Taming Non-local Stragglers using Efficient Prefetching in MapReduce

Ze Yu*, Min Li*, Xin Yang*†, Han Zhao*‡ and Xiaolin Li*

*University of Florida, †Facebook Inc., ‡Qualcomm Research

zeyu@ufl.edu, minli@ufl.edu, xin@cise.ufl.edu, han@cise.ufl.edu, andyli@ece.ufl.edu

***Abstract*—MapReduce has been widely adopted as a programming model to process big data. However, parallel jobs in MapReduce are prone to be plagued by *stragglers caused by non-local tasks* for two reasons: first, system logs from production clusters show that a *non-local task* can be two times slower than a *local task*; second, a job's completion time is bottlenecked by its slowest parallel tasks. As a result, even one single non-local task can become the straggler of the whole job, causing significant delay of the whole job.**

In this paper, we propose to alleviate this problem by *proactively prefetching* input data for non-local tasks. However, performing such prefetching efficiently in MapReduce is difficult, because it requires both application-level information to generate accurate prefetching requests at runtime, and an appropriate network flow scheduling mechanism to guarantee the timeliness of prefetching flows. To address these challenges, we design and implement FlexFetch, which 1) leverages a novel mechanism called *speculative scheduling* to accurately generate prefetching flows, 2) explicitly allocates network resources to prefetching flows using a criticalty-aware deadline-driven flow scheduling algorithm. We evaluate FlexFetch through both testbed experiments and large-scale simulations using production workloads. The results show that FlexFetch reduces the completion time by 41.8% for small jobs and 26.8% on average, compared with the default MapReduce implementation in Hadoop.

***Keywords*-MapReduce; Non-local Straggler; Prefetch; SDN;**

Extracting informative insights from massive data plays a crucial role in both Internet services and scientific discoveries. MapReduce has become the de facto standard to program such data-intensive applications. The proliferation of MapReduce is due to its ability to shield users from managing the complexity of parallelism, scalability and fault-tolerance.

A MapReduce job has three phases: *map, shuffle and reduce*. A map task reads and processes raw input data. Then the generated intermediate results are shuffled to reduce tasks, where the reduce function is applied to generate final results. Typically, the map phase is IO-intensive. A recent study reveals that the map phase constitutes 79% of a job's duration on average in the workloads from Facebook and Microsoft Bing datacenters [5]. Thus, optimizing the input IO performance of the map phase is critical to improving the overall job performance.

Stragglers are those tasks that run slower than other tasks in the same job [23], [4]. In particular, we study the straggler problem caused by *non-local tasks* in this paper. A non-local task has to read input data remotely since its input data is not stored on the node where it's executed. According to the system logs from Facebook [21], a non-local can be $2\times$ slower than a local task, due to the performance gap between local and remote data accessing. Consequently, such non-local tasks significantly prolong the job completion time.

Some recent proposals mitigate the straggler problem using replication at both task level and job level [23], [4]. However, this general approach fails alleviate the non-local straggler problem, since replicated tasks can not read input data locally either. Other locality-aware schedulers [21], [9] improve the overall data locality by *passively waiting* for the chances to launch local tasks. But it is difficult for these approaches to achieve perfect locality for all tasks of a job; even one single non-local task can delay the whole job severely. In summary, most existing research does not efficiently address the straggler problem caused by non-local tasks.

In this paper, we propose to alleviate the non-local straggler problem using *data prefetching*. To be more specific, the input data block of a non-local task is *proactively* prefetched to the computation node where the task will be executed. Prefetching data before they are accessed has a long history in various computer systems [16], [13], [18], [1], [6], [7], [5]. However, incorporating efficient prefetching in MapReduce introduces two new challenges.

First, it is challenging to generate prefetching requests for non-local tasks *accurately*. Most traditional cache prefetching mechanisms assume that the applications follow a certain spatial/temporal access pattern, e.g. sequential access. Under this assumption, the prefetching can be performed without any runtime application information. In MapReduce, however, such access pattern is not prominent, if ever exists. Instead, to accurately prefetch data for non-local tasks, we need to predict *where and when* a non-local task will be launched. Such information, however, can only be obtained at run time from the application layer. As a result, the traditional application-agnostic prefetching strategy can potentially prefetch unnecessary data blocks or misplace them. Thus, they might not improve or even hurt the performance.

Second, like other prefetching systems, we need to make sure that all the prefetched blocks are ready before they are

accessed. But unlike them, prefetching data for non-local stragglers always involves data transferring through the network. Thus, the *timeliness* of prefetching depends crucially on the network performance associated with prefetching flows. However, we lack both efficient mechanisms to allocate network resources for prefetching flows and appropriate network control primitives to implement such mechanisms. Ignoring the interactions between prefetching systems and the network, it is difficult to fully embrace the advantage of prefetching.

These two challenges motivate us to design an integrated prefetching framework for MapReduce, called FlexFetch, which incorporates both application layer and network layer in prefetching data for non-local stragglers. FlexFetch is a paradigmatic example of the *corss-layer* design philosophy. In FlexFetch, data prefetching is managed by a Prefetching Controller (PC). At the northbound, the PC is informed by the Application Controller (AC), i.e. the MapReduce task scheduler, to generate accurate prefetching requests. At the southbound, the PC manipulates the network via network control APIs exposed by a Network Controller (NC). Being both application-aware and network-controllable enables accurate and timely prefetching to mitigate the non-local straggler problem.

In FlexFetch, we devise the following two mechanisms to generate prefetching requests and allocate network resources to prefetching flows, respectively. To generate prefetching requests, we propose a mechanism called *speculative scheduler*, which *pre-executes MapReduce scheduler ahead of time* to predict where and when a future non-local task will need to read input remotely. Then the PC generates prefetching requests according to the speculative scheduler's predictions.

To guarantee the network performance of prefetching, we first utilize a *deadline-driven* approach to translate the predictions of non-local tasks into network rate allocations of the corresponding prefetching flows. One important question to answer here is how to resolve congestions when multiple parallel prefetching flows compete for network resources with each other, and with non-prefetching traffics. We formally describe such parallel prefetching flow scheduling problem and show that it is both NP-Hard and APX-Hard. To solve this problem, we further propose a heuristic-based algorithm called CARDINAL, which captures the *criticality* of prefetching flows to maximize the efficiency of prefetching given the capacity of the network.

Although we focus our discussion on MapReduce in this paper, we believe the design of FlexFetch can be extended to other data-parallel programming frameworks such as Dryad [11] and Spark [22]. In summary, we make the following major contributions in this paper:

- We propose FlexFetch, an integrated prefetching framework in MapReduce, to mitigate the straggler effect of non-local tasks. FlexFetch enables efficient prefetching by establishing coordinated scheduling mechanisms across the application layer, the prefetching layer and the network layer.
- We devise two novel mechanisms in FlexFetch: a speculative scheduling mechanism that generates prefetching requests accurately, and a criticalty-aware deadline-driven algorithm that schedules concurrent prefetching flows.
- We provide an efficient implementation of FlexFetch. Especially, we build the NC module based on the Software-Defined Networking (SDN) technique.
- We evaluate our system through both simulations and testbed experiments. Our results show that FlexFetch improves the baseline MapReduce significantly.

The rest of this paper is orgnized as follows. Section I introduces the motivations and challenges of using prefetching to mitigate stragglers caused by non-local tasks. Section II and Section III describe the design of FlexFetch and the two novel mechanisms it employs. The implementation details are introduced in Section IV. Section V presents our experiment results. We finally summarize related works in Section VI and conclude in Section VII.

I. Background and Motivation

A. Necessity of Prefetching in MapReduce

MapReduce is one of the most popular parallel programming frameworks to process massive data. In MapReduce, the input files are chunked into blocks (64MB per block by default). A job is split into parallel map tasks, each of which processes one input block. A *scheduler* assigns *computation slots* to tasks. If a map task is scheduled on a computation node containing its input block, it directly reads data from the local disk. Otherwise, it must read data from a remote node storing its input data. The execution time of a non-local task can be twice longer than a local task, since accessing data remotely may potentially suffer from network resources contentions in MapReduce clusters.

The straggler effect caused by non-local tasks prolongs the job completion time. Especially, it hurts the performance of small jobs, i.e., jobs that include only a few tasks. Small jobs typically get computation slots for all tasks at the same time [5], [4]. Consequently, even a single non-local task can delay the whole job significantly. On the other hand, small jobs dominate the real-world workloads. For example, Facebook reports that 85% of the jobs use less than ten tasks [21], [5]. Therefore, non-local task straggler problem is pronounced in production environments.

Some recent research efforts focus on improving the *overall* data locality of MapReduce. The traditional approach to achieve this is leveraging *locality-aware schedulers*. Locality-aware schedulers take location information of the input blocks into consideration and strives to schedule a map task on a node which contains a replica of the corresponding input

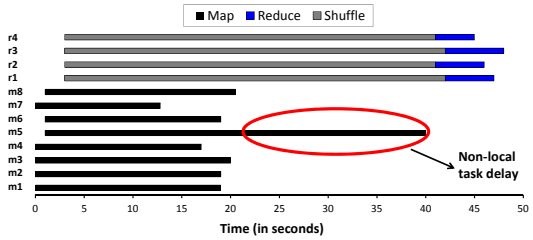


Figure 1. Delay caused by a single non-local map task. The whole job is delayed by 78%.

block [9]. Following this idea, Delay Scheduler [21] further improves locality while providing max-min fairness. Despite the increase of the overall number of local tasks, such locality-aware schedulers do not account for the straggler effect of non-local tasks – one single task will delay the whole job. Figure 1 illustrates an example taken from a real-world experiment[1]. We observe that although the locality is high (nearly 90%), the only non-local task increases the job completion time by almost 80%. Furthermore, the overall locality improvement typically comes at the cost of waiting. For example, to achieve better locality for a job, the Delay Scheduler [21] will skip the slots that are non-local to the job and *passively* wait for the slots to launch local tasks. Even though this approach can achieve higher locality, the extra time spent on waiting might lead to nonnegligible delay, especially for those small jobs or in a medium-sized cluster where the resource return rate is not sufficient to make the waiting time negligible [21].

Other prior research mitigates straggler effect by using replication. These straggler mitigators do not consider the root cause of stragglers. Instead, they just run multiple copies of a task [23] or an entire job [4] and take results from the one that finishes the earliest. This replication-based strategy is not applicable for our case, because replicated tasks can not read input block locally either.

In this paper, we explore a different design space to solve the non-local straggler problem. From our perspective, the root cause of non-local straggler is that most existing schedulers *passively wait* for the slots to be available and strive to match them to tasks so as to maximize locality. Instead, we propose to *proactively prefetch data* for the unavoidable non-local tasks. Prefetching is beneficial: when a non-local task is launched, it can directly read the already-prefetched data locally. Besides, storing the prefetched data in memory further improves the task performance by avoiding expensive disk IOs. In the ideal case, all map tasks are able to access input data locally when they are launched, either from local disks or local memories.

Note that we are not discarding locality-aware schedulers. On the contrary, we believe that these two techniques are perfectly complementary to each other: locality-aware schedulers reduce the number of non-local tasks, keeping prefetching overheads as low as possible; in return, prefetching is a perfect remedy for unavoidable non-local tasks due to imperfect locality-aware schedulers which are incapable of addressing the non-local straggler problem.

[1]This is a WordCount job processing 1GB webpages with 10 nodes

B. Challenges of Prefetching in MapReduce

Prefetching data in MapReduce poses new challenges. Traditionally, prefetching can be implemented as an independent layer in operating systems or file systems, being transparent and agnostic to both the application layer above it and the network layer below it. However, this design is not suitable for MapReduce.

On the one hand, compared with traditional systems, it is challenging to generate effective prefetching requests in MapReduce. Existing prefetching techniques all assume that applications follow a certain spatial/temporal access pattern, e.g. sequential scaning. The prefetching subsystem then leverages such presumed access patterns to generate prefetching requests and brings data blocks from the lower-level locale to the higher-level locale, without involving the applications. In our case, since the ultimate goal is to prefetch data for those non-local tasks, we need the information of when and where a non-local task will be executed. Otherwise, we might prefetch data blocks that are useless to mitigating the non-local straggler problem or place the prefetched data on a wrong node. Both cases are harmful to the cluster performance since prefetching data incurs extra network overheads. Because such information can not be obtained in priori, it is required to dynamically generate prefetching requests in coordination with the MapReduce framework.

On the other hand, the network performance has high impacts on the efficiency of prefetching. Prefetching for non-local tasks means moving their input data to computation nodes through network transferring. As a result, it is highly desirable to dynamically allocate network resources according to the traffic demands of prefetching flows. Without such network performance guarantee, prefetching flows might suffer from severe bandwidth contention with other traffic (e.g., shuffling flows) and among different prefetching flows. Unfortunately, we fall short of both *flexible tools* and *appropriate mechanisms* to control the network. First, we can not make bricks without straw: MapReduce is only able to explicitly allocate local resources, such as compute slots and memory [19], leaving network resource under control of TCP's AIMD mechanism. Such feeble network primitive inhibits us from exerting accurate and flexible control over how to share the network resources. Second, we also need to put an efficient mechanism in place to allocate the network resources to the prefetching flows, resolve contentions and guarantee the performance improvement of jobs.

These two challenges motivate us to 1) design a new prefetching framework that enables the integration of multi-

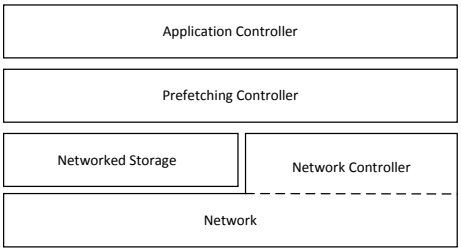


Figure 2. Multi-layer System Model

ple layers (Section II), 2) devise novel mechanisms under the proposed architecture (Section III) to prefetch data efficiently.

II. FlexFetch Design

The discussion in the previous section leads to the following design goals of our MapReduce prefetching system:

- Prefetching requests should be generated according to the dynamic information about non-local tasks, which is provided by the application layer at run time.
- Prefetching system should be able to explicitly allocate network resources according to the demands of prefetching flows.

To meet these design goals, we design the FlexFetch system architecture as illustrated in Figure 2. A Prefetching Controller (PC) lies in the center of the system. Like normal prefetching systems, it generates and sends prefetching requests to the underlying storage system which performs the prefetching. Unlike normal prefetching systems, the PC cooperates with an Application Controller (AC) to obtain accurate predictions of non-local tasks. Furthermore, it explicitly controls the network resource allocation through a high-level network control API exposed by the Network Controller (NC). The NC translates the high-level API invocations to low-level network configurations and installs them on the network devices.

FlexFetch can be instantiated and integrated with MapReduce as illustrated in Figure 3. The AC is the *task scheduler* of Mapreduce. Both PC and NC have one *coordinator* and multiple *daemons*. The PC coordinator obtains the states of the running jobs from the MapReduce task scheduler and generates prefetching requests accordingly. These requests are sent to PC daemons on worker nodes. PC daemons accept requests and call the underlying distributed file system APIs (e.g., HDFS APIs) to fetch data blocks. Besides, PC daemons also manage the local memory to cache fetched data blocks. The PC coordinator also controls network resource allocation through a set of network control APIs exposed by the NC. The NC coordinator accepts network control API invokations and translates them to low-level network control primitives. These low-level network control primitives are then installed on NC daemons. Physically, both the NC coordinator and the PC coordinator can be put

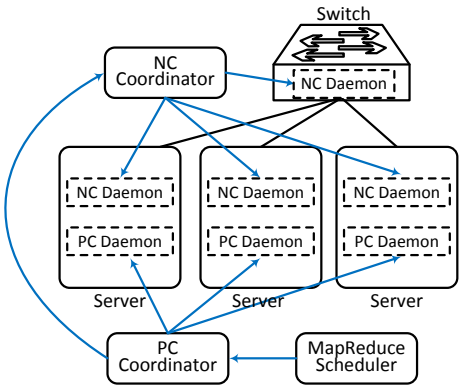


Figure 3. System Architecture

either at the MapReduce master node or any other node of the cluster. Note that PC daemons are only on endhosts but NC daemons are on both endhosts and in-network devices.

Based on the FlexFetch framework, we design appropriate mechanisms to generate prefetching requests, and manage the network resource allocation of prefetching flows. Together they enables accurate and timely prefetching and mitigate the non-local straggler problem of MapReduce. We will present the details in the next section.

III. Prefetching Controller

The function of PC is two-fold. First, it generates prefetching requests for non-local tasks. Second, it invokes the NC's network control API to allocate network resouces to the prefetching flows. This section presents two novel mechanisms – speculative scheduling and Criticality-AwaRe Deadline-drIveN rate ALlocation (CARDINAL) – to realize these two functions, respectively.

A. Speculative Scheduling

The difficulty with prefetching lies in *accurately predicting when/what/where to prefetch*. Especially in FlexFetch, we need to know when and where a non-local task will be launched. However, this is controlled by the AC, i.e., the MapReduce task scheduler[2], and can only be obtained at run time.

To address this problem, we propose a novel concept called *speculative scheduling*. Inspired by the idea of *speculative execution* of application code to generate I/O hints [7], the PC creates a shadow copy of the MapReduce task scheduler, called *speculative scheduler*, which runs ahead of the normal scheduler to generate *pseudo scheduling actions*. It then picks out non-local tasks from the generated pseudo scheduling actions and translates them into prefetching requests described as triples of $\langle node, time, task \rangle$. Figure 4 depicts the whole workflow of the speculative scheduling mechanism.

[2]In the rest of this paper we use AC and task scheduler interchangeably.

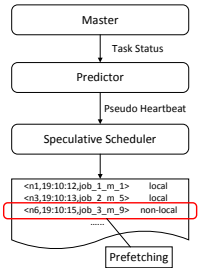


Figure 4. Speculative Scheduling

The speculative scheduler takes *pseudo heartbeats* as input, just like a normal scheduler takes normal heartbeats. Intuitively, pseudo heartbeats are the status reports coming from "the future". The speculative scheduler needs to take speculative scheduling actions when it detects from the pseudo heartbeats that some running task finishes and frees its slots. Thus, we need to predict the future task completion events and piggyback them in the pseudo heartbeats fed to the speculative scheduler.

We use a linear model to predict the completion time of a task as follows. We observe that most MapReduce jobs scan input linearly. Therefore, we can approximate the elapsed time of a map task as a linear function of the size of input data that it has already processed. Formally, $T_e = \alpha \cdot I + \beta$, where T_e is the elapsed time and I is the size of processed input data. To learn this model, we continuously monitor the *progress* of running tasks to get samples of T_e and I. We use a standard linear regression solver to estimate the parameter α and β when new samples piggybacked in pseudo heartbeats are available. We consider the estimatation to be stable when the difference between two consecutive estimatations is less than a threshold. To apply this model in predicting the completion time of running tasks, we just need to replace I with the size of the whole input block and calculate the corresponding $\widehat{T_e}$. Since different tasks of a job might proceed at different speeds, we perform a per task prediction as described above.

B. CARDINAL Algorithm

Next we will describe how FlexFetch guarantees the timeliness of prefetching by explicitly controlling the network. In existing MapReduce clusters, the network is shared among various types of traffics. The default network resource allocation is TCP's AIMD mechanism. However, such an application-agnostic network resource sharing mechanism can potentially lead to unpredicted prefetching time. In this section, we introduce a network control mechanism to guarantee the timeliness of prefetching. We will present the details of the implementation of the mechanism in Section IV-A.

To efficiently allocate network resources to prefetching flows, we need to answer the question of how to resolve the congestions between prefetching flows and other traffics, and among multiple prefetching flows. The solution to the first question is straightforward: we prioritize the prefetching flows over other flows. To resolve contentions among different prefetching flows, we first model it as a deadline-based scheduling problem and show its complexity in Section III-B1. Then we introduce a heuristic based algorithm called CARDINAL (Criticality-AwaRe Deadline-drIveN rate ALlocation) to address this problem in Section III-B2.

1) Problem Analysis: The key insights to guarantee the efficiency of prefetching are as follows:

- A prefetching request generated by the speculative scheduler indicates the time when a future computation slot will be available. If this slot will be assigned to a non-local task and we plan to prefetch data for the task, we must guarantee that the prefetching flow for this task is finished before launching it. Thus, the predicted launching time of a future non-local task indicates the corresponding *deadline* of its prefetching flow.
- Typically, the size of each prefetching flow equals to the block size of the underlying file system (e.g. HDFS), which is is known in advance. Thus, one can determine the minimum flow rate that is required to satisfy the flow deadline. Formally, denote the deadline of a prefetching as d and the block size as B. Then the minimum required rate should be $\frac{B}{d-t}$, where t is the current time.

According to these two observations, the PC allocates required rate to all prefetching flows, so that they can complete before the deadlines. However, if the network does not have sufficient capacity to accomodate all prefetching flows, namely, the bandwidth demands exceed the capacities of some links, the PC should make rate allocation decision in a manner that maxmizes the job level performance.

We formalize our problem as follows. We are given a capacitated directed graph $G = (V, E, C)$ and a set of jobs $\mathcal{J}$. For a job $j_i \in \mathcal{J}$, denote all its prefetching flow as $\mathcal{F}_i = \{f_i^k\}$. Each flow f_i^k is associated with a required rate r_i^k. Let $\mathcal{M}$ be any non-empty subset of $\mathcal{J}$. We call $\mathcal{M}$ *satisfiable*, if 1) all prefetching flows belong to the jobs in $\mathcal{M}$ are allocated their required rates, formally, $\forall j_i \in \mathcal{M}, \forall f_i^k \in \mathcal{F}_i, k = [1, |F_i|]$, we have $f_i^k \geq r_i^k$; 2) the rate allocation satisfies the capacity constraint of the graph. To maximize the overall performance of all jobs, we should *maximize the number of jobs in* $\mathcal{M}$ while *satisfying their prefetching flow rates under the capacity constraint*. This objective is equivalent to finding the largest satisfiable subset of $\mathcal{J}$ (LSS). The LSS problem can be reduced to the *all-or-nothing multi-commodity flow problem* (AN-MCF), which is both NP-hard and APX-Hard [8]. To see this, we just need to let each job contain only one task. Then this simplest version of the LSS problem is equivalent to the standard AN-MCF problem. Also note that the above formalization is for the

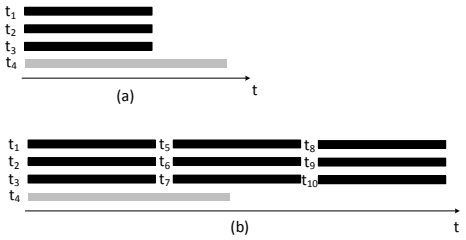


Figure 5. Examples of Critical and Non-critical Prefetching Flows.

offline version of the problem. We claim that the online version of this problem is also NP-Hard and APX-Hard. Thus, in this paper, we seek a heuristic policy to overcome such intrinsic complexity.

2) Criticality-aware Deadline-Driven Rate Allocation: When the network does not have sufficient bandwidth to accomodate all prefetching flows, we have to *abandon* some of them, missing their deadlines and resulting in prolonged completion time of corresponding tasks. To optimally select the abandoned flows, we propose the concept of *criticality* of a prefetching: a prefetching has a high criticality if abandoning it directly results in the delay of the job.

The key insight here is that *jobs with different degrees of parallelism have different criticalities with their prefetchings*. For example, Figure 5(a) illustrates the task launch timeline of a job with four parallel tasks. Suppose task t_4 is non-local. Then it is straightfoward to see that abandoning the prefetching of t_4 will directly prolong the completion time. Figure 5(b), on the other hand, shows a job with a higher degree of parallelism. Note that in MapReduce, it is difficult for a job with large number of parallel tasks to obtain computation slots for all tasks at the same time. Consequently, the delay caused by failing to prefetch data for a non-local task is likely to be masked by future tasks. In Figure 5(b), for example, even if we choose to abandon the prefetching of t_4, the job is not directly delayed due to the overlap between task t_4 and tasks t_5-t_{10}. This intuition gives rise to the following heuristic policy to differentiate the *criticality* of prefetching flow.

Less Residual Task First (LRTF) Policy states that jobs with less residual tasks are more vulnerable to the delay caused by non-local tasks. Thus their prefetching flows have higher criticalities. We should favor *the jobs with less residual tasks when selecting victim prefetching flows to abandon*.

We implement the LRTF policy in an algorithm called CARDINAL. The pseudocode of CARDINAL is shown in Algorithm 1. Given a new prefetching, CARDINAL checks the links on the path of the flow one after another. It skips all links that have sufficient capacity left for the new flow (line 4). For a link that does not have enough capacity, it first ranks all prefetching flows of this link in a decreasing order of their criticalities, i.e., in a decreasing order of the residual

Algorithm 1 CARDINAL Algorithm

Input: f_j^k: new prefetching flow from task t_k of job j; r_j^k: the required rate of flow f_j^k; $\mathcal{L}$: set of all links that flow f_j^k passes
Output: $\mathcal{A}$: set of abandoned flows

```
1: A = ∅
2: for l_i in L do
3:     if l_i has enough capacity to support f_j^k then
4:         continue
5:     else
6:         let S be the set of flows which pass l_i and belong to jobs that have more residual tasks than j
7:         sort S in decreasing order of number of residual tasks
8:         availableCapacity = 0
9:         for f_m^n ∈ S do
10:            A = A ∪ {f_m^n}
11:            availableCapacity = availableCapacity + r_m^n
12:            if availableCapacity ≥ r_j^k then
13:                break
14:            end if
15:        end for
16:        if availableCapacity < r_j^k then
17:            return {f_j^k}
18:        end if
19:    end if
20: end for
21: return A
```

tasks number (line 7). Note that it only considers the flows that are not as critical as the given flow (line 6). Then it scans the victim candidates linearly and abandons the prefetching flows until it accumulates the required bandwidth (line 12). Finally, we simply skip the flow if we cannot satisfy its required rate (line 17).

Another direct advantage of the CARDINAL algorithm is that it favors the jobs with small numbers of map tasks. As such small jobs dominate the real-world workloads [21], [5], we believe that CARDINAL will significantly improve the overall job performance on real-world workloads. We will evaluate the improvement of CARDINAL in Section V-B

IV. Implementation

This section presents the implementation details of FlexFetch, including the network controller (Section IV-A) and the prefetching controller (Section IV-B). Then we discuss about fault tolerance (Section IV-C).

A. Network Controller

The NC exposes a set of resource control interfaces to the PC. It enables the PC to dynamically allocate flow rate for prefetching at runtime according to the prefetching

Table I
NETWORK CONTROL APIS

API	Functionality
Allot(flow f, long r)	allocate rate for flow f
Abandon(flow f)	deallocate rate for flow f
GetPath(flow f)	query the links that flow f passes
GetCapacity(link l)	query current capacity of link l
GetFlow(link l)	query all flows passing link l

flow scheduling mechanism. Described at a high level, the NC should achieve the following design goals: 1) end-to-end per-flow rate guarantee: in order to catch a prefetching flow deadline, we need to guarantee the end-to-end rate for the flow. 2) high utilization: for other network traffics with no rate guarantee, such as shuffling traffics, we should maximize the network throughput.

1) Control API: In FlexFetch, the NC provides network control APIs listed in Table I. Specifically, *Allot* is invoked when the PC needs to allocate bandwidth for a prefetching flow. *Abandon* is invoked when PC decides to abort a prefetching. *GetCapacity* returns the current residual capacity of a link. *GetPath* and *GetFlow* return the links that a prefetching flow passes and all prefetching flows passing a given link, respectively. Note that although our discussion is limited to MapReduce, we believe that the API of NC is a general network control primitive and can be adopted and extended for other parallel programming frameworks as well, such as Dryad [11] and Spark [22].

2) OpenFlow-based Implementation: We choose to implement the network controller based on OpenFlow [15] . The reasons for choosing OpenFlow are as follows. First, it provides a flexible and direct way to control the rate allocation, on both switches and endhosts. Second, the OpenFlow switches are off-the-shelf from the vendors and their performance is verified in production environments [3]. This increases the applicability of FlexFetch in the real-world deployment. Last but not the least, it dramatically decreases the complexity of our implementation. Note that our design does not exclude other implementation by itself, such as using distributed rate control protocols [10].

We identify different flows by assigning a unique ID to each flow. We implement end-to-end per-flow rate allocation by creating minimum rate queues [3]. To be specific, for each flow $f: s \rightsquigarrow t$, we create a minimum rate queue on every forwarding device (i.e. switch and endhost network interface) along the flow path. The minimum rate is set to the required rate calculated by the PC according to prefetching deadlines. Then we setup the forwarding rules on each related device, mapping one flow to its queue. Figure 6 illustrates such queue-based end-to-end rate allocation.

[3]As minimum-rate queues are not yet supported by our OpenFlow switch, we used *ovs-dpctl* instead to implement this function. One can also use OpenFlow of-config protocol if it is supported in the future

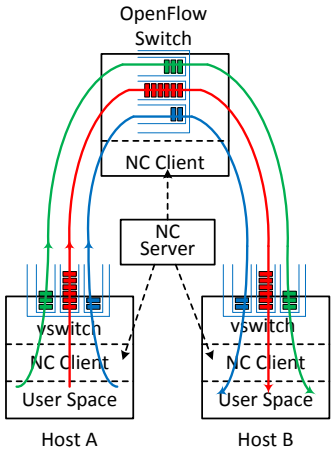


Figure 6. End-to-end per-flow rate allocation. Different prefetching flows (represented by different colors) use different min-rate queues

To reduce the cost of queue setup/teardown operations, we reuse existing queues by remapping new flows to them and adjusting the rate of the queues.

3) Work Conservation: To achieve high utilization of the network resources, we also implement a work-conserving mechanism in the NC. The goal is to first satisfy the prefetching flows with rate requirements and then evenly distribute the spare bandwidth capacity of each link among all flows with no rate guarantee. In our prototype, we achieve this by using work-conserving queues supported by our switches. We create a default queue with no rate guarantee and map all flows that are without rate guarantee to the default queue. Thus, all the spare capacity will be distributed among flows according to TCP's AIMD mechanism within the default queue. This work-conserving mechanism guarantees high utilization of the network resources.

B. Prefetching Controller

In our prototype, we use Hadoop, the de facto standard open source implementation of MapReduce, as our code base. We implement the PC coordinator in the Hadoop job tracker. We also modify the Hadoop task tracker to implement the PC daemon. After generating prefetching requests using the proposed speculative scheduling mechanism, the PC sends these requests to PC daemons. The PC daemons query the meta data of the input block (e.g. file name and block number) and call the HDFS API to fetch the block. Also, we note that MapReduce keeps multiple replicas of input files. In order to reduce the traffic across racks, we prefetch the required block from the replica that is closest to the destination of the prefetching. When a task is launched, the task tracker first queries the local PC daemon for the input block of this task. If miss, the input will be read from HDFS as usual.

Table II
MACROBENCHMARKS

Bin	Job Type	# Maps	# Jobs
1	Search	1	6
2	Select	2	6
3	Search	5	20
4	Select	10	5
5	Search	10	5
6	Aggregation	30	2
7	Search	50	2
8	Select	50	2
9	Aggregation	150	1
10	Search	200	1

C. Fault Tolerant

The failure of FlexFetch does not hinder the normal execution of jobs as data can always be read from the underlying distributed filesystem. However, to improve the reliability of FlexFetch, we do implement the fault tolerant function. Specifically, we exploit a secondary NC coordinator and a secondary PC coordinator as cold standbys. The secondary PC coordinator only receives duplicated job running states from the AC and performs exactly the same operations as its main copy. The secondary NC coordinator keeps up-to-date states of the prefetching flows and the queues. When the NC coordinator is down, the secondary NC coordinator uses these states to reconstruct the global view of prefetching flows and queues. We assume that the main NC coordinator has small downtime and can recover from failure eventually, so that the probability of the event that both the NC coordinator and its secondary backup are down is low. If this worst case really occurs, the FlexFetch stops prefetching input for any task.

V. EVALUATION

We evaluate FlexFetch through both testbed experiments and large-scale simulations. We show that FlexFetch outperforms the default MapReduce implementation in Hadoop, especially for those smaller ad-hoc jobs that dominate the real-world workloads.

A. Macrobenchmarks

1) Setup: We implement FlexFetch and deploy it on a testbed consisting of 50 virtual nodes. Each node has 2 vCPUs virtualized from Xeon E5-2600 and 2GB memory. The physical servers connect to a non-oversubscripted two-level tree network as adopted by most data centers and clouds [4]. The link bandwidth of the network is 1Gbps unless pointed out otherwise.

[4]Due to the lack of multiple switches, we use one 48-ports Pronto 3290 switch to emulate multiple top-of-rack switches

2) Workloads: The goal of our macrobenchmarks is to evaluate the overall performance improvement of FlexFetch over the default MapReduce. Our baseline is Hadoop 0.20.203 with a naïve implementation of the fair scheduler to enhance the responsiveness of small jobs. We call this *baseline Hadoop* in the rest of this section. We generate benchmarks based on the Facebook workloads [5], [21]. Specifically, we mix workloads of different sizes and types as shown in Table II. One important insight taken from the Facebook workloads is that small jobs account for a large majority. For example, about 85% of the jobs have less than 10 map tasks [5]. We scale down the job size to fit in our cluster, so that the jobs can complete in a reasonable execution time. As to the job type, we choose text search and database select jobs (labeled as Search and Select in Table II, respectively) as IO-intensive workloads. We set 0.01% of the input records to contain the searched/selected pattern in the IO-intensive workloads. We also use database aggregation as the communication-intensive workloads (labeled as Aggregation in Table II). In the macrobenchmarks we mix the IO-intensive jobs and communication-intensive jobs at a ratio of $7:3$. All these parameter settings are similar to previous works which follow the traces from Facebook [21].

We also generate job submissions by simulating the job submission pattern in the Facebook workload trace [21]. We distributed the inter-arrival times roughly exponential with a mean value of 14 seconds.

3) Results: The results of the macrobenchmarks are shown in Figure 7 and Figure 8. Figure 7 shows the difference of data locality between FlexFetch and the baseline Hadoop. First, we observe that FlexFetch significantly improves the overall data locality. Note that we count a non-local task whose input data is successfully prefetched as a local one in FlexFetch. Consequently, the locality improvement indicates that FlexFetch turns non-local tasks into local ones by prefetching data, mitigating the straggler problem caused by non-local tasks. Second, we notice that the locality improvement of small jobs is more pronounced. This is because we cannot achieve perfect data locality by merely using the locality-aware scheduler. Especially for these small jobs, even one single non-local task can delay the whole job severely. On the other hand, FlexFetch alleviates such straggler effect by prefetching data for the unavoidable non-local tasks. Finally, we also find that the locality improvement becomes marginal as the job size grows. This is partially due to the fact that, unlike small jobs, the locality of large jobs is already high (e.g., 94% for Bin 10). As a result, there is little improvement space for large jobs. Nevertheless, since large jobs only account for a small portion in production workloads [21], FlexFetch can benefit most jobs in the real-world.

Such improvement in data locality is further reflected in the average speedup, as illustrated in Figure 8. We observe that the average job completion time is reduced by 41.8%

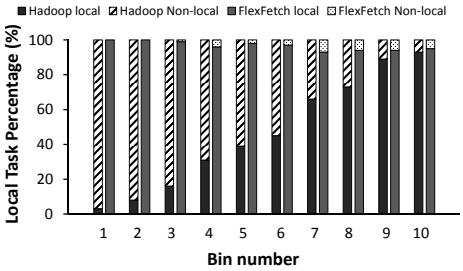


Figure 7. Locality percentage of macrobenchmark

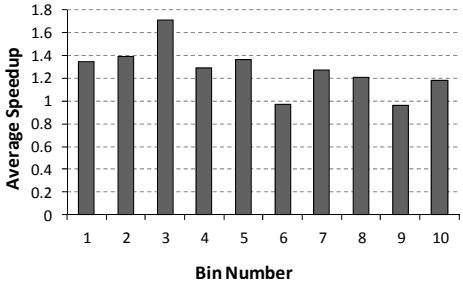


Figure 8. Average speedup of FlexFetch over Hadoop

for small jobs in Bin 1-5 and 26.8% for all jobs. The IO-intensive workloads benefit the most. For example, workloads in Bin 3 show 1.7× speedup on average. Indeed we observe some jobs get up to 2× speedup in this Bin. On the other hand, we also notice that large jobs gain less improvement. The worst case occurs to the communication-intensive jobs in Bin 6 and Bin 9. This behavior is expected because guaranteeing bandwidth for prefetching flows comes at the cost of reducing the throughput of shuffling flows, which dominates the completion time of these communication-intensive jobs. However, as we can see, FlexFetch limits such side-effect within an acceptable range (less than 5% slowdown).

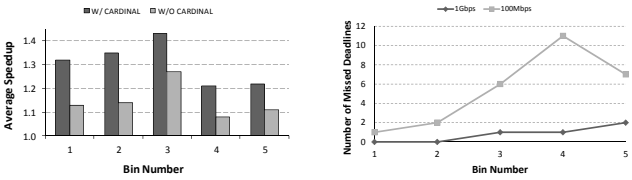
B. Microbenchmarks

In order to understand the benefits of CARDINAL prefetching flow scheduling mechanism, we evaluate the job performance improvement and the missed prefetching deadlines on several microbenchmarks.

First, to understand the feasibility of prefetching under network-constrained environments, we downgrade the link bandwidth from 1Gbps to 100Mbps and repeat the experiments using workloads in Bin 1 - Bin 5. The results are shown in Figures 9(a) and 9(b). Compared with the performance improvement under higher network bandwidth shown in Figure 8, we observe a minor reduction in the speedup ratio when running the same workloads with 100Mbps network. This is an expected behavior because lower network bandwidth leads to a slower prefetching rate. Thus the prefetching flows are more likely to miss their deadlines as shown in Figure 9(b). Nevertheless, we still

Table III
SYNTHETIC WORKLOADS FOR THE EVALUATION OF CARDINAL

Bin	Job Type	# Maps	Map Output Ratio
1	GridMix Scan	5	0.01
2	Search	5	0.01
3	GridMix Scan	20	0.85



(a) Speedup over Hadoop (b) Missed prefetch deadline

Figure 9. Microbenchmarks

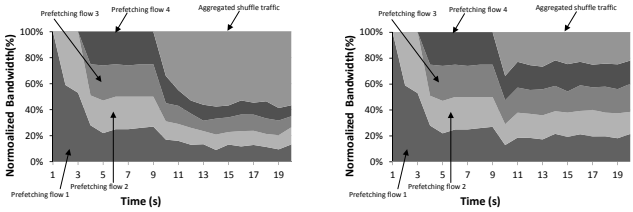
gain more than 1.3× speedup on average.

Furthermore, we observe from Figure 9(a) that if we disable CARDINAL, the advantage of using prefetching becomes marginal. This is a good indicator of the importance and efficiency of CARDINAL: when the network capacity is not sufficient to meet all prefetching flows' deadlines, it first satisfies the jobs which have higher criticalities with their prefetching flows. In this way, CARDINAL maximizes the effectiveness of prefetching and minimizes the job completion time, even if the network is highly-congested.

Details of CARDINAL. To reveal the details of how FlexFetch improves the efficiency of prefetching when the network resource is limited, we also generate a batch of synthetic workloads and run it in a more controlled manner. The jobs we use for this experiment include GridMix Scan [5] and Text Search, whose details are listed in Table III. Bin 1 and Bin 2 are normal IO-intensive jobs and Bin 3 is used to simulate communication-intensive workloads to stress the network.

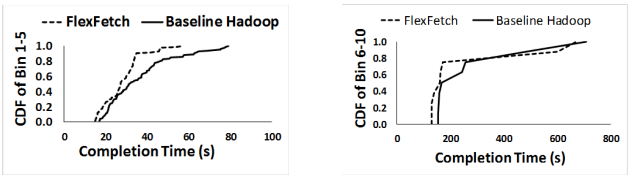
We begin by submitting jobs in Bin 1 and Bin 2 of Table III repeatedly. Then we carefully adjust the submission time of jobs in Bin 3 of Table III, overlapping the huge volume of shuffling traffics incurred by jobs in Bin 3 with the prefetching flow traffics of jobs in Bin 1 and Bin 2. Figures 10(a) and 10(b) show the dynamics of flow rates measured on one worker node. Figure 10(a) demostrates the case without using CARDINAL. There are four prefetching flows sharing the measured link. Initially, these flows equally share the network bandwidth according to the AIMD mechanism of TCP/IP. Then at time 9s, a communication-intensive job finishes a wave of map tasks and the shuffle flows start to congest the network. As one can observe, the shuffle traffic gains more than half of the network bandwidth. This is because a reduce task can start multiple shuffle flows

[5]GridMix Scan [2] is a synthetic job in which each map task scans and randomly outputs its input records according to the given output ratio.



(a) Without CARDINAL (b) With CARDINAL

Figure 10. Effect of CARDINAL



(a) Bin 1-5 (b) Bin 6-10

Figure 11. Simulation Results

concurrently (up to 5 by default), squeezing the bandwidth share of prefetching flows. This behavior results from the AIMD flow control mechanism of TCP. The prefetching speed is reduced by half as Figure 10(a) illustrates. Such unmediated network contention causes the four prefetching flows to miss their deadlines. Missing prefetching deadline not only renders the prefetching invalid, but also hurts the performance of reduce tasks by interfering with their shuffle flows.

Figure 10(b) shows how FlexFetch handles such network contentions. There are two main observations. First, FlexFetch guarantees that the prefetching flows can obtain sufficient bandwidth allocations to complete in time. Second, prefetching flows are assigned "just-enough" bandwidth to catch the deadlines. The spare bandwidth is given to shuffling flows. Consequently, FlexFetch guarantees the effectiveness of prefetching flows while maintaining the overall throughput.

C. Simulation

Due to the limited resource, we conduct simulations to evaluate FlexFetch at a large scale. We implement a simulator called FlexSim to run the simulation. FlexSim includes a MapReduce simulator, similar to the existing NS2-based simulator [20], to simulate the job execution. However, FlexSim differs from the previous work [20] in that it does not perform packet-level network simulations. A packet-level simulation is not necessary for our goal. Instead, we adopt a flow level simulation, which reduces the simulation time and simplifies the implementation of the simulated network control. We validate FlexSim by comparing the simulated job completion time with the results of real cluster experiments. Our validation shows that the error between the simulated and the real experiment results is less than 5% (not shown in this paper due to the space limitation).

We simulate a cluster of 200 nodes using FlexSim. The simulated cluster has a three-level non-oversubscripted fat-tree network with 1Gbps link bandwidth. The nodes are distributed across 10 racks. We repeat experiments using macrobenchmarks on this simulated cluster. The CDFs of job completion time is shown in Figure 11(a) and Figure 11(b). We observe that the simulation results are similar to the testbed experiment results. The short jobs gain significant improvement: more than 40% jobs in Bin 1-5 are improved; the average completion time is reduced by 31%. Such improvement comes at the cost of slowing down some of the communication-intensive jobs in Bin 6-10, because prefetching data for non-local map tasks is prioritized over shuffle traffics. However, FlexFetch keeps this side effect as low as possible.

FlexFetch Overhead. In our testbed experiments we do not perceive noticeable scheduling overhead. To measure the performance of FlexFetch under heavier workloads we use mock jobs to simulate a cluster with 1000 nodes, each of which is with 4 slots. Under this workload, FlexFetch is able to respond to 1000 concurrent tasks within one second on a desktop with 3.5GHz Intel Core i7-3770K. We believe such responsiveness is graciously adequate for most clusters.

VI. Related Work

Improving data locality has been one of the most important design philosophies behind data-intensive parallel programming frameworks. Existing locality-aware schedulers such as Delay [21] and Quincy [12] resolve the trade-off between locality and fairness. However, they do not consider using data prefetching to mitigate the slowdown caused by non-local tasks, which is the main focus of this paper. Our design principle is based on the observation that although network bandwidth is relatively scarce for data-intensive applications, it can be used to prefetch data for non-local map tasks when necessary. Because we combine locality-aware scheduling techniques with prefetching, the network overhead of prefetching can be controlled within an acceptable level, without severely impacting the overall throughput of the cluster.

General-purpose prefetching/caching techniques are widely adopted in various computer systems [5], [16], [13], [18], [1], [6], [7]. Most of these techniques are suitable for operating systems and file systems. To the best of our knowledge, the only work applying prefetching to MapReduce is HPMR [18]. HPMR uses both inter- and intra- block prefetching to improve locality. However, it neither guarantees the performance of prefetching flows nor considers how to accurately generate prefetching requests. As we have demostrated through experiments, unmediated prefetching might not improve or even hurt the performance.

There also exists plenty of work studying the straggler effect in parallel/distributed systems. For example, Kwon et al. [14] propose an automatic skew mitigation approach for map tasks; Ramakrishnan et al. [17] utilize a progressive sampling approach to re-balance workloads among reduce tasks. Another related work is LATE [23] whose purpose is to detect and re-execute straggler tasks in heterogeneous environments such as clouds. In the most recent work, Ananthanarayanan et al. [4] introduce a general straggler mitigation strategy based on job level replications. However, this approach can not solve the non-local task straggler either. In summary, FlexFetch differs from the prior proposals: it focuses on mitigating the straggler effect caused by non-local map tasks, which is not fully studied by previous work.

VII. Conclusion

This paper proposed a prefetching-based solution in MapReduce to mitigate the straggler problem caused by non-local tasks. The challenges in realizing such efficient prefetching are how to accurately generate prefetching requests and how to guarantee sufficient network resources for the prefetching flows. To address these challenges, we designed an integrated prefetching framework called FlexFetch. In the FlexFetch framework, we speculatively executed the application scheduler ahead of time to generate accurate prefetching requests. At the same time, we efficiently scheduled prefetching flows by capturing their criticalities. Through both private cluster testbed and large-scale simulations using production workloads, we showed that FlexFetch shortens the completion time of small jobs by 41.8% and 26.8% on average.

References

[1] The Global Memory System (GMS) Project. http://homes.cs.washington.edu/~levy/gms/. Accessed: 03/20/2013.

[2] The Hadoop GridMix Benchmark. http://hadoop.apache.org/docs/r1.1.2/gridmix.html. Accessed: 03/20/2013.

[3] Openflow at Google. http://opennetsummit.org/archives/apr12/hoelzle-tue-openflow.pdf. Accessed: 03/20/2013.

[4] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective straggler mitigation: Attack of the clones. NSDI '13, 2013.

[5] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. Pacman: coordinated memory caching for parallel jobs. NSDI '12.

[6] M. Annavaram, J. M. Patel, and E. S. Davidson. Call graph prefetching for database applications. *ACM Trans. Comput. Syst.*, 21(4):412–444, Nov. 2003.

[7] F. Chang and G. A. Gibson. Automatic i/o hint generation through speculative execution. OSDI '99.

[8] C. Chekuri, S. Khanna, and F. B. Shepherd. The all-or-nothing multicommodity flow problem. STOC '04.

[9] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. OSDI '04.

[10] N. Dukkipati. *Rate Control Protocol (RCP): Congestion control to make flows complete quickly*. PhD thesis, Stanford University, 2007.

[11] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. EuroSys '07.

[12] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: fair scheduling for distributed computing clusters. SOSP '09.

[13] T. M. Kroeger and D. D. E. Long. Design and implementation of a predictive file prefetching algorithm. In *USENIX ATC '01*, 2001.

[14] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skewtune: mitigating skew in mapreduce applications. SIGMOD '12.

[15] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, Mar. 2008.

[16] V. N. Padmanabhan and J. C. Mogul. Using predictive prefetching to improve world wide web latency. *SIGCOMM Comput. Commun. Rev.*, 26(3):22–36, July 1996.

[17] S. R. Ramakrishnan, G. Swart, and A. Urmanov. Balancing reducer skew in mapreduce workloads using progressive sampling. ACM SoCC '12.

[18] S. Seo, I. Jang, K. Woo, I. Kim, J.-S. Kim, and S. Maeng. Hpmr: Prefetching and pre-shuffling in shared mapreduce computation environment. In *CLUSTER '09*.

[19] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha. Sharing the data center network. NSDI '11.

[20] G. Wang, A. R. Butt, P. Pandey, and K. Gupta. A simulation approach to evaluating design decisions in mapreduce setups. In *MASCOTS '09*.

[21] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Eurosys '10*.

[22] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. NSDI '12.

[23] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. OSDI '08.