acceptable.

Sparrow [46] adopts randomized-sampling based decentralized schedulers and aims to process short jobs. Frameworks that use the failed scheduler have to detect the failure and connect to a backup scheduler themselves. Due to no state persisted to disks, the in-progress jobs have to be restarted. Apollo [13] uses distributed schedulers to achieve scheduling scalability. However, it marks the job failed and all running tasks in the queue will be evicted once the node agent failovers. All these motivates us to design and implement full component failures recovery approaches.

7 Conclusions

In this paper, we have presented a novel approach for implementing fault tolerance in large-scale Cloud datacenters by means of rapid user-transparent failover. Effective recovery of system components is achieved through a combination of hard state backup and soft-state inference, thereby reducing the recovery overhead significantly. Our design and architecture also take into account the scale and complexity of a Cloud datacenter and cope with different types of failure and failover scenarios. The method has been deployed and validated by Alibaba in their live production environment under different failure scenarios. The main conclusions from our research are summarized as follows:

State-based failure recovery is typically expensive and costly in large-scale distributed systems. We have demonstrated that the combined use of hard state and soft-state inference is in fact an effective and efficient means to achieve failover in such systems. The recovery overhead to instance execution time in our system is limited within 7 microseconds and 228.5 microseconds in the presence of single and simultaneous component failures respectively, with a minimal increment to system CPU utilization.

Large-scale distributed systems such as Cloud datacenters may run millions of instances concurrently, with an increased probability of frequent and simultaneous failures. These failures have to be understood properly and addressed appropriately together with a correct scheduling strategy for instances.

Inappropriate scheduling of instances has the potential to dramatically affect the whole system reliability due to the complex co-relation between rescheduling and communication caused by application failures. Our technique has also attempted to tolerate timing failures, an increasingly dominating failure type for modern service applications.

Future research directions of this work include further investigation of the patterns of AppMaster failover at different system scales, and in diverse operational environments. Reducing further recovery time would be another interesting challenge with more adaptive customization of time for soft-state collection. Furthermore, apart from node crashes, other dominating types of failure need to be re-examined, in particular those contributing to the long tail straggler [47] [48] identified from production workload execution analysis.

ACKNOWLEDGMENTS

Special thanks must go to the overall Fuxi distributed resource scheduling team in Alibaba Cloud Inc. and the SIGRS group from Beihang University for their supports and collaborative contributions. We would also like to extend our sincere thanks to the anonymous reviewers for their valuable comments and help in improving this paper. This work is supported by China 973 Program (No. 2011CB302602), China 863 Program (No. 2015AA01A202), UK EPSRC WRG Platform Project (No. EP/F0577644/1), HGJ Program (No. 2013ZX01039-002-001-001), and NSFC Program (No. 61202424, 91118008, and 61170294).

REFERENCES

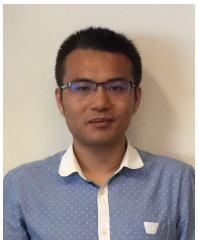
[1] R. Buyya, R. Ranjan, and R. N. Calheiros, "Intercloud: Utility-oriented federation of cloud computing environments for scaling of application services," in *Algorithms and architectures for parallel processing*. Springer, 2010.
[2] C. G. C. Index, "Forecast and methodology, 2012–2017, white paper," 2013.
[3] 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 *USENIX NSDI*, 2011.
[4] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *ACM SoCC*, 2013.
[5] 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 *VLDB*, 2014.
[6] L. A. Barroso, J. Clidaras, and U. Hölzle, "The datacenter as a computer: An introduction to the design of warehouse-scale machines." Morgan & Claypool Publishers, 2013.
[7] J. Li, M. Humphrey, Y.-W. Cheah, Y. Ryu, D. Agarwal, K. Jackson, and C. van Ingen, "Fault tolerance and scaling in e-science cloud applications: Observations from the continuing development of modisazure," in *IEEE e-Science*, 2010.
[8] R. Yang and J. Xu, "Computing at massive scale: Scalability and dependability challenges," in *IEEE SOSE*, 2016.
[9] (2008) Amazon suffers u.s. outage on friday internet. [Online]. Available: http://news.cnet.com/
[10] B. Randell and J. Xu, "The evolution of the recovery block concept," *Software Fault Tolerance*, 1995.
[11] P. Jalote and P. Jalote, *Fault tolerance in distributed systems*. PTR Prentice Hall Englewood Cliffs, 1994.
[12] N. Aghdaie and Y. Tamir, "Fast transparent failover for reliable web service," in *IEEE PDCS*, 2003.
[13] 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 *USENIX OSDI*, 2014.
[14] https://issues.apache.org/jira/browse/YARN-556.
[15] https://issues.apache.org/jira/browse/YARN-1336.
[16] A. Avižienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," in *IEEE TDSC*, 2004.
[17] P. Garraghan, P. Townend, and J. Xu, "An empirical failure-analysis of a large-scale cloud computing environment," in *IEEE HASE*, 2014.
[18] 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 *ACM SoCC*, 2012.
[19] Q. Wei, B. Veeravalli, B. Gong, L. Zeng, and D. Feng, "Cdrm: A cost-effective dynamic replication management scheme for cloud storage cluster," in *IEEE CLUSTER*, 2010.
[20] S. Malik and F. Huet, "Adaptive fault tolerance in real time cloud computing," in *IEEE SERVICES*, 2011.
[21] R. Yang, T. Wo, C. Hu, J. Xu, and M. Zhang, "D2ps: a dependable data provisioning service in multi-tenants cloud environments," in *IEEE HASE*, 2016.
[22] Q. Zheng, "Improving mapreduce fault tolerance in the cloud," in *IEEE IPDPS*, 2010.
[23] S. Fu, "Failure-aware resource management for high-availability computing clusters with distributed virtual machines," 2010.
[24] (2013) Amazon web services suffers outage. [Online]. Available: http://www.zdnet.com/article/amazon-web-services-suffers-outage-takes-down-vine-instagram-others-with-it/
[25] A. Avizienis, "The methodology of n-version programming," *Software fault tolerance*, 1995.
[26] M. R. Lyu *et al.*, *Handbook of software reliability engineering*, 1996.
[27] F. Longo, R. Ghosh, V. K. Naik, and K. S. Trivedi, "A scalable availability model for infrastructure-as-a-service cloud," in *IEEE DSN*, 2011.
[28] R. Subramanian and V. Anantharaman, "Reliability analysis of a complex standby redundant systems," in *Elsevier Reliability Engineering & System Safety*, 1995.
[29] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems." in *USENIX ATC*, 2010.
[30] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *USENIX OSDI*, 2006.
[31] P. Garraghan, I. S. Moreno, P. Townend, and J. Xu, "An analysis of failure-related energy waste in a large-scale cloud environment," in *IEEE TETC*, 2014.
[32] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," *IEEE TSE*, 1987.
[33] G. Staples, "Torque resource manager," in *ACM SC*, 2006.
[34] M. P. Papazoglou, "Service-oriented computing: Concepts, characteristics and directions," in *IEEE WISE*, 2003.
[35] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke, "Grid services for distributed system integration," in *IEEE Computer*, 2002.
[36] L. Cui, J. Li, T. Wo, B. Li, R. Yang, Y. Cao, and J. Huai, "Hotrestore: a fast restore system for virtual machine cluster," in *USENIX LISA*, 2014.
[37] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *ACM EuroSys*, 2015.
[38] A. Moody, G. Bronevetsky, K. Mohror, and B. R. De Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *IEEE SC*, 2010.
[39] Y. Huang, R. Yang, L. Cui, T. Wo, C. Hu, and B. Li, "Vmcsnap: Taking snapshots of virtual machine cluster with memory deduplication," in *IEEE SOSE*, 2014, pp. 314–319.
[40] J. Li, J. Zheng, L. Cui, and R. Yang, "Consnap: Taking continuous snapshots for running state protection of virtual machines," in *IEEE ICPADS*, 2014, pp. 677–684.
[41] Y. Chen, S. Alspaugh, and R. H. Katz, "Design insights for mapreduce from diverse production workloads," Tech. Rep., 2012.
[42] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan, "An analysis of traces from a production mapreduce cluster," in *CCGrid 2010*. IEEE.
[43] A. K. Mishra, J. L. Hellerstein, W. Cirne, and C. R. Das, "Towards characterizing cloud backend workloads: insights from google compute clusters," in *ACM SIGMETRICS*, 2010.
[44] I. Solis Moreno, P. Garraghan, P. Townend, and J. Xu, "Analysis, modeling and simulation of workload patterns in a large-scale utility cloud," in *IEEE TCC*, 2014.

[45] P. Garraghan, P. Townend, and J. Xu, "An analysis of the server characteristics and resource utilization in google cloud," in *IEEE IC2E*, 2013.
[46] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: distributed, low latency scheduling," in *ACM SOSP*, 2013.
[47] J. Dean and L. A. Barroso, "The tail at scale," in *ACM Communication*, 2013.
[48] P. Garraghan, X. Ouyang, P. Townend, and J. Xu, "Timely long tail identification through agent based monitoring and analytics," *IEEE ISORC*, 2015.

Renyu Yang is a Ph.D. candidate at Beihang University, and a Research and Development intern at Alibaba Cloud Computing Inc. He received his B.Sc degree from Beihang University in 2011, and was a visiting researcher at University of Leeds, UK in 2012 and 2013 respectively. His research interests include resource management in massive-scale distributed systems, Cloud computing, system dependability etc.

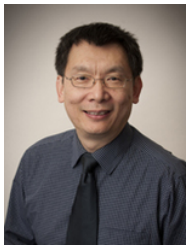
Yang Zhang is currently a software engineer in Alibaba Cloud Computing Inc. He received his M.Sc and B.Sc degree both from Beijing University of Posts and Telecommunications(BUPT) in 2014 and 2011 respectively. His research interest is large-scale distributed systems.

Peter Garraghan is a Research Fellow in the School of Computing, University of Leeds and a visiting researcher at Beihang University, China. He has industrial experience building large-scale systems and his research interests include distributed systems, large-scale simulation, dependability, data analytics and energy-efficient Cloud datacenters.



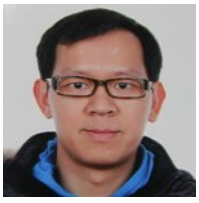
Yihui Feng is currently a senior expert in Alibaba Cloud Computing Inc and he received his M.Sc degree from school of computing in Beihang University in 2007. His research interest is large-scale distributed systems.

Jin Ouyang is currently a senior software engineer in Alibaba Cloud Computing Inc. He received his M.Sc degree from University of Science and Technology of China(USTC) in 2014 and his research interests are large scale distributed systems, resource scheduling techniques and performance optimization.



Jie Xu is Chair Professor of Computing at University of Leeds and Director of UK EPSRC WRG e-Science Centre. He has industrial experience in building large-scale networked systems and has worked in the field of dependable distributed computing for over 30 years. He is a Steering/Executive Committee member for numerous IEEE conferences including SRDS, ISORC, HASE, SOSE and is a co-founder for IEEE IC2E. He has led or co-led many research projects to the value of over $30M, and published in excess of 300 academic papers, book chapters and edited books.

Zhuo Zhang is a Senior Staff Manager at Alibaba Cloud Computing Inc. and leads the teams of Cloud resource management and job scheduling. He previously worked at IBM China Research & Development Lab in DB2 performance monitor/optimization. His current research interests include distributed systems, Cloud computing etc.



Chao Li is a Senior Staff Manager at Alibaba Cloud Computing Inc. and leads the teams of Cloud resource management and job scheduling. His research interest is distributed systems and large scale data processing techniques.