Speculative Pipelining for Compute Cloud Programming

H. T. Kung, Chit-Kwan Lin, Dario Vlah, Giovanni Berlanda Scorza
Harvard University
Cambridge, MA

Abstract

MapReduce job execution typically occurs in sequential phases of parallel steps. These phases can experience unpredictable delays when available computing and network capacities fluctuate or when there are large disparities in inter-node communication delays, as can occur on shared compute clouds. We propose a pipeline-based scheduling strategy, called speculative pipelining*, which uses speculative prefetching and computing to minimize execution delays in subsequent stages due to varying resource availability. Our proposed method can mask the time required to perform speculative operations by overlapping with other ongoing operations. We introduce the notion of "open-option" prefetching, which, via coding techniques, allows speculative prefetching to begin even before knowing exactly which input will be needed. On a compute cloud testbed, we apply speculative pipelining to the Hadoop sorting benchmark and show that sorting time is shortened significantly.*

1. Introduction

The drive to provide large-scale computing and storage resources as well as lower data center operating costs has led to *cloud computing* infrastructures, where big compute jobs belonging to different users can be statistically multiplexed to maximize resource utilization (see, e.g., [1], [2]). As these services have come online, new programming models tailored to the compute cloud have evolved in tandem. Applications using MapReduce [3], a programming model for parallel data processing on large-scale commodity computing clusters, have formed a significant category on compute clouds like Amazon's Elastic Compute Cloud (EC2), where data center resources can be rented by the hour. For the defense and intelligence communities, compute cloud programming models like MapReduce may be a cost-effective way for processing the extremely high volume of sensor data streaming from the tactical edge [4].

In MapReduce, work is decomposed by the application developer into *map* and *reduce* tasks, which are executed in several sequential phases of parallel steps (Figure 1). While this is an elegantly simple way to perform processing of arbitrary big data, existing implementations of MapReduce are predicated on being hosted in private data centers (e.g., Google's), where a single administrative authority manages all resource needs and consumption of computing and network resources by unplanned competing applications can be minimized by fiat. As a result, some fundamental design choices and assumptions on the underlying resource pool in these implementations are a mismatch for public shared compute clouds (e.g., Amazon EC2), where the resource availability can be highly unpredictable due to varying competing loads.

First, in such cloud environments, resource sharing among different users is the norm. MapReduce implementations like Hadoop do not consider CPU or network load due to other, exogenous applications when scheduling tasks on worker nodes. This is a problem in environments like EC2 because a varying number of virtual machines (VMs), belonging to different users and encapsulating unknown workloads, may be assigned to the same physical node over time. Momentary spikes in CPU load from exogenous VMs could result in uncontrollable and unpredictable variations in the performance of MapReduce tasks sharing the same node. Across heterogeneous nodes, these effects may even be magnified (e.g., if the load spike occurs on a node with slower or already overloaded hardware). This can lead to *straggler* MapReduce tasks that can severely impede job progress [5]. Currently, Hadoop identifies stragglers with a simple heuristic and then starts a copy of the offending task on another node in a process called *speculative task execution*. Timely execution of such speculative tasks is critical to the completion time of MapReduce jobs.

Second, data center network topologies are commonly multi-rooted trees [6]. These topologies are intended to statistical multiplex traffic but have aggregate bandwidth that may not scale proportionally as the number of nodes increases. For instance, cross-rack network capacity between Top-of-Rack (TOR) switches, is usually lower than the bandwidth required for peak cross-rack traffic, i.e., such links are oversubscribed. In general, the bisection bandwidth of such topologies is limited by the bandwidth of core and aggregation switches [7], and the available bandwidth can vary, depending on competing traffic load. With inadequate network bandwidth, I/O-intensive parallel computations (e.g., some classes of jobs, such as sorting) are I/O-bottlenecked, rendering them sensitive to even slight variations in available network resources.

We seek to address these two issues. To mitigate problems due to varying and unbalanced resource availability in computation and I/O, we speculatively execute future tasks using otherwise idle resources. First, during compute-intensive phases of a MapReduce job, we utilize an idle network to perform *speculative prefetching* of input data, overlapping computation with I/O. Thus, in future pipeline stages, a task's input will be readily available as a result of the prefetching. This is in contrast to the present MapReduce approach of reactive data fetching [3], which is limited in that the fetching occurs at arbitrary points in time, causing an input delay for the associated computation. This delay can be substantial when network resources are unavailable or severely limited.

Second, we wish to minimize the amount of cross-rack traffic that is on the critical-path of a MapReduce computation. During I/O-intensive phases of a MapReduce job, we perform *speculative computing* to increase the likelihood that map outputs will be rack-local to subsequent reducers, thus decreasing cross-rack network traffic. In other words, we trade additional computation, performed on otherwise idle CPUs, for reduced cross-rack traffic. This optimization is orthogonal to speculative prefetching, which hides I/O time rather than reduces I/O.

The optimization principles we present here are neither limited to the Hadoop implementation nor specific to the MapReduce programming model and can be considered as general optimization primitives. Taken as a general speculative pipelining model, they can be systematically and automatically used when compiling programs written in higher-level languages, such as Pig Latin [8], Sawzall [9] and LINQ [10], down to their constituent distributed tasks.

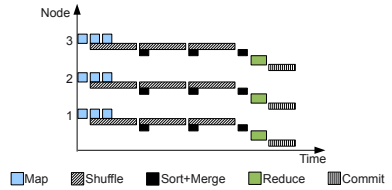


Figure 1: Hadoop job progress occurs over multiple nodes in sequential phases: map, shuffle, sort+merge, reduce, and commit.

2. MapReduce on Public Compute Clouds

Amazon EC2 is a public shared compute cloud, where users can dynamically scale the number of virtual machines (or "EC2 instances") according to their needs. Its infrastructure is organized into several availability zones (AZs), each corresponding to a geographically distinct data center. The Amazon Elastic MapReduce (EMR) service dynamically instantiates EC2 instances within an AZ to form a Hadoop cluster on-demand. Users can elect to start instances in a specific AZ, but have no control over whether these instances are physically located on the same machine or rack. Within the EC2 environment, we performed the following measurements to determine the extent to which CPU and network resources fluctuate.

First, we show the frequency and severity of stragglers by repeatedly running a compute-intensive Hadoop job and logging completion time per task on EMR clusters of 20 EC2 instances; the tasks are computationally identical, so an instance that takes longer to complete is easily identified as a straggler. We create 20 new EC2 instances at the beginning of each run and perform multiple runs in order to measure different subsets of physical nodes within the same data center, as EC2 instance placement is not directly under user control. Over 10 trials conducted during regular business hours, we discovered that 80% of tasks took an average of 12.25 minutes to complete. The remaining 20% were stragglers, taking an average of 18.42 minutes to complete – a 50% increase in task completion time. The relatively frequent appearance and severity of stragglers provides a clear case for our speculative prefetching strategy.

Second, we evaluated the likelihood of stragglers caused by network outages by repeatedly running simultaneous TCP flows between 10 sender-receiver pairs picked from a pool of 20 EC2 instances. As in the compute-intensive experiments, at the start of each run, we created 20 new EC2 instances. Each TCP flow was offered unlimited load and lasted for 20 minutes; we

logged the reception progress every 100ms. Thus, over 100 runs we ran a total of 1000 TCP flows, and out of those identified "straggling flows" as those where the average throughput over some 60-second window is smaller than a percentage α of that in the preceding 60-second window. We found that for $\alpha = 50\%$, there were 31 straggling flows, for $\alpha = 33\%$ there were 3, and for $\alpha = 25\%$ there was 1, spread across 29 runs, 3 runs and 1 run, respectively. These figures demonstrate a high likelihood of straggling events (e.g., 29 out of the 100 runs experienced a 50% or greater slowdown in one of their flows) which could be mitigated through our speculative techniques.

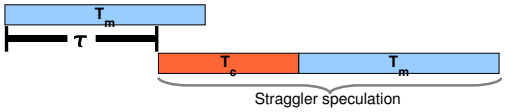
3. Speculative Pipelining Overview

Here, we introduce the notion of speculative pipelining and describe in detail how the speculative pipelining approach is useful in cloud implementations of MapReduce. In general, the goal is to use idle resources, noticed at present or anticipated in the future, to shorten the overall job completion time. We support the case with two canonical MapReduce scenarios.

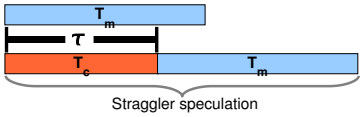
Speculative pipelining exploits the concept of overlapping I/O with computation. Consider a MapReduce job whose map phase is much longer than the shuffle phase, i.e., the job is compute-intensive. Such jobs might perform statistical analysis, natural language processing, or machine learning. Due to the processing requirements, however, these jobs are sensitive to slow nodes with overloaded CPUs; thus, speculative execution will likely be invoked to re-run any slow map tasks on different nodes.

Since the number of mapper tasks is usually many-fold larger than number of nodes, only a fraction of map tasks actively run at any given moment, while the rest are enqueued. Any time a new "wave" of map tasks starts executing, a mapper must fetch input from a remote node whenever the data does not happen to be on its the local disk, thereby incurring a delay.

A simple way to avoid the remote fetch delay consists of using the idle I/O capacity during one wave of map tasks to prefetch inputs for future map tasks. This way, the cost of fetching the inputs can be masked by nodes performing other useful compute tasks. Prefetching of this type can always shorten the job completion time with or without variations in resource availability. While Hadoop does not currently prefetch map inputs, the technique has been studied previously [11]. Here, we extend the notion of prefetching for the next wave of map tasks to speculative prefetching and computing.



(a) Map phase without speculative prefetch



(b) Map phase with speculative prefetch

Figure 2: A comparison of a map phase without (a) and with (b) speculative prefetch.

3.1. Scenario 1: Speculative Prefetching for Reducing Input Delay

Suppose that input data for the MapReduce task is not stored redundantly. This is a common practice when the MapReduce job is one of many in a chain of jobs, so that its input data is ephemeral and the previous job opts to store its output unreplicated for performance reasons. As a result, in recovering from a straggler, a speculative copy of a map task requires input from a remote file system node, incurring a fetch delay in addition to the re-execution time.

How could we avoid the fetch delay? Note that the network during the CPU-heavy map phase is idle. Thus, nodes can proactively create redundant copies of input blocks for map tasks which are likely to become stragglers, in anticipation of a speculative re-execution. We call this operation *speculative prefetching*. If a straggler node does appear, the re-executing node will more likely have the input already on its local disk, thus avoiding waiting for a fetch. However, should a straggler not materialize, speculative prefetching does not necessarily represent an additional time cost, provided that it overlaps with the node's computation.

Information obtained from profiling, scheduling information, virtual machine load characteristics and placement, etc. can be used to determine which nodes are likely to become stragglers. Knowing this, we can designate likely stragglers for speculative prefetching and unlikely ones as re-execution nodes, thereby improve the speculative prefetching hit rate.

3.1.1. Analysis. We demonstrate the advantage of speculative prefetching via a simple analysis. Suppose we have M map tasks across N compute nodes, and ϵ of these nodes are reserved for speculative task execution. Suppose we also have S straggler map tasks, which can only be identified as stragglers after

τ seconds (in Hadoop, $\tau = 60$ by default). We make the following assumptions:

A1. $M \leq N - \epsilon$, i.e., all map tasks are scheduled in one wave and started simultaneously.

A2. $S \leq \epsilon$, i.e., all stragglers are detected after τ seconds and scheduled simultaneously.

A3. Input size is uniform across map tasks.

Without speculative prefetching, the map phase completion time with S stragglers is $T_{standard} = \tau + T_c + T_m$, where T_c is the time needed to copy the input for the speculated tasks and T_m is the time for a normal, non-straggling map task to finish (Figure 2a). In contrast, with speculative prefetching, the map phase completion time is

$$T_{prefetch} = \begin{cases} \tau + T_m & \text{if } T_c \leq \tau \\ T_c + T_m & \text{if } T_c > \tau \end{cases}$$

as shown in Figure 2b. To completely overlap computation with I/O and to achieve full network utilization, we set $\tau = T_c$. Thus,

$$\begin{aligned} T_{standard} &= 2\tau + T_m \\ T_{prefetch} &= \tau + T_m. \end{aligned}$$

For small T_m, we can at most reduce the map completion time by 50%. For compute-intensive tasks, e.g., when $\tau \leq T_m \leq 2\tau$, the reduction is by 25% to 33%, a smaller but still substantial improvement.

3.1.2. Open-option Speculative Prefetching. Let us expand upon Scenario 1, in the context of a MapReduce system with i nodes, $n_1, n_2, \ldots, n_i$, hosted on a public shared compute cloud. Consider two map tasks: t_1, which operates on input data d_1, and t_2, which operates on d_2. Since it prefers data locality, the task scheduler assigns t_1 to n_1 because it stores d_1. For similar reasons, t_2 is assigned to n_2.

If n_1 and n_2 experience some slow-down (e.g., due to load external to MapReduce), t_1 and t_2 become candidate stragglers. Under our speculative prefetch strategy, we prefetch d_1 and d_2 to two other nodes, where t_1 and t_2 will be speculatively re-executed. Now, if n_2 recovers from the slow-down because its load subsides, there is only one true straggler, t_1. In this situation, prefetching d_2 was a waste of bandwidth.

Ideally, we would like to prefetch only d_1. However, we cannot determine what to prefetch because we do not know which task will turn out to be the true straggler. Even when t_1 emerges as the true straggler, since n_1 is a slow node, fetching d_1 from it may also be slowed. The problem is thus two-fold: (1) how do we ensure that we do not prefetch from a slow node; and (2) can we prefetch something that is always of benefit no matter which task is the true straggler?

Our solution, called *open-option speculative prefetching*, derives inspiration from erasure coding techniques used in RAID [12]. Briefly, if an erasure-coded disk array contains d raw data blocks and k coded blocks, then it can reconstruct the d raw data blocks from a subset of the $d+k$ blocks. For example, using Reed-Solomon codes [13], if any k of the $d+k$ blocks are lost, the remaining blocks can be used to reconstruct the d raw data blocks.

In our scenario, suppose in addition to having data blocks d_1 and d_2, we also had a coded block $c_{1,2} = d_1 \oplus d_2$ on node n_3. Since n_2 turns out not to be a slow node, n_3 can fetch d_2 quickly. With d_2, n_3 can decode $c_{1,2}$ to yield d_1, allowing it to run t_1. This property is symmetric: had t_2 been the true straggler instead, $c_{1,2}$ could be decoded with d_1 to obtain d_2, allowing t_2 to run on n_3. The coded block $c_{1,2}$ on n_3 has left the option open for n_3 to speculatively execute either t_1 or t_2 with reduced prefetching overhead. Hence, we term this open-option speculative prefetching.

A simple alternative is to create replicas of each data block, as is done in the Hadoop Distributed File System (HDFS). This is reasonable if we are only concerned with a few simultaneous stragglers. However, to guard against k simultaneous node slow-downs, we would need to maintain k replicas per block, totaling $d(k+1)$ blocks. Clearly, this is impractical as k increases. In contrast, open-option speculative prefetching can protect against k slow-downs while storing just $d+k$ blocks using Reed-Solomon codes. Note, however, that there is a trade-off between the amount of data that needs to be fetched to decode and the storage overhead of the encoding. For instance, Reed-Solomon encodings require relatively little storage ($d+k$ blocks), but require d blocks to be fetched to decode. We will explore efficient codes that strike a balance in future work.

3.2. Scenario 2: Speculative Computing for Reducing Cross-Rack Traffic

Suppose that a MapReduce input data set is stored using triply redundant blocks—a common default value in Hadoop installations. Furthermore, assume that the MapReduce job is large enough to run on nodes located in multiple racks. During the shuffle phase, such groups of nodes will need to communicate across shared links between Top-of-Rack switches, causing them to act as bottlenecks.

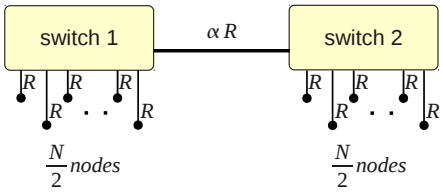


Figure 3: A two-rack network topology from Scenario 2. Two groups of $N/2$ nodes each connect to a non-blocking switch via rate R links. The inter-switch link runs at a different rate, αR, where α is a constant typically greater than 1.

The shuffle phase can be sped up by eliminating some traffic across the bottleneck link as follows. Suppose that a reducer r requests intermediate data from some mapper m in a remote rack. We assume that this could not be avoided by running the reducer in the remote rack, as r is likely to request data from multiple racks. To eliminate this cross-rack traffic, we can take advantage of any file-system replicas of m's input in r's rack and run a speculative redundant mapper m' there. After the map phase finishes, r will be able to fetch its input from m' in its local rack, while leaving the output of m intact.

3.2.1. Analysis. We conduct a simple analysis of the job completion time improvement due to redundant mappers. Consider a topology where N nodes, each assigned one map and one reduce task, are split into two $N/2$-node racks. As shown in Figure 3, suppose that the $N/2$ nodes within each rack are attached to a non-blocking rack switch, to ports operating at rate R. Further, suppose that the two switches are connected by a single link operating at rate αR, where $\alpha \geq 1$ lets us model scenarios where links between Top-of-Rack switches are faster than local ones (e.g., using 10 Gb Ethernet interconnects). Lastly, for the job's data flow pattern, suppose that the output of each mapper has the same size, and is partitioned uniformly such that an equal size slice is assigned to each reducer. Such a balanced pattern could occur, for example, in sorting uniformly distributed keys, or where sampling is used to create balanced partitions.

We make the following simplifying assumptions for an I/O-intensive scenario:

A1. Map-Reduce CPU demands are negligible compared to I/O.
A2. All tasks start at the same time.
A3. Flows fairly share network links without incurring any overhead.
A4. Flows from mappers to reducers on the same node pass through that node's network link, sharing it with any external flows.

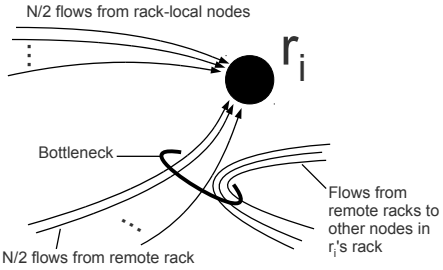


Figure 4: Diagram accompanying the analysis for Scenario 2. The flows shown correspond to the baseline case without speculative redundant computation.

Let us consider just one of N reducers, r_i. As shown in Figure 4, there are two classes of flows going to r_i: the rack-local ones, coming from $N/2$ nodes on the same rack, and remote ones, coming from $N/2$ nodes on the remote rack. Of these, the remote flows are constrained by the bottleneck link between racks; given that $(N/2)^2$ flows pass through this link, they will share the link at bottleneck rate $R_B = \alpha R/(N/2)^2$. We will focus on the case where R_B is not larger than the fair sharing rate of a rate R link among N flows; that is, $R_B \leq R/N$, which is satisfied for $N \geq 4\alpha$. In this case, the total completion time will be equal to the completion of any of the remote flows.

In computing the completion time, consider that given a total data size D, and given our balanced data flow pattern, reducer r_i receives a D/N amount of data. Since this data consists of equal-sized outputs of N mappers, we see that each mapper contributes a D/N^2 amount of data to reducer r_i. The flows run in parallel, so to find the overall completion time $T_{BASELINE}$, we just find the completion time for a single remote mapper-to-reducer transfer:

$$T_{BASELINE} = \frac{D/N^2}{\alpha R/\left(N/2\right)^2} = \frac{D}{4\alpha R}.$$

Let us next rearrange the remote flows by creating X, where $1 \leq X < N/2$, redundant mappers on r_i's rack, leaving $N/2 - X$ mappers outside. Now, there will be a total of $(N/2)(N/2 - X)$ flows crossing the bottleneck cross-rack link, leading to the bottleneck rate of

$$R_B = \frac{\alpha R}{N/2\left(N/2 - X\right)}$$

Again the remote flows will dominate the completion time as long as they take up no more than the fair share of r_i's link, that is, $R_B \leq R/N$. Such a

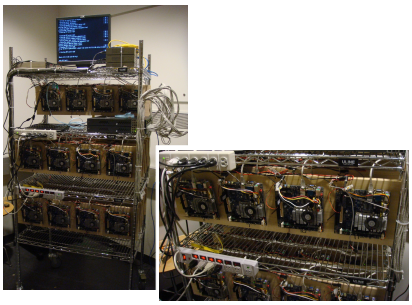


Figure 5: Our Hadoop cluster testbed of 46 nodes.

condition holds if

$$X \leq \frac{N}{2} - 2\alpha.$$

The completion time based on R_B and D/N^2 units of data delivered per flow is now:

$$T_{REDUNDANT} = \frac{D\,(N/2 - X)}{2\alpha NR}.$$

We can see that the completion time is smallest for $X = N/2 - 2\alpha$, where it becomes a factor of $4\alpha/N$ smaller than the baseline. The order of N-fold speedup may be counter-intuitive at first sight but it can be easily understood by recognizing that cross-rack traffic is proportional to $(N/2) * (N/2 - X)$. Thus the traffic decreases from $(N/2)^2$ when $X = 0$ to αN when $X = N/2 - 2\alpha$. Furthermore, note that when the cross-rack's bottleneck becomes severe, i.e., when α decreases in value, the speedup increases as $N/4\alpha$, as expected.

4. Experiments with Speculative Pipelining Applied to Sort

We built a 46-node cluster of commodity machines powered by VIA VB8001 motherboards and VIA Nano 1.6GHz CPUs. Each board is connected to a switch via a 100Mbps Ethernet link. Lastly, each board is equipped with 1GB of RAM and 8GB of flash storage. Figure 5 shows photos of the cluster.

We used either one or two switches to configure a one- or two-rack system, respectively. By using two switches we can create cross-rack bottleneck links typical in large data centers (see Figure 3). Each switch was a Cisco Catalyst 3500 Series XL non-blocking 48-port switch, with all ports set to a fixed, 100 Mbps bit rate. Where we used two switches, the switches were bridged via a 100 Mbps Ethernet link.

The nodes ran Hadoop v0.20.2. One node in the rack is designated the master and runs both the Hadoop and HDFS master daemons, but does not execute any map or reduce tasks or store any HDFS data blocks. All other nodes run both Hadoop and HDFS slave daemons and are responsible for running computation tasks as well as storing HDFS data blocks. This arrangement is consistent with that of Amazon EMR clusters.

In the experiments below, we use the existing MapReduce sort implementation found in the Hadoop code base, and previously used as a benchmark [14]. We apply speculative pipelining principles to improve its performance in Scenarios 1 and 2 above.

4.1. Experiments on Speculative Prefetching

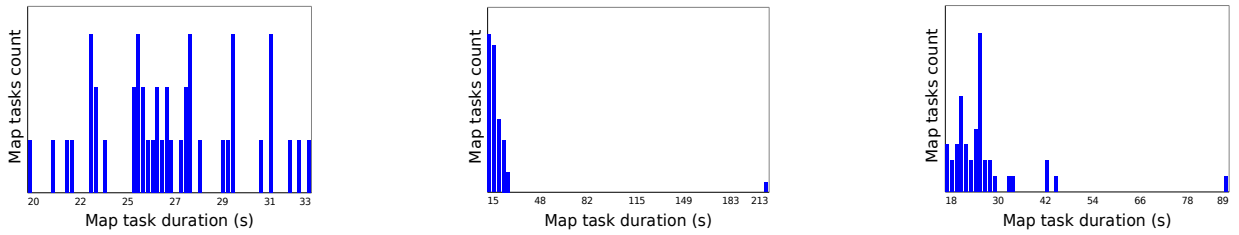
We first consider Scenario 1, where an unexpected exogenous CPU load spike causes one node in our rack to slow down and where speculative prefetching can improve map phase completion time.

We configured the cluster to be a single rack with one master and 44 worker nodes. The sort application is asked to sort 5.6GB of random binary data stored on HDFS. We modified the default block size in HDFS to 128MB so that each node has exactly one 128MB data block stored locally. We set the HDFS replication factor to 1, forcing Hadoop to assign a single data-local map task to each node, i.e., each map task's entire input split is on its local disk.

Before starting the sort job, we created an artificial CPU workload on node n_1 such that the Hadoop processes on this node would receive only $\sim$1% of CPU time. As a result, Hadoop triggers speculative execution of task t_1, originally assigned to n_1, on some node n_x. Since n_x does not have a local replica of t_1's input split, it is forced to reactively fetch from n_1. Because n_1 is heavily loaded, this fetch is slow.

First, we demonstrate map phase behavior when no speculative prefetching is done. Figure 6a shows the map task completion time distribution in a control experiment where no exogenous CPU load was introduced on n_1. The spread of map completion times is fairly narrow (22s to 33s). In contrast, when we introduce exogenous CPU workload to n_1 and t_1 is speculated to n_x, t_1's completion time inflates to 213s. This is due to Hadoop's 60s straggler detection threshold and the time needed to fetch the input split from n_1 to n_x. Note that the longest map task bounds the duration of the entire map phase, meaning this single straggling task impedes the entire job.

Next, we emulate speculative prefetching by increasing the replication factor in HDFS such that any node chosen to speculatively re-execute t_1 will have a local copy of its input split. This is an indirect measurement method but accurately reflects our scenario, where the



(a) No exogenous CPU load (b) Without Speculative Prefetching (c) With Speculative Prefetching

Figure 6: Map tasks duration distributions. The total number of map tasks is 44 in all the plots. (a) Baseline case, with no exogenous CPU load on node n_1. (b) With exogenous CPU load on n_1, but without speculative prefetching. (c) With exogenous CPU load on n_1 and with speculative prefetching, the overall map phase completion time is reduced from 213s to 89s.

speculative prefetch time is completely masked by the map computation, and is thus equivalent to having a local replica of t_1's input split available at n_x at the time of task speculation. Figure 6c shows the map task completion time distribution with speculative prefetching enabled. Note that t_1 now only takes 89s (60s straggler detection time + 29s map computation time) to finish, clearly demonstrating the advantage of speculative prefetching.

4.2. Experiments on Speculative Computing

As discussed earlier in Section 3.2, we can alleviate some of the cross-rack network load by performing extra computation on rack-local replicas of the input data. For example, in the sorting application, as shown in Figure 7, mappers send their output to reducers in both racks. This involves cross-rack flows, as depicted in Figure 8a. However, suppose that the input data has two replicas per block, one on each rack. For such blocks, instead of running one mapper, we can run one mapper per replica, making the results available on both racks without transferring them across the inter-rack link. Figure 8b shows how redundant mappers can avoid sending data between racks.

We demonstrate the potential performance gain due to redundant mappers by running the unmodified Hadoop Sort application with parameters tuned to emulate a real implementation of speculative redundant computing as follows. First, we divide our testbed into two 22-node worker groups, each connected to a separate non-blocking switch. The switches are bridged with one 100Mbps link. An additional node, serving as the master, is also connected to one of the switches. We create input data such that each worker holds a 512MB slice, saved as four 128MB blocks. Then, we run Sort in two configurations:

- **Configuration A: Baseline.** Using all 44 workers; this results in a standard MapReduce sorting run on a 22GB data set.
- **Configuration B: Emulated Speculative Computation.** Using only one 22-worker group, sorting a 11GB data set.

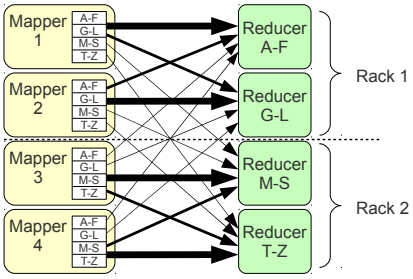


Figure 7: Data-flow diagram for a MapReduce sort run. Thickest arrows represent the fastest, node-local flows where no network is used. Medium-thick arrows represent fast, rack-local flows. Thin arrows represent slowest, cross-rack flows.

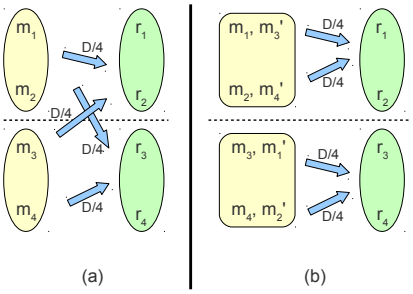


Figure 8: Map task assignment with (a) no redundant mappers, and (b) redundant mappers. Redundant mapper of primary mapper m_i is indicated as m_i'. The dotted line represents the rack boundary. In both cases the same total amount of data, D, is moved to reducers. Note that in (b), no flows cross the rack boundary.

We argue that the traffic pattern of the shuffle phase of Configuration B is equivalent to that of one rack

	Configuration A	Configuration B
Map phase	185 s	215 s
Shuffle phase	597 s	199 s
Sort+Merge phase	1 s	1 s
Overall	646 s	254 s

Table 1: Durations of map, shuffle and reduce phases in Sort. We omit the commit phase from measurements. Note that the overall duration is not a sum of individual phase durations, since the phases overlap.

of Configuration A, where every mapper is run redundantly, since the total amount of data transferred from mappers to reducers in one rack is the same, and there is no cross-rack traffic in either case. In other words, Configuration B emulates the top half of Figure 8b. The major difference is that the redundant mappers shown in Figure 8b do not get run in Configuration B; however, this does not affect the shuffle phase duration.

We show the running times of the two configurations in Table 1, broken down by MapReduce phase. Also, we present the progress of sort jobs of both configurations in Figure 9. Clearly, Configuration B gained significant speedup from reduced cross-rack traffic.

5. Related Work

Several related approaches to optimization in compute cloud programming have recently been proposed in the context of MapReduce/Hadoop. Zaharia et al. [5] tackled deficiencies in Hadoop's speculative task scheduler in heterogeneous environments such as compute clouds, where resource availability is dependent on exogenous factors. A major weakness in Hadoop is that its straggler detection mechanism permits false stragglers, causing excessive task speculation and taking away resources from regular tasks. To remedy this, the authors propose a speculative task scheduling mechanism called Longest Approximate Time to End (LATE), in which the task that is predicted to finish last is speculated first, since this will give the best odds of the speculated task overtaking the original. LATE estimates completion time by tracking task *progress rate* instead of percentage of work completed. LATE is complementary to our proposed optimizations; in fact, we can use the progress rate tracked by LATE to inform what to speculative prefetch (e.g., input splits of slow tasks) and where to place the prefetched data (e.g., nodes that consistently make good progress or that have or are about have free CPU cycles).

Seo et al. [11] present related ideas on prefetching in MapReduce. In their High Performance MapReduce (HPMR) implementation, prefetching occurs on two levels. First, while a task works on its current input key-value pair, the idle network can be used to prefetch the next key-value pairs. Second, during computation of its current task, a node can simultaneously prefetch input data for the next tasks that are in queue. While the authors also employ pipelining to overlap computation and I/O, a key difference with our work is that we use the opportunity to do *speculative* prefetching.

Condie et al. [15] propose changes to MapReduce's batch-oriented data processing in order to service data streams. Their system, Hadoop Online (HOP), sets up producer-consumer relationships between tasks, pushing results from producers to consumers instead of having consumers pull results. While a push model naturally leads to task pipelining, it must fall back to a pull model when multiple waves of map tasks must be scheduled before reduce tasks (i.e., when no consumers are alive to receive the pushed results). In contrast, we attempt to overlap computation with network I/O only when resources permit. Furthermore, whereas HOP is designed to recover from hard faults, we have proposed a more general method that handles lagging tasks caused by CPU and network load fluctuations.

6. Conclusions

In this paper, we address a frequently occurring phenomenon in public shared compute clouds, as we observed on Amazon EC2: unpredictable variations in resource availability for MapReduce jobs. Furthermore, these resources may be unbalanced in their computation and I/O capacities. Combined, these can lead to CPU and network underutilization.

We have presented an approach, called speculative pipelining, which can mitigate the impact of uncertain or insufficient resource availability on MapReduce job completion time. We have described two specific schemes: (1) speculative prefetching to reduce input delay when recovering from straggler tasks, and (2) speculative computing to reduce traffic traversing the cross-rack network bottleneck. We have shown the effectiveness of these schemes through a sorting benchmark run on a lab testbed. Furthermore, we have shown performance analyses and introduced novel ideas such as open-option speculative prefetching.

Our speculative approach differs from current practices found in MapReduce implementations, where input fetching takes place only when input is needed. With speculative pipelining, we prefetch input to reduce future input delay, and perform redundant computation to reduce future network congestion. These

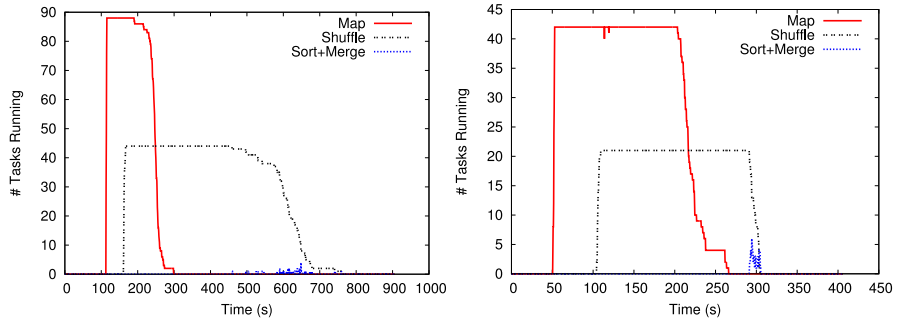


Figure 9: Progress of sort jobs on Configuration A (left) and B (right). The vertical axis shows the number of running tasks belonging to one of the MapReduce phases.

speculative operations put otherwise unused CPU and network resources to good use and, in principle, can be extended to compute cloud programming models beyond MapReduce.

Acknowledgements

This material is based on research sponsored by Air Force Research Laboratory under agreement numbers FA8750-09-2-0180 and FA8750-10-2-0180. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory or the U.S. Government.

References

[1] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel, "The cost of a cloud: research problems in data center networks," *SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 1, pp. 68–73, 2009.

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the clouds: A berkeley view of cloud computing," UC-Berkeley, Tech. Rep. UCB/EECS-2009-28, Feb 2009.

[3] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," in *OSDI*, 2004.

[4] D. Meiron, S. Cazares, P. Dimotakis, F. Dyson, D. Eardley, S. Keller-McNulty, D. Long, F. Perkins, W. Press, R. Schwitters, C. Stubbs, J. Tonry, and P. Weinberger, "Data analysis challenges," JASON, Tech. Rep. JSR-08-142, Dec 2008.

[5] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *OSDI*, 2008.

[6] M. A. Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *SIGCOMM*, 2008.

[7] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "Vl2: a scalable and flexible data center network," *SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 4, pp. 51–62, 2009.

[8] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: a not-so-foreign language for data processing," in *SIGMOD*, 2008.

[9] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan, "Interpreting the data: Parallel analysis with sawzall," *Sci. Program.*, vol. 13, no. 4, pp. 277–298, Oct 2005.

[10] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey, "Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language," in *OSDI*, 2008.

[11] 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 *IEEE CLUSTER*, 2009.

[12] G. A. Gibson, L. Hellerstein, R. M. Karp, and D. A. Patterson, "Failure correction techniques for large disk arrays," *SIGARCH Comput. Archit. News*, vol. 17, no. 2, 1989.

[13] J. S. Plank, "A tutorial on reed-solomon coding for fault-tolerance in raid-like systems," *Software: Practice and Experience*, vol. 27, no. 9, pp. 995–1012, 1997.

[14] O. O'Malley, "Terabyte sort on apache hadoop," http://sortbenchmark.org/YahooHadoop.pdf.

[15] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "Mapreduce online," UC-Berkeley, Tech. Rep. UCB/EECS-2009-136, Oct 2009.