Fuxi: a Fault-Tolerant Resource Management and Job Scheduling System at Internet Scale

Zhuo Zhang*, Chao Li*, Yangyu Tao*, Renyu Yang†*, Hong Tang*, Jie Xu†§
Alibaba Cloud Computing Inc.* Beihang University† University of Leeds§
{zhuo.zhang, li.chao, yangyu.taoyy, hongtang}@alibaba-inc.com
yangry@act.buaa.edu.cn j.xu@leeds.ac.uk

ABSTRACT

Scalability and fault-tolerance are two fundamental challenges for all distributed computing at Internet scale. Despite many recent advances from both academia and industry, these two problems are still far from settled. In this paper, we present Fuxi, a resource management and job scheduling system that is capable of handling the kind of workload at Alibaba where hundreds of terabytes of data are generated and analyzed everyday to help optimize the company's business operations and user experiences. We employ several novel techniques to enable Fuxi to perform efficient scheduling of hundreds of thousands of concurrent tasks over large clusters with thousands of nodes: 1) an incremental resource management protocol that supports multi-dimensional resource allocation and data locality; 2) user-transparent failure recovery where failures of any Fuxi components will not impact the execution of user jobs; and 3) an effective detection mechanism and a multi-level blacklisting scheme that prevents them from affecting job execution. Our evaluation results demonstrate that 95% and 91% scheduled CPU/memory utilization can be fulfilled under synthetic workloads, and Fuxi is capable of achieving 2.36TB/minute throughput in GraySort. Additionally, the same Fuxi job only experiences approximately 16% slowdown under a 5% fault-injection rate. The slowdown only grows to 20% when we double the fault-injection rate to 10%. Fuxi has been deployed in our production environment since 2009, and it now manages hundreds of thousands of server nodes.

1. INTRODUCTION

We are now officially living in the *Big Data* era. According to a study by Harvard Business Review in 2012, 2.5 exabytes of data are generated everyday and the speed of data generation doubles every 40 months [13]. To keep up with the pace of data generation, data processing has also been progressively migrating from traditional database-based approaches to distributed systems that scale out much easily. In recent years, many systems have been proposed from both academia and industry to support distributed data processing with commodity server clusters, such as Mesos [11], Yarn [18] and Omega [16]. However, two major challenges remain unsettled in these systems when faced with the challenges of managing resources for systems at Internet scale:

1) ***Scalability***: Resource scheduling can be simply considered as the process of matching demand (requests to allocate resources to run processes) with supply (available resources of cluster nodes). So the complexity of resource management is directly affected by the number of concurrent tasks and the number of server nodes in a cluster. Furthermore, other factors also impact the complexity, including supporting resource allocation over multiple dimensions (such as CPU, memory, and local storage), fairness and quota constraints across competing applications; and scheduling tasks close to data. A naive approach of delegating everything decision to a single master node (as in Hadoop 1.0) would be severely limited by the capability of the master. On the other hand, a fully-decentralized solution would be hard to design to satisfy scheduling constraints that depends on fast-changing global states without high synchronization cost (such as quota management). Both Mesos and Yarn attempt to deal with these issues in slightly different ways. Mesos adopts a multiple-level resource offering framework. However, Mesos master offers free resources in turn among frameworks, the waiting time for each framework to acquire desired resources highly depends upon the resource offering order and other frameworks' scheduling efficiency. Yarn's architecture decouples resource management and programming paradigms. However, the decoupling is only for the separation of code logic between resource management and MapReduce job execution, so that other programming paradigms can be accommodated in Yarn. Nevertheless, it does not reduce the complexity of resource management, and still inherits the linear resource model as in Hadoop 1.0.

In both cases, states are exchanged with periodic messages and the interval configuration is another intricate challenge. A long period could reduce communication overhead but would also hurt utilization when applications wait for resource assignment. On the other hand, frequent adjustments would improve response to demand/supply changes (and thus improve resource utilization); however, it would also aggravate the message flooding phenomenon. In fact, Yahoo! reported that they did not run Hadoop clusters bigger than 4,000 nodes [18] and Yarn had yet to demonstrate its scale limit.

2) ***Fault-tolerance***: With increasing scale of a cluster,

Proceedings of the VLDB Endowment, Vol. 7, No. 13

the probability of hardware failures also arises. Additionally, rare-case software bugs or hardware deficits that never show up in a test environment could also suddenly surface in production. Essentially, failures become the norm rather than the exception at large scale [15]. The variety of failures include halt failures due to OS crash, network disconnection, and disk hang, etc [9]. Traditional mechanisms like health monitoring tools or heartbeat can help but cannot completely shield the failures from applications. Another challenge is how to cope with master failures. In Yarn, when the resource manager fails and then recovers, it has no recall of the cluster states. Thus all running applications (including application masters) will start over. Similarly, the failure of a node manager could also result in the re-execution of all running tasks on the node, even if it soon recovers.

At Alibaba, hundreds of millions of customers visit our Web sites everyday, looking for things to buy from over one billion items offered by our merchants. Hundreds of terabytes of user behavior, transaction, and payment data are logged and must go through elaborated processing everyday to support the optimization of our core business operations, such as online marketing, product search and fraud detection; and to improve user experiences such as personalization and product recommendation. In this paper, we present Fuxi[1], the resource management and job scheduling system that supports our proprietary data platform (called Open Data Processing Service or ODPS [6]) to handle the Internet-scale workload at Alibaba.

Fuxi's overall architecture bears some resemblance to Yarn. In this paper, we mainly focus on the the following three aspects that are key to scaling Fuxi to thousands of nodes and hundreds of thousands of concurrent processes, maintaining high resource utilization, and shielding low-level failures from impacting applications. We believe these techniques are in the right direction to solving the scalability and fault-tolerance problems being faced by similar systems handling Internet-scale traffic.

Incremental resource management protocol. Fuxi's resource management protocol allows an application to specify its resource demand at once, and then make incremental updates only if necessary. The protocol saves an application from repetitively asserting full resource demands, and thus significantly reduces the communication and message processing overhead. Resource grants are also offered to applications in iterations. The central scheduler, called FuxiMaster, maintains the state of unfulfilled demands of each application and employs a locality-tree based method to achieve micro-seconds level response.

User transparent failure recovery. Once a user job is submitted, Fuxi ensures that its execution will be carried out until completion and makes the failover process of individual component failures as transparent to the user as possible, with the help of the following techniques: a) For FuxiMaster, we adopt the typical hot-standby approach and lightweight state preservation for its failover. In order to reduce the overhead of state bookkeeping and to accelerate state restoration, we separate the states into soft states and hard states. Soft states could be collected at runtime from other Fuxi components while only hard states like job description need to be recorded. b) For an application master, the FuxiMaster leverages heartbeat to determine whether to start a new master or not. The application master can also conduct failover to recover the finished and running workers by means of a light-weighted checkpoint scheme; c) For the daemon process on each machine, existing running tasks will be adopted rather than being killed.

Faulty node detection and multi-level blacklist. We adopt a multi-level machine blacklist method to effectively identify machines behaving abnormally yet not dead. In Fuxi, both FuxiMaster and application masters employ machine blacklist mechanisms. FuxiMaster uses three schemas to find out the bad machines while application master makes use of a bottom-up approach to distinguish temporary abnormality from persistent bad machines. Additionally, FuxiMaster and application masters share the blacklist to make collaborative judgments for faulty nodes.

Our evaluation results demonstrate the major advantages of Fuxi. Under a stressful synthetic workload, the average scheduling time for each request is merely 0.88ms while 95% memory and 91% CPU can be planned out by FuxiMaster. Additionally, we achieve a 2.36TB/minute sort throughput from GraySort, a 66.5% improvement over the previous record by Yahoo! [8]. With a 5% fault-injection rate, the same Fuxi job only experiences a 15% slowdown. Even after we double the fault-injection rate to 10%, the slowdown only grows to 20%. Fuxi has been deployed in our production environment since 2009. It now manages hundreds of thousands of server nodes at Alibaba.

The remaining sections are structured as follows: Section 2 discusses the system overview and core design philosophy. Section 3 focuses on resource management while section 4 describes job scheduling, including job execution and fault tolerance handling; Section 5 presents the experimental results, followed by related work in Section 6. We conclude our paper and discuss future work in Section 7.

2. SYSTEM OVERVIEW

In this section, we provide an introduction of the Apsara cloud platform, and then present an overview of Fuxi system.

2.1 Apsara Cloud Platform

Apsara is a large-scale general-purpose distributed computing system developed by Alibaba Cloud Computing Inc [1] (aka Aliyun). Apsara is responsible for managing the physical resources of Linux clusters within a data center and controlling the parallel execution of distributed applications. It also hides low-level management chores such as failure recovery and data replication, thereby providing a highly reliable and scalable platform to support distributed processing with many common computing models. Apsara is the common foundation for a suite of cloud services offered by Aliyun, such as elastic computing, data storage and large-scale data-driven computation.

The overall architecture of Apsara is shown in Figure 1, where the boxes in skyblue are Apsara modules, and the white boxes are cloud services running on top of Apsara. Apsara consists of four major parts: 1) common low-level utilities for distributed computing, such as distributed coordination such as locking and naming, remote process calls, security management, and resource management; 2) distributed file system; 3) parallel job scheduling; and 4) cluster deployment and monitoring. The resource management and

[1]Fuxi (pronounced (\ˈfuʃiː\)) derives from the first of the Three Sovereigns of ancient China and the inventor of square and compass, trigram, Tai-Chi principles and calendar [4], thereby being metaphorized into a powerful dominator in our system.

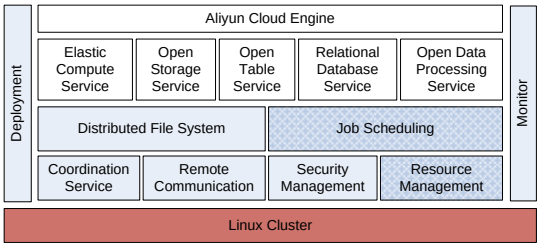

Figure 1: Apsara system overview.

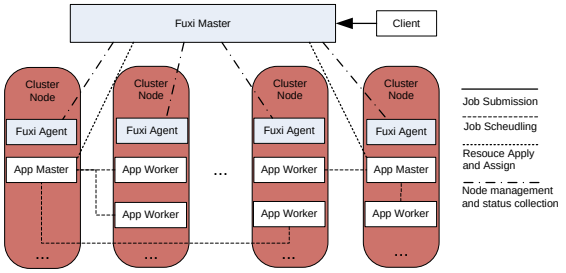

Figure 2: An application workflow in Fuxi system.

job scheduling modules are collectively called Fuxi and is the topic of this paper.

2.2 Fuxi Overview

As shown in Figure 2, Fuxi follows a common master-slave architecture and it has three components: central resource manager (called FuxiMaster), the distributed node managers (called FuxiAgent), and the application masters.

FuxiMaster: In order to schedule cluster resources among different applications, decisions are made from both producers and consumers points of view. For example, how many resources are available for scheduling and what each application's specified requirements are. It is FuxiMaster that acts as the match-maker in between. It not only passively collects total free resources/virtual resources from each machine but gathers resource requests from all application masters as well. Due to the highly dynamic information from both sides, FuxiMaster must collect information in time and perform the scheduling quickly and fairly. While the resource assigned is no longer needed by the application, FuxiMaster ought to reschedule it to another application as soon as possible for higher cluster utilization. After scheduling, results will be delivered to the related application masters and FuxiAgents. Thereafter, the master can use this resource to start requisite computation processes with the guaranteed resource amount and the isolation for each application process supported by FuxiAgent based on the scheduling instructions.

FuxiAgent: A single FuxiAgent will run on each machine, mainly serving two-folded roles. The first is to collect local information and status periodically, and report them to FuxiMaster for further scheduling judgement. The second one is to ensure application processes to execute normally with the aid of process monitor, environment protection and process isolation.

To achieve process isolation, we have adopted three schemes which is the most important responsibility on FuxiAgent. Firstly, FuxiAgent will start processes for one application only if it has obtained sufficient resource on this machine from FuxiMaster. We call this procedure resource capacity ensurance. Specifically, when the resource capacity decreases and application master does not choose one process to stop, FuxiAgent will kill one process of this application compulsorily to ensure the resource capacity. Secondly, each process is configured with Cgroup [3] soft and hard limit. When a machine encounters with resource overload, one or more processes will be killed to maintain the machine load within an acceptable threshold. One simple rule is to select the process whose real resource usage exceeds its own resource usage most. Thirdly, sandbox is leveraged to isolate different processes from invalid operations such as file access. In fact, different root folders are created for each process preventing interference and resource access from others.

ApplicationMaster: Different computation paradigms (e.g. MapReduce, Spark, or Storm etc.) would be represented by different application masters that are responsible for application-specific execution logic.

Common Workflow: Figure 2 illustrates how these components cooperate with each other. A user firstly submits a request to launch an application (e.g. a MapReduce job) to FuxiMaster with the *description*, which contains the necessary information such as application type, master package location and application-specific information. FuxiMaster then finds a FuxiAgent with sufficient resources for the application master, and requests that FuxiAgent to start the corresponding application master. Once started, the application master retrieves the application description, and determines the resource demand for different stages of the job execution, including the appropriate parallelism, the granularity of each allocation, and preferred locality. Afterwards, it sends resource allocation requests to FuxiMaster in an incremental manner and waits for resource grants and revocations (if any) from FuxiMaster. Upon the receipt of resource grants, the application master sends concrete *work plans* to the corresponding FuxiAgents. The work plan contains the necessary information to launch a specific process, such as its package location, resource usage limits and startup parameters. Upon receiving a new work plan, FuxiAgent starts the application worker and uses Linux Cgroup [3] to enforce resource constraints. FuxiAgent watches the worker's status and restarts it if it crashes. In the meantime, the application worker also registers itself to the application master. When a worker is no longer needed, the application master makes an incremental request to returns the granted resource back to FuxiMaster.

3. INCREMENTAL RESOURCE MANAGEMENT PROTOCOL

In this section, we introduce an efficient resource management protocol that allows Fuxi to support the scale at Alibaba. We first introduce the basic idea behind the protocol design. We then describe the key data structure used in the protocol. Afterwards, we describe the scheduling algorithm, which is centered around a data structure called a *locality tree*. We conclude the section with a few other considerations for multi-tenancy and scalability.

3.1 Basic Idea

Incremental Scheduling: Consider a cluster with hundreds of thousands of concurrent tasks, running for tens of

seconds on average, the resource demand/supply situation would change tens of thousands times per second. Making prompt scheduling decisions at such a fast rate means that FuxiMaster cannot recalculate the complete mapping of CPU, memory and other resource on all machines to all applications tasks in every decision making.

With the locality tree based incremental scheduling, only the changed part will be calculated. For example, when {2CPU, 10GB} of resource frees up on machine A, we only need to make a decision on which application in machine A's waiting queue should get this resource. There is no need to consider other machines or other applications. Microseconds level scheduling can be achieved in light of this intuitive but effective locality-based approach.

Incremental Communication: Similarly, in the environment consisting of thousands of applications and machines, the massive message communication among different components (such as FuxiMaster, application masters and FuxiAgents) will be a non-negligible factor that greatly affects the performance of FuxiMaster. An application may require hundreds of thousands of processes to run. A simple iterative process that keeps asking for unfulfilled resources will take too much bandwidth and get worse when cluster is busy.

For this reason, we try to reduce the flooding messages by only sending messages from application masters and FuxiAgents to FuxiMaster when changes occur. In addition, only the delta portion will be transferred. Applications could publish their resource demands in incremental fashion. In the simplest form, an application only specifies resource demand once and waits for resources being assigned iteratively, and returns the resources back when it exits. On the other hand, such a protocol requires states to be maintained in both FuxiMaster and application masters, and many problems arise in keeping the states consistent across different entities. For example, we must ensure the changed portions be delivered and processed in the same order at the receiver side as they are generated on sender side. Additionally, we must ensure the idempotency of the handling of duplicated delta messages, which could happen as a result of temporary communication failure. In a word, we must make sure the full version of information on two communication peers is exactly the same. As a safety measurement, application masters exchange with FuxiMaster the full state of resources periodically to fix any possible inconsistency.

As shown in Figure 3, a simplified example is given to demonstrate the principle of incremental scheduling and communication. We define ScheduleUnit to be an unit size description of resource such as 1 core CPU, 1GB memory. Number 1 to 8 in small rectangles represent steps in temporal order as explained in sequence below:

1) AppMaster1 applies for 10 ScheduleUnits (say SU_A) of resources in the cluster. Detailed requirements include: a). one SU_A contains one cpu core and 2GB of memory; b). if free resource on M1 is available, at least 2 SU_As on M1 are preferred; c). there is no locality preference on other machines; d). maximum 10 SU_As is needed.

2) On receiving request from AppMaster1, FuxiMaster will check the available resource, AppMaster1's group quota availability before scheduling. In this case, 2 units of resource on M1 can be met (actually 3 units are assigned on this machine). After 3 units on M2 and 2 units on M3 are assigned, the application master still needs another 2 units free resources on any machine in the cluster. It is noteworthy that AppMaster1 does not need to update its resource request for SU_A here according to our proposed incremental strategy.

3) AppMaster2 happens to return 1 unit of resource on M3.

4) On receiving the return message, FuxiMaster checks its waiting queue and happens to identify AppMaster1 to be the application with the highest priority. Owing to its unit size much smaller than AppMaster2, 2 units of request can be fulfilled. Therefore, the scheduling decision is to assign 2 units of resource to AppMaster1 after revoking one unit resource from AppMaster2 on machine M3. In time 4, incremental scheduling results are transferred to the related application master and FuxiAgent separately, also in the manner of incremental message.

5) 8 units of resource are returned from AppMaster1 on different machines, with the messages up to FuxiMaster in incremental mode.

6) On the receipt of this message, FuxiMaster conducts the scheduling and assigns these returned resources to other waiting applications in the queue.

7) More resources remained in AppMaster1 are given back in the incremental manner.

8) The same procedure repeats during which FuxiMaster schedules and assigns these returned resources to other waiting applications by means of the proposed incremental communication and scheduling.

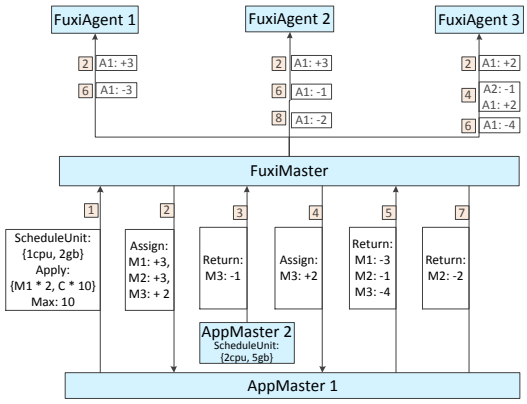

Figure 3: Incremental scheduling and communication

3.2 Key Data Structure

3.2.1 *Resource Description: Both Physical and Virtual*

On large clusters running different types of computations, there is a real necessity for a process to have non-proportional resource demands over different dimensions, e.g. CPU, Memory. Fuxi unifies all these diverse demands into an uniform multi-dimensional resource description, which includes CPU and memory of physical resources at present, but could be easily extended to more dimensions in the future. It can be used to describe the total resources and available resources of a single node and the quantification of a resource request. All resource allocations are based on this resource description abstraction and all dimensions of this description must be satisfied in the meantime.

```
slot_id: 1
max_slot_count: 10
slot_def {
   priority: 1000
   resource {
      resource_type: "CPU"
      amount: 100
   }
   resource {
      resource_type: "Memory"
      amount: 1024
   }
}
Locality_hints {
   value: "r42g04021.aliyun.com"
   count: 2
   type: LT_MACHINE
}
Locality_hints {
   value: "r42g.aliyun.com"
   count: 10
   type: LT_RACK
}
```

Figure 4: Resource request example.

Apart from physical resource, we also introduce the concept of 'virtual resource', for applications to easily limit the concurrency of a certain task on a node. For example, to run a distributed *sort* application called ASort in a cluster, if we only allow 5 concurrent computing processes to be run on the same node, we can configure each node to only contain 5 virtual resource. We can give the virtual resource a name (e.g., 'ASortResource') and the ASort application must be configured to request one 'ASortResource' for each computing process. The total virtual resource on each node can be changed at any time.

3.2.2 Resource Request

A resource request consists of ScheduleUnit definition, quantities for each ScheduleUnit, and other attributes (location preference, avoidance machine list, priority etc.). It is sent from application master to FuxiMaster to apply for resources.

ScheduleUnit is an unit size description of resource(s) such as {1 core CPU, 1GB Memory}. Resources can be fallen into categories of three-level-tree hierarchy: machine, rack and cluster. A machine can have dozens of CPU cores and gigabytes of memory while a rack consists of tens or hundreds of machines. Correspondingly, tens of racks with thousands of machines constitute a cluster. Each application can request resources on one specific machine, or any machine in certain rack or machine inside the cluster. This is mainly determined by data locality preferred by application computation processes (e.g., computation at best happens where data resides or at least within the same network switch).

Any subsequent resource request is based on this ScheduleUnit and it only needs to specify the number of this unit on each machine/rack/cluster (See Figure 4). Each application can have multiple ScheduleUnits and each unit has same or different priority, but different unit size. Scheduling in FuxiMaster will be based on different scheduling units.

Furthermore, application master can change its resource request dynamically at any time after the first submission of request. The quantities can be either positive or negative, meaning increase or decrease of resource request respectively. This will probably happen when location preference is adjusted due to the fact that some bad nodes are detected, or unused resources are returned, or more resources are required for backup running of some instances etc.

To take a map-reduce job with 10 mappers and one reducer as an example, we need to firstly request for 10 units of mapper. If only 6 units are granted from FuxiMaster, it is unnecessary to send new requests to FuxiMaster for additional 4 units of resource. FuxiMaster will automatically insert the request into the scheduler's waiting queue. When free resource is available, additional 4 units of resource will be granted to the application master subsequently. When some mappers finish, the application master returns the resource via the same protocol, but only the unit number needs to be sent.

3.2.3 Resource Response

A resource grant offers application master the privilege to run worker processes on a specific node. No matter which level of resource (machine, rack or cluster) the application master requests, resource on specific machines will be granted to each application after scheduling. FuxiMaster will notify the application master of resource grant in response message in the form of (M_1,3), (M_2,4), ..., (M_n,1) with each resource amount attached. The quantities can be either positive or negative, indicating grant or revocation of resource respectively. A resource revocation means that a previously offered resource grant is no longer valid, possibly due to reasons such as node down or preemption etc. Application master might react to the message by terminating the corresponding worker at an opportune time, or the worker will be involuntarily terminated by FuxiAgent.

Fuxi separates the notion of *task* (the application process that performs the actual work) and *container* (the unit of resource grant). Once an application master receives a grant, it explicitly controls its life-cycle and may reuse the container to run multiple tasks. This is one of major differences between Fuxi and Yarn. In Yarn's implementation, there is no separation between a task and the resources for the task. Whenever a task completes, the node manager always reclaims back the resources, even though the application master has more ready tasks to execution. Thus the resource manager has to conduct additional rounds of rescheduling, thereby creating substantial overhead and unnecessary request messages.

3.3 Locality Tree Based Scheduling

FuxiMaster scheduler maintains two data structures: available resource pool and locality tree. Upon the receipt of resource requests from application master, FuxiMaster will check the free resource pool, and try to find free resource which meets the application's locality requirements. Meanwhile, load balance will also be considered. If the free resource is insufficient, the resource requests will be queued in FuxiMaster's scheduler. Different machine, rack and cluster have their individual waiting queue and applications that request resource on the same machine, rack or cluster will be put into the same queue. These queues on machine, rack and cluster constitute a locality tree.

In particular, this locality tree based scheduling is another main difference between Fuxi and Yarn. Fuxi queues these unsatisfied resource requests in FuxiMaster scheduler, and automatically grants resource to application master upon resource free-up. In this way, we can speed up resource free-up and re-scheduling while increasing cluster utilization. When resources of one machine are returned by one applica-

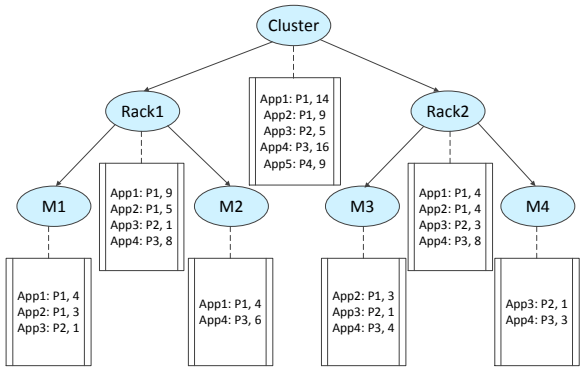

Figure 5: Scheduling tree example.

tion master, certain waiting application will be selected to get the released resources. Priority is the principal consideration, and the application with higher priority will get the resources first. When priority is the same, the waiting time will be taken into account. Additionally, applications waiting on the machine queue will take precedence over those waiting on the rack/cluster queue that the machine belongs to. The objective is to ensure cluster overall locality computation if priority is identical. An application can choose to wait on a specific machine, certain rack or any machine in the cluster and all applications waiting on the same tree are sorted by priority and submission time. Figure 5 can be illustrated as an example of the scheduling tree. App1 waits for each 4 units of resource on M1 and M2 with 9 units, 4 units and overall 14 units on Rack1, Rack2 and the whole cluster respectively. When any of these waiting requests can be satisfied, the resources will be assign to App1 and the relevant waiting requests will be decreased by the amount of assigned units.

3.4 Other Design Considerations

Multi-Tenancy Support: As for the fairness, resource quota concept is introduced. In this manner, one cluster can have multiple quota groups while each application must belong to one and only one group. When applications from one quota group are idle and cannot take up all resources, applications from other quota groups can exploit it instead. When all quota groups are busy, a minimal quota for each group will be ensured. Dynamic quota adjustment is outside the scope of this paper and will not be introduced in detail.

To enact the quota redistribution, FuxiMaster must revoke some resource grants from applications with quota deficits via preemption. When resource requests of applications from one quota group increase and the minimal resource quota is not satisfied, the quota groups that over-use resources will be preempted to make space for this quota group.

Besides quota preemption, we provide preemption on priority, which targets the scenario that the application with higher priority submits its resource request late but the cluster resources happen to be all scheduled out. Applications with lowest priority in its quota group will be preempted to make space for higher ones. The second level is quota preemption.

Prioritized Request Handling: As the only central master in the cluster, FuxiMaster is prone to become bottleneck as numerous messages and requests have to be transmitted, handled and responded in time. For example, when thousands of jobs are started in a batch in a production cluster at midnight, FuxiMaster might encounter spike requests asking for resource. To make it even worse, FuxiMaster also takes important responsibilities for resource scheduling, message disseminations among a great many of peers, which will occupy FuxiMaster a great amount of handling time. Additionally, failure is such a routine phenomenon that cluster availability guarantee and fault tolerance issues are supposed to be tackled in FuxiMaster, which will substantially aggravate its burden.

In order to disperse the spike requests and alleviate the burden, multi-level and prioritized response mechanism is proposed in which we classify the work done by FuxiMaster into different levels of importance and latency requirements. In this scenario, the request handling order is determined according to the emergency priorities. Specifically, urgent requests like resource reversion and re-assignment will be triggered by events in order to offer timely response whilst achieving higher cluster utilization. Furthermore, some similar requests (e.g., frequently changing resource requests from one application) are merged compactly and handled in a batch mode, mitigating the communication overheads. In contrast, other heavy but not emergent requests such as quota automatic adjusting or bad node detection will be captured at a fixed time interval (e.g., once a minute) in a roll-up manner. Owing to these strategies, highly imperative items can obtain more processing time and instant responses, thus improving the effectiveness of resource management.

4. FAULT TOLERANT JOB SCHEDULING

Different computation paradigms can be implemented on top of Fuxi's resource management framework. In this paper, we focus on one specific batch dataflow processing programming model and how we can achieve fault-tolerance in job scheduling. In the following sections, we will firstly show the overview of Fuxi job model, and then discuss two techniques: user-transparent failure recovery and multi-level blacklisting. Finally, we share our experience on practical optimizations.

4.1 Overview

Fuxi Job uses DAG (Directed Acyclic Graph) to present the general user workflow, that is similar to other systems like Tez[2] and Dryad [12]. It is very simple to configure the Fuxi DAG job. The framework accepts a JSON file as job description. We take Figure 6 as an example to explain the job description. The JSON file has a field "Tasks" which describes the properties of each task including the executable binary path and other user customized parameters. The field "Pipes" depicts all the data shuffle with each one having a "Source" and "Destination" access point associated with tasks. The circular numbers clarify the correspondences between JSON file and DAG figure on the right side. Users embed their task logic by using Fuxi Job SDK and programming API. For data shuffle, we encapsulate the common data operators like *sort*, *merge-sort*, *reduce* into a library named Streamline along with the released SDK. The library can facilitate the usage of Fuxi to customize user logic of a job.

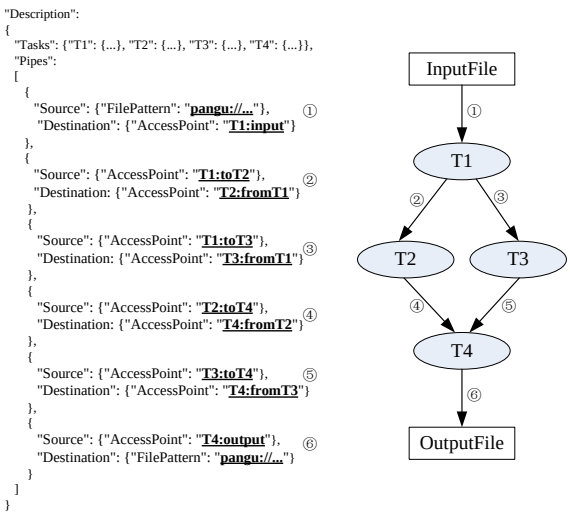

Figure 6: Job description sample.

The major consideration in design of Fuxi Job model is fault tolerance. We design user transparent failover schema to deal with failure of FuxiMaster or FuxiAgent crash. We also design the JobMaster to recover all the instances states including the running instances when the JobMaster restarts. Moreover, a multi-level blacklist strategy is adopted to handle general machine failure such as disk hang, network disconnection etc.

4.2 Job Execution

We provide a plenty of command line tools for users to manipulate the job. The whole life-cycle of a job execution consists of the following stages, all of which are fully automatic except for job starting and stopping by user.

Job Submission: The user should firstly write the specific task logic code based on the interface functions defined in Fuxi SDK. Thereafter, the user code is compiled and built to an executable binary packed in a gzip file. The package is then unloaded to Fuxi system. After the preparation, user can submit the job with the description JSON file to FuxiMaster. Consequently the job life-cycle begins at this time.

Job Master Launch: When receiving a job submission, the FuxiMaster schedules an available FuxiAgent to launch the JobMaster process and sends the command to it. The selected FuxiAgent will quickly start the JobMaster process which will report job running status to FuxiMaster.

Instance Scheduling: After the JobMaster parses the job description, it will request proper resources from FuxiMaster by the resource protocol. The request contains both machine level preference and arbitrary machine in cluster. Based on the offered resource by FuxiMaster, JobMaster determines which task to be executed and the corresponding TaskMaster begins to schedule its instances. The data locality and load balance is taken into account during the scheduling. When failure is found in instances or machines, TaskMaster will re-schedule the instances to be tolerant of disk error or network congestion.

Job Monitoring: All TaskWorkers will periodically report their status including execution progresses to the TaskMasters. User can also query the whole job status from JobMaster by command line tool. Moreover, instance failure details are encapsulated in the reported status for the sake of easy fault diagnosis.

4.3 Fault Tolerance

From the viewpoint of job execution, the faults can be categorized into three types: a) FuxiMaster, FuxiAgent and JobMaster process failure which might result in failure of the whole job; b) machine or network fault which leads to failures of the instances running on it; c) instance long-tail due to hardware and/or software performance dropdown.

4.3.1 User Transparent Failure Recovery

All the roles in Fuxi system, including FuxiMaster, FuxiAgent and JobMaster can support user transparent failover. Here we describe the detailed design and implementations.

1) ***FuxiMaster Failover***: FuxiMaster as the central managing point has a significant impact on the system overall availability. We adopt the typical hot-standby approach and start two FuxiMaster processes in a cluster for high availability. These two masters are mutually excluded by using a distributed lock on the Apsara lock service. The primary master that has grabbed the lock will take charge of resource scheduling while the other master is standby. When the primary FuxiMaster crashes, the standby will immediately grasp the lock and become the new primary master.

To reconstruct whole resource scheduling results, FuxiMaster needs the checkpoint of all states before its crash. On the other hand, full record checkpoint of all states could heavily affect the scheduling performance of FuxiMaster. In order to reduce the overhead of state bookkeeping and accelerate state restoration, we separate the states into *hard states* and *soft states*. Only *hard states* such as job description and cluster-level machine blacklist are recorded by a light-weighted checkpoint. The checkpoint is conducted only when the job is submitted or stopped. The *soft states* are collected from all FuxiAgents and application masters at runtime during FuxiMaster failover.

Figure 7 depicts the requisite information when FuxiMaster experiences the failover. Only application configurations are loaded from checkpoint while other scheduled resource information is all collected from FuxiAgents and application masters. Each application master re-sends its ScheduleUnit configuration, resource request and location preference to FuxiMaster. Meanwhile, each FuxiAgent re-sends the resource allocation on this machine for each application master. FuxiMaster will use the information above to reconstruct the scheduling states, thus keeping all resource allocation and existing processes stable. Additionally, the application master will keep the already assigned resource during the whole failover.

2) ***FuxiAgent Failover***: FuxiAgent also supports transparent failover. During its failover, FuxiAgent firstly collects running processes started previously, and then requests the full worker lists from each corresponding application master. With the full granted resource amount from FuxiMaster for each applications, FuxiAgent finally rebuilds the complete states before failover.

3) ***JobMaster Failover***: JobMaster process crash could result in whole job failures. Therefore, it is necessary for JobMaster to support user transparent failover in real cluster with thousands of commodity machines. For failover, JobMaster exports a snapshot of all instances' status. The

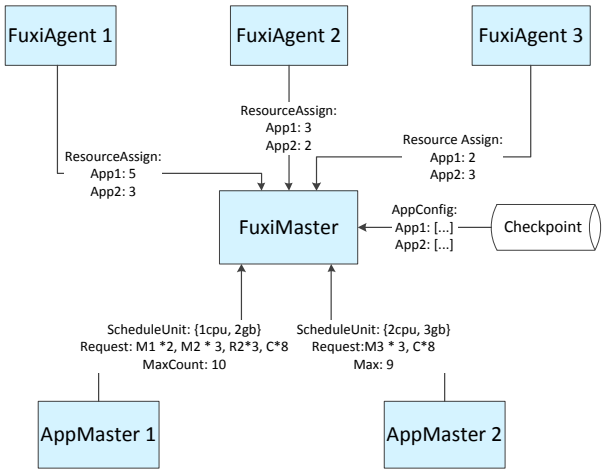

Figure 7: FuxiMaster's failover procedure

snapshot exporting is performed by the event of any instance status change, thus it brings in very little overhead to JobMaster instance scheduling. This kind of job snapshot is also light-weighted since only the status like "Running" is recorded. When the JobMaster process restarts, it will initially load the snapshot of instance status, collect the status from TaskWorker, and finally recover the inner instance scheduling results before its crash. During the absence of JobMaster process, all the workers are still running the instances without interruption. This transparent master failover is extremely beneficial for long-running instances when they are approaching completion.

4.3.2 *Faulty Node Detection and Multi-level Blacklist Mechanism*

Partial failures or performance drop down are notoriously hard to handle by applications and could lead to long tail execution at best or cascading failures that bring down the whole cluster at worst [10]. We design a multilevel machine blacklist to eliminate this kind of machines from resource scheduling.

In the cluster level, we keep a heartbeat between each FuxiAgent and FuxiMaster to indicate the health situation of each cluster node. Once FuxiMaster finds a heartbeat timeout, the FuxiAgent will be removed from scheduling resource list and a resource revocation is sent to JobMaster so that the JobMaster could migrate running instances from the timeout FuxiAgent. We also introduce a plugin scheme to collect hardware information from the operating system to aid judgement of machine health. Disk statistics, machine load and network I/O are all collected to calculate a score. Once the score is too low for a long time, FuxiMaster will also mark the machine as unavailable. With this plugin schema, administrators can add more check items to the list and customize specified error detection.

In the job level, JobMaster will estimate the machine health based on the worker statuses as well as the failure information collected by FuxiAgent. The estimation is also performed in a multilevel way. In particular, if one instance is reported failed on a machine, the machine will be added into the instance's blacklist. If a machine is marked as bad machine by a certain number of instances, this machine will be added into task's blacklist and no longer be used by this task. Among different jobs, FuxiMaster will turn this machine into disabled mode if a same machine is marked bad by different JobMasters. To avoid abuse of this bad machine detection and blacklist, an upper bound limit can be configured.

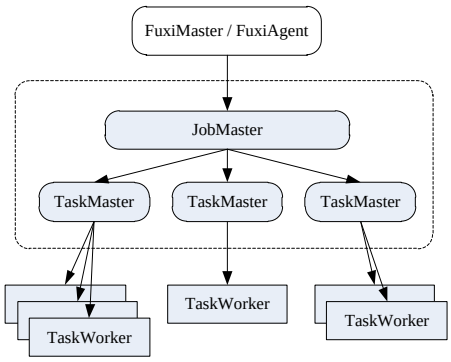

Figure 8: Hierarchical Job Master.

To deal with long tail instances, we also adopt a backup instance schema that will launch another instance with the same input data when the original instance is found running much slower than others. There are three criteria for the backup instance schemes. Firstly, the majority of total instances(e.g., 90%) have finished in which case judgement of long tail instances and estimation of average instance running time can be meaningful. Secondly, the long tail instance must have already run for several times longer than the average instance running time estimated from the finished instances. Finally, the instance can really run for a long time because of input data skew sometimes. To distinguish this kind of instance from the long tail, users should also specify a normal running time of the instances when configuring the backup instance schema in job description.

4.4 Large Scale Handling

There are two provenances of the challenges for large scale job: the performance of scheduling a huge number of instances while taking into account the optimal data locality and load balance; and tons of small shuffle files between massive instances. In this section, we present several techniques used to scale out the job framework in our production environment.

To solve the scale problem, we design a two-level hierarchical scheduling model in Fuxi Job framework. Technically, there is only one JobMaster object for each job, and it takes charge of high-level task scheduling. The JobMaster firstly parses the job description and analyzes the shuffle pipes to figure out the task topological order. Each time only the tasks whose input data are ready can be scheduled and then executed. JobMaster performs the communications with FuxiMaster for resource negotiation and with FuxiAgent to start/stop workers. This hierarchical model is illustrated in Figure 8.

When the JobMaster intends to execute a task, an individual TaskMaster object is created. The TaskMaster will conduct the fine-grained instance scheduling to determine which worker to execute each instance. For the instance scheduler, we design an efficient algorithm taking the following factors into account: a) instances will be scheduled to the worker with the most local input data; b) instances

Table 1: Statistics on a production cluster

	avg	max	total
Instance Number	228/task	99,937/task	42,266,899
Worker Number	87.92/task	4,636/task	16,295,167
Task Number	2.0/job	150/job	185,444

are scheduled to available workers uniformly, thus making computing and network load balanced; c) the scheduling is performed incrementally by scanning only the unassigned instances each time. It has been observed that less than 3 seconds is taken to schedule 100 thousand instances, which demonstrates the effectiveness of the proposed scheduling algorithm. The scheduled instances are then sent to the scheduled task workers on which the instances are actually executed. In summary, we design a hierarchical job scheduling model which decouples the DAG task scheduling and task instance scheduling, reinforcing the parallelism of task execution efficiently. The experimental results exhibit significant benefits from this model for large scale job.

5. EVALUATION

In this section, we discuss and explain the actual performance of Fuxi system to verify its feasibility and efficiency. We firstly present our practice in real production environment. We then evaluate the scheduling performance when 1,000 synthetic workloads are submitted simultaneously in our cluster. To follow up, Sort benchmarks are utilized to illustrate the high performance and capability of task execution. Finally, several faults injection experiments are conducted in order to demonstrate the robustness of the system as well as the fault tolerance mechanisms.

Our testbed is formed by 5000 servers with each machine consisting of 2.20GHz 6cores Xeon E5-2430, 96GB memory and 12*2T disk. Machines are connected via two gigabit Ethernet ports. All the machines run a version of Linux.

5.1 Fuxi in Production

Fuxi plays a significant role and has been deployed in Alibaba's production system since 2009. Resource in our daily production scenarios contains 7 dimensions, i.e. CPU, memory and 5 other types of virtual resources. Table 1 shows the statistics of our workloads based on the tracelog collected from one production cluster in a short period of time. The trace contains 91,990 jobs and over 185,000 tasks, containing total 42 millions parallel instances. The workload were scheduled onto approximately 16.3 millions worker. The most complex jobs could have 150 tasks in the job and the largest task could have almost 10,000 instances, requiring over 4,600 workers to execute.

5.2 Synthetic Workloads Experiment

5.2.1 Experiment Setup

To evaluate the capability of Fuxi system to handle with scalability and performance issues, we keep 1,000 jobs concurrently running by starting a new job when one job finishes. To simplify the experiment, we use WordCount and Terasort with the following specifications evenly distributed. The number of map instance and reduce instance are (10,10), (100,10), (100,100), (1k,100), (1k,1k) and (10k,5k) in each type respectively. The average execution time ranges from 10 seconds to 10 minutes and each instance resource request is configured as 0.5core CPU with 2GB memory.

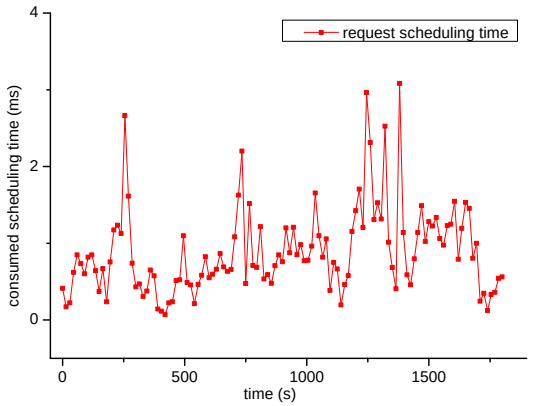

Figure 9: FuxiMaster scheduling time with 1,000 concurrent jobs.

5.2.2 Results Evaluation

We take the following metrics into consideration: scheduling time, resource utilization and scheduling overheads.

FuxiMaster Scheduling Time: We monitor the time cost of FuxiMaster to schedule each request. Figure 9 reveals that the request scheduling time begins to rise as the experiment starts and the average value is merely 0.88ms in spite of a slight fluctuation. Therefore, the transient workload surges will not give rise to the overall performance degradation. Even the peak time consumption for scheduling is no more than 3ms, indicating the system is rapid enough to reclaim the allocated resource and to respond to incoming requests in a timely manner. To summarize, the millisecond level performance is quite reasonable for the overall scheduling process in production scale environment.

Resource Utilization: the memory consumption related metrics of the holistic computing cluster is illustrated in Figure 10(a). The *FM_total* represents the total available resource and 442TB memory in sum among all servers can be scheduled by FuxiMaster. The total amount of assigned resources to all application masters is outlined by *FM_planned*. Apparently, *FM_planned* is roughly 429.26TB indicating that 97.1% resource will be initially utilized by the scheduler.

Additionally, the acquired memory quantity of all application masters is demonstrated by *AM_obtained*. FuxiAgent receives process plan from application master and *FA_planned* shows the total resources consumed by all these processes. The average results for each metrics are 424.56TB and 421.52TB, achieving 95.9% and 95.2% utilization of overall available memory respectively.

Gaps among these curves can be regarded as the overheads of master's ability to process requests. In fact, they are formed by cumulative scheduling impacts and negative communication delays on diverse instances. Obviously, higher throughput and more effective request handling mechanism of the master will give rise to a reduced resource usage gap. We can also draw the conclusion that most of the job requirements can be satisfied because the marginal gap is small enough to be nearly neglected.

Moreover, the results also reveal a very similar phenomenon

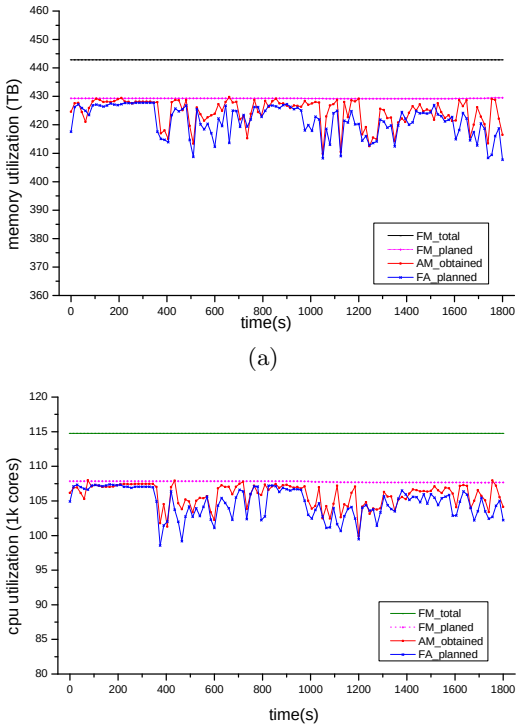

Figure 10: The planned memory and CPU usage by FuxiMaster when 1000 jobs are simultaneously launched.

for the cluster CPU utilization shown in Figure 10(b) reaching almost 92.3% and 91.3% CPU resource usage.

The real memory usage on average is approximately 40%. The reason for this phenomenon is mainly because of user's resource over-estimation. The real CPU usage is less than 10% because our synthetic workloads are memory-intensive with slight CPU stress. Further improvements will be included in the future work.

Scheduling Overhead: Lower overhead indicates rapid arrangement and effective turnover where more task instances can be processed. In this case, we consider the following overheads which are relevant to scheduling effects:

1) JobMaster Start Overhead: the time duration from when the job execution RPC call is invoked to when the corresponding application master starts.

2) Worker Start Overhead: the latency between when application master plans to start a worker and when it receives the first status reported from the worker.

3) Instance Running Overhead: the difference between the instance running time on application master and that on worker.

As shown in Table 2, the total overhead is only 3.9%. The time latency is supposed to be caused by the communication delay among distributed nodes as well as the scheduling throughput especially when a large number of concurrent requests arrive on FuxiMaster resulting in some congestions. Worker start overhead is relatively high due to downloading the worker binaries (average 400MB).

Table 2: Scheduling overhead when 1000 simultaneous jobs are launched

Types	Avg values (s)
Job Running Time	359.89
JobMaster Start Overhead	1.91
Worker Start Overhead	11.84
Instance Running Overhead	0.33

Table 3: Experimental configuration of faults injection.

Injected Fault Type	Node Numbers	Node Numbers
NodeDown	2	2
PartialWorkerFailure	2	4
SlowMachine	11	23
Total Number (Ratio)	15 (5% failures)	29 (10% failures)

5.3 Sort Benchmark

GraySort is a very common benchmark used to evaluate the efficiency of task execution system. In this regard, we conduct 100TB dataset sort and make a comparison with other execution results published in [7]. Table 4 reveals the comparative results and our end-to-end execution time is 2538s which is equivalent to a sort throughput of 2.364TB/minute, achieving 66.5% improvement compared with the most advanced Yahoo's Hadoop implementation. This significant performance improvement benefits from the effective job scheduling mechanisms we presented in this paper.

We also evaluate the PetaSort benchmark in a 2,800 nodes cluster with 33,600 disks and the uncompressed data size is 1 Petabyte with 1x sort spill compression factor. The elapsed time is 6 hours, comparable with Google's result in 2008 [5], which shares a similar architecture with Fuxi.

5.4 Performance of Fault Handling

To illustrate the fault handling abilities proposed in Fuxi system, we design the following scenarios and simulate them by injecting faults into our 300-nodes cluster:

NodeDown: The machine halts unexpectedly. In this case, we randomly shutdown servers in the running period.

PartialWorkerFailure: Disk I/O hang or unstable network connection etc. will undoubtedly lead to unresponsive workers. We can then simulate it by making disk corrupted. The processes thus can not be launched.

SlowMachine: we deliberately add several sleep intervals in the worker program to mock the system slowdown.

FuxiMasterFailure: we shutdown the server on which FuxiMaster runs.

As shown in Table 3, we generate and simulate the 5% and 10% failure scenarios by combinations of different fault types with different machine number. The results illustrate that the execution time extends to 1662s and 1762s separately, producing 15.7% and 19.6% overhead compared with the normal execution time (1437s). We attribute it to our multilevel faulty node detection and backup instance mechanism. Considering the high incidence of failures, it is worth consuming this extra time to ensure the entire workload's execution. Finally, we further terminate FuxiMaster artificially for once based on 5% failure scenario and only extra 13s is needed which can be almost ignored.

Table 4: GraySort Indi result comparison

Provenance	Configurations	GraySort Indi Result
Fuxi (2013)	5000 nodes (2 2.20GHz 6cores Xeon E5-2430, 96 GB memory,12x2TB disks)	100TB in 2538 seconds (2.364TB/min)
Yahoo!Inc. (2012)	2100 nodes (2 2.3Ghz hexcore Xeon E5-2630, 64 GB memory,12x3TB disks)	102.5 TB in 4,328 seconds (1.42TB/min)
UCSD (2011)	52 nodes (2 Quadcore processors,24 GBmemory, 16x500GB disks) Cisco Nexus 5096 switch	100 TB in 6,395 seconds (0.938TB/min)
UCSD&VUT (2010)	47 nodes (2 Quadcore processors,24 GB memory, 16x500GB disks) Cisco Nexus 5020 switch	100 TB in 10,318 seconds (0.582TB/min)
KIT (2009)	195 nodes x (2 Quadcore processors,16 GB memory,4x250GB disks)288-port InfiniBand 4xDDR switch	100 TB in 10,628 seconds (0.564TB/min)

6. RELATED WORKS

In this section, we compare Fuxi system against other existing alternative solutions such as Mesos [11], Yarn [18], Omega [16], Sparrow [14] etc. when tackling with resource sharing and management problems.

Both Mesos [11] and Yarn [18] share the same two-levels scheduling architecture. Mesos provides an offer-based mechanism in which the primary master decides the resource assignment to each individual computing framework. On the contrary, Yarn and our Fuxi system adopt request-based strategy instead, facilitating location-based allocation and further local optimization. Application master could self-estimate and make decisions during the resource allocation process. However, application master in Yarn has to give up the assigned resource and cannot reuse it for those waiting tasks due to the resource reclaim mechanism in resource manager.

As for fault tolerance, Mesos master shares some similarities with FuxiMaster's failover, but Mesos leaves all the things to frameworks themselves to deal with faulty node and individual framework scheduler's failure. Additionally, Mesos slave's failover is not depicted in detail. Moreover, the failover mechanisms for Resource Manager and application master in Yarn are not efficient enough to support larger scale resource management due to the fact that pure and mandatory terminating and redoing of running applications and processes will result in substantial wastes and overheads.

Omega [16] is a shared state scheduler which emphasizes on distributed and scalable merits. The core idea is the lock-free optimistic concurrency control to the shared states which are collected timely from different nodes. In particular, Omega adopts Multi-Version Concurrency Control mechanism, thus substantially improving the concurrency ability of the system. In this case, each framework can compete for the same piece of resource with a central coordinated component for arbitration. However, the evaluation of the model is conducted only based on simulations and needs to be further demonstrated within real production environment.

Sparrow [14] targets a decentralized scheduler design to significantly improve the performance of parallel jobs with a large number of short tasks. With the scalability as well as high availability considered, multiple concurrently running schedulers may work if needed. In this regard, all tasks of a job can be scheduled together rather than scheduling each single task one by one. Sparrow aims to reduce the latency and to improve high throughout for scheduling sub-second tasks. In large scale computing environment, improving the resource utilization is a non-negligible factor and crucial metric to estimate the system efficiency. However, these issues are not presented in Sparrow. Other than short task, Fuxi also support comprehensive-purpose task models including DAG task, long running service etc.

Condor [17] is on behalf of a large category of HPC scheduler solutions. Each job does not need to schedule its tasks local to their data while data locality is a key concern in Fuxi job scheduling mechanism. Furthermore, the resource request for HPC application is announced when launching the job and will not be changed. In contrast, Fuxi will dynamically allocate and revoke the resource to improve the utilization.

7. CONCLUSION AND FUTURE WORK

We have presented Fuxi, a distributed resource management and job scheduling system at Alibaba. We proposed three novel techniques that allow Fuxi to tackle the scalability and fault tolerance issues at Internet scale, including a incremental resource management protocol, a user-transparent failure recovery mechanism; and a faulty-node detection and multi-dimensional blacklisting mechanism. Additionally, we presented various practical considerations and optimization techniques based on our experience of running Fuxi over several hundred thousand server nodes since 2009. We employed a combination of synthetic workload and GraySort benchmark to evaluate the effectiveness of proposed techniques. We found that Fuxi can deliver a 95% memory and 91% CPU scheduled resource utilization under stress load while maintaining a sub-millisecond response time. We beat the previous GraySort record by a margin of 67%. Under a 5% and 10% fault-injection rate, the GraySort benchmark only slows down for 16% and 20% respectively.

A lot of future works remain. Our current scheduling seeks to maximize scheduled resources as requested by applications. However, an application typically would overestimate its resource demands, and thus leaves a big gap between real system utilization and scheduled utilization. We also plan to work on other features such as enforcing IO and networking resource constraints, fine tuning scheduling algorithms to improve backup task scheduling and guard against starvation in corner cases, and better support for short or even interactive jobs.

8. ACKNOWLEDGMENTS

We would like to thank our colleague Jiamang Wang for his elaborate assistance in experiments, our colleague Jin Ouyang and Peter Garraghan from University of Leeds for their kind suggestions. We would also like to extend our sincere thanks to the entire Fuxi and Apsara team members. Finally, we thank VLDB anonymous reviewers for their

helpful feedbacks. Renyu Yang is supported by National Basic Research Program of China (No.2011CB302602), China 863 Program (No.2013AA01A213, 2011AA01A202) and National Natural Science Foundation of China(No.91118008, 61170294). Renyu Yang is the corresponding author of this paper.

9. REFERENCES

[1] Alibaba Cloud Computing. http://www.aliyun.com/.
[2] Apache Tez. http://hortonworks.com/hadoop/tez/.
[3] Cgroup. http://en.wikipedia.org/wiki/Cgroups.
[4] Fuxi. http://en.wikipedia.org/wiki/Fu_Xi.
[5] Google Petabyte Result. http://www.datacenterknowledge.com/archives/2008/11/24/google-sorts-1-petabyte-of-data-in-6-hours/.
[6] ODPS: Open Data Processing Service. http://www.aliyun.com/product/odps/.
[7] Sort Benchmark. http://sortbenchmark.org/.
[8] Hadoop at Yahoo! Sets New Gray Sort Record. https://developer.yahoo.com/blogs/hadoop/hadoop-yahoo-sets-gray-sort-record-yellow-elephant-180650399.html, 2013.
[9] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing(TDSC)*, 1(1), 2004.
[10] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2), 2013.
[11] 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 *Proc. NSDI*. Usenix, 2011.
[12] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proc. EuroSys*. ACM, 2007.
[13] A. McAfee and B. Erik. Big data: The management revolution. *Harvard Business Review*, 10 2012.
[14] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: distributed, low latency scheduling. In *Proc. SOSP*. ACM, 2013.
[15] R. K. Sahoo, M. S. Squillante, A. Sivasubramaniam, and Y. Zhang. Failure data analysis of a large-scale heterogeneous server environment. In *Proc. DSN*. IEEE, 2004.
[16] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proc. EuroSys*. ACM, 2013.
[17] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: The condor experience. *Concurrency and Computation: Practice and Experience*, 17(2-4), 2005.
[18] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proc. SoCC*. ACM, 2013.