# TR-Spark: Transient Computing for Big Data Analytics

Ying Yan, Yanjie Gao, Yang Chen, Zhongxin Guo[1], Bole Chen[2], Thomas Moscibroda

Microsoft Research

{ying.yan,yanjga,yachen,moscitho}@microsoft.com, guozhongxin@bupt.edu.cn[1], bolec@andrew.cmu.edu[2]

## Abstract

Large-scale public cloud providers invest billions of dollars into their cloud infrastructure and operate hundreds of thousands of servers across the globe. For various reasons, much of this provisioned server capacity runs at low average utilization, and there is tremendous competitive pressure to increase utilization. Conceptually, the way to increase utilization is clear: Run time-insensitive batch-job workloads as secondary background tasks whenever server capacity is underutilized; and evict these workloads when the server's primary task requires more resources. Big data analytic tasks would seem to be an ideal fit to run opportunistically on such *transient resources* in the cloud. In reality, however, modern distributed data processing systems such as MapReduce or Spark are designed to run as the primary task on dedicated hardware, and they perform badly on transiently available resources because of the excessive cost of cascading re-computations in case of evictions.

In this paper, we propose a new framework for big data analytics on transient resources. Specifically, we design and implement TR-Spark, a version of Spark that can run highly efficiently as a secondary background task on transient (evictable) resources. The design of TR-Spark is based on two principles: resource stability and data size reduction-aware scheduling and lineage-aware checkpointing. The combination of these principles allows TR-Spark to naturally adapt to the stability characteristics of the underlying compute infrastructure. Evaluation results show that while regular Spark effectively fails to finish a job in clusters of even moderate instability, TR-Spark performs nearly as well as Spark running on stable resources.

***Categories and Subject Descriptors*** B.8.1 [*Reliability and Fault-Tolerance*]:

*SoCC '16*, October 05-07, 2016, Santa Clara, CA, USA.
© 2016 ACM. ISBN 978-1-4503-4525-5/16/10...$15.00.
DOI: http://dx.doi.org/10.1145/2987550.2987576

***Categories and Subject Descriptors*** Performance, Algorithms

***Keywords*** Transient computing, Spark, Checkpointing

## 1. Introduction

The amount of money large-scale public cloud providers like Amazon, Microsoft, or Google invest into their cloud infrastructure is mind-boggling. Microsoft, for example, publicly reported to operate a fleet of more than a million servers as early as 2013 for its worldwide cloud operations, and it is known that it has since expanded its cloud operations dramatically. New data centers are being built and expanded across the globe, with no end in sight to this growth. Given these extraordinary investments, it is worthwhile to note that even with the latest state-of-the-art cluster management and scheduling techniques (e.g. [16, 24, 25, 29, 32, 34]) the average resource utilization of servers in data centers is very low. To say it pointedly, at any moment in time, the equivalent of entire data centers of computing resources are wasted. The reasons for this low resource utilization are not accidental, to a large extent they are fundamental: Some capacity is required as buffers to handle the consequences of failures; natural demand fluctuation causes capacity to be unused at certain times; servers are over-provisioned to handle load-spikes; fragmentation at the node and cluster level prevents all machines to be fully utilized; churn induces empty capacity; and so forth.

Given the large infrastructure investments and fierce competitive pressure, it is only natural that companies are urgently trying to increase utilization by running delay-insensitive or non-customer facing workloads using these temporarily spare capacities. Ideal candidate workload for this purpose would be big data analytics and other type of batch workloads (e.g. machine learning training). Using such workloads to fill up unused capacity would be particularly desirable given the growing volume and importance of big data analytic workloads in cloud computing companies across the industry. The total volume of processed data, the number of executed jobs per day and the number of machines used for this purpose are all dramatically increasing at rapid speed. The amount of money Microsoft and other companies invest into computing and analyzing big data for their daily and weekly production pipelines is staggering.

The inherent challenge with running any workload on temporarily spare cloud resources is that these resources are fundamentally instable, or *transient*: Nodes kept empty as a resource buffer may suddenly be utilized in case of a failure; spare capacity provisioned for load-spikes vanishes whenever a spike occurs; etc. For the same reason, cloud providers started to offer such transiently available resources (typically at a lower price) to both internal and external customers, e.g. Amazon EC2 spot instances [3], Google preemptible instances[5], or Azure Batch [4]. As mentioned, big data analytics jobs would be an ideal fit to run on such "transient resources" since these jobs are expensive, but not latency critical. Thus, concretely we would like to run big data processing systems such as MapReduce, Spark[7], or Scope[12] on transient resources. Logically, the big data processing system could run as a lower-priority task in the compute cluster and use the available resources whenever they are available. While conceptually being a simple and natural idea, the opportunity of *big data analytics on transient resources* is tremendous: large-scale cloud companies could save \$100s millions dollars if the approach was successfully put into practice.

Unfortunately, things are not as simple. The key technical challenge is that existing big data processing systems (including MapReduce, Spark, or Scope) *perform exceedingly badly on transient resources*. These systems are designed to tolerate failures, but they run stably and efficiently only if such failures are very rare events. When computing on transient resources in the cloud, however, the event of a computing resource becoming unavailable is *not* a very rare event; instead it is fundamentally and by design a rather common occurrence. The reason that all existing data processing systems are unable to cope with transient resources is that they have been designed to run on dedicated resources (typically, companies have dedicated MapReduce clusters, Scope clusters, Spark clusters, etc.), and they do not run efficiently on transiently available resources due to the exceedingly high cost of cascading re-computations caused by failures. Again, if such failures are very rare, the cost of re-computation is acceptable, but—as we show in this paper—even at small degrees of resource instability, existing data processing systems either take an excessive amount of time to complete a job, or even fail to complete the job entirely.

In this work, we design and implement TR-Spark, a version of Spark that can execute highly-efficiently as a secondary background task on transient (evictable) resources. Our motivation to build TR-Spark follows the outline above: We intend to run jobs from Microsoft's big data analytic Spark pipeline as background tasks on nodes that are temporarily not fully utilized for their primary task (e.g. hosting Azure services or Bing index serve). In order to address the aforementioned problem of excessive re-computation, TR-Spark is based on the following two principles: resource-stability and data-size reduction-aware scheduling (*TR-Scheduling*) and lineage-aware checkpointing (*TR-Checkpointing*). Our solution is based on the following intuitions obtained from intensive job trace analysis and system profiling in Microsoft's cloud environments. First, a jobs' re-computation cost can be reduced through backing up intermediate results. However, such check-pointing decisions (what, when and to where to do the backup) should be made according to the level of resources instability, current re-computation cost and data lineage. Secondly, the number of re-computations can be significantly reduced if the task that outputs the least amount of data is prioritized during task scheduling (task to resource assignment). The importance of such data-size reduction-aware scheduling is that in this way, the downstream stage maintains less output data waiting for backup.

Our contributions are summarized as follows:

- We identify the performance problem of existing big data analytic systems when run on transient resources and propose two key principles to address the performance problems.
- We propose optimized resource-stability and data-size reduction-aware scheduling (*TR-Scheduling*) and lineage-aware checkpointing (*TR-Checkpointing*) strategies to enable big data analytic systems to run efficiently on transient resources.
- We design and implement ***TR-Spark***, a version of Spark which can efficiently run on transient resources based on the above two principles. Experimental results show that TR-Spark scales near-optimally on transient resources with different instabilities. We deploy and test TR-Spark on Azure Batch.

## 2. TR-Spark

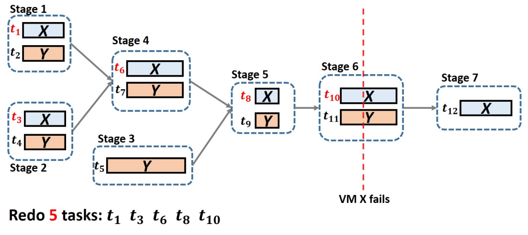
### 2.1 Background on Spark

Spark is an open source distributed big data processing engine [7]. Its performance, ease of programming and deployment, and rich set of high-level tools make it attractive to an increasing number of enterprise users and contributors. All major cloud providers offer managed Spark services to ease its usage.

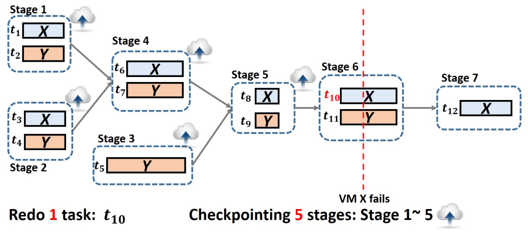
Spark uses RDD (Resilient Distributed Datasets)[38] to abstract and manage distributed data sets. Each data record in an RDD is divided into partitions and computed on different nodes. An RDD is an immutable distributed dataset which avoids the complexity of concurrent consistency. RDD combines data flow with map reduce type primitives to support high level APIs. Spark parses and translates a user submitted job into a DAG of stages to execute on distributed environment. The execution model likens BSP (Bulk synchronous parallel) which makes Spark general and easy to use. Spark schedules the tasks in different stages based on

FIFO or FAIR strategies [8]. It assigns tasks to machines based on data locality using delay scheduling [37]. Spark's fault tolerance is achieved through logging the RDD's operations and dependencies to recompute the partition when a failure occurs. Spark's checkpointing unit is also an RDD. The problem is that in the context of running on transient resource, Spark's checkpointing has multiple several limitations: (1) Coarse grained checkpointing – Spark must checkpoint all data within an RDD in each checkpointing action. (2) High level Spark APIs like SQL do not support checkpointing. (3) Programming Complexity – the checkpoint has to be written by the developer in the application; it cannot be adaptive to a dynamic unstable environment. Also, the developer needs to take care of what is to be checkpointed. When running Spark on unstable resources, these limitations make it hard in practice for Spark to be stable and efficient.

## 2.2 TR-Spark: Motivation



(a) No checkpointing



(b) Stage level checkpointing (Always Checkpointing)

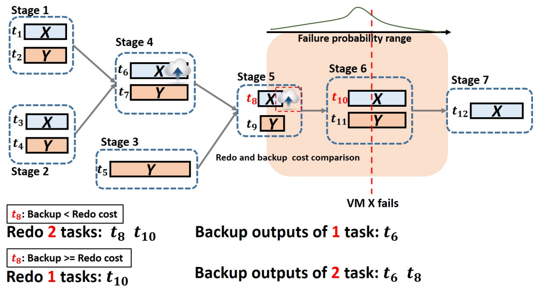

Failure probability range
Stage 1
Stage 2
Stage 3
Stage 4
Stage 5
Stage 6
Stage 7
Redo and backup cost comparison
VM X fails
$t_8$: Backup < Redo cost
Redo 2 tasks: $t_8$ $t_{10}$
Backup outputs of 1 task: $t_6$
$t_8$: Backup >= Redo cost
Redo 1 tasks: $t_{10}$
Backup outputs of 2 task: $t_6$ $t_8$
(c) TR-Checkpointing: Based on transient resource stability (our solution)

**Figure 1.** Different checkpointing strategies

TR-Spark provides task level self-adaptive checkpointing (TR-Checkpointing) and scheduling (TR-Scheduling) mechanisms for reducing the number of re-computations caused by the resource instability, and therefore ensures stability and efficiency.

Figure 1 illustrates the basic idea of TR-Checkpointing compared to existing solutions. Figure 1(a) shows the original Spark, without periodic in-job check-pointing: once the VM fails (VM $X$ in the example)[1], the current task running on the VM (task $t_{10}$) and all its dependencies ($t_6$, $t_1$ and $t_3$) need to be re-computated. As resources become less stable, original Spark becomes so inefficient that the job even fails to complete. The simplest possible solution to reduce re-computation cost is to do periodic checkpointing. For example, we can do checkpointing at every stage as shown in Figure 1(b) (through manually partitioning the job into sub-jobs based on the stage and calling the checkpointing API of Spark). Since this has to be done by the developer in the application at compile time, the checkpoint interval cannot be adaptive to the actual resource availability condition in the cluster. Checkpointing everything is the most conservative choice to guarantee the fewest number of re-computations. In this example, when a failure occurs, only one task $t_{10}$ needs to be recomputed. However, all the 5 stages which $t_{10}$ depends on are checkpointed, which is also expensive, since all data needs to be stored to disk, requiring substantial communication and I/O. Thus, while original Spark has excessive recomputation cost on transient resources, always checkpointing has excessive checkpointing cost. Therefore, neither solution achieves an acceptable performance when run as a background task on a compute cluster in practice.

An example of the checkpointing strategy in TR-Spark is shown in 1(c). With a resource instability distribution estimation, the system can smartly compute an VM failure probability range (in the example, TR-Spark decides to checkpoint only the outputs of $t_6$ and $t_8$) as well as the expected cascading re-computation and checkpointing cost. Then, TR-Spark takes the decision of whether or not to checkpoint to achieve an optimal trade off between the number of re-computations and checkpoints. This is done in a way that is transparent to the applications. Regular Spark jobs can thus be made to run on transient resources efficiently.

TR-Spark provides a fine grained checkpointing policy - *task level checkpointing*. The granularity of the data object that TR-Spark checkpoints is the result of individual tasks. An example is shown in Figure 2. Each task, e.g. Task 1, outputs a list of data blocks identified with a *ShuffleBlockID*. The number of data blocks equals the number of reducers. During the checkpointing process, the backup of a task's output is successful if and only if all the data blocks in the current task's output list are finished. In this paper, we use 'data block' as a simplification of a task's output data block list when discussing our checkpionting strategy. Compared to the checkpointing API of Spark which checkpoints the RDD of the entire stage, TR-Spark checkpoints only the subset of the tasks of a stage that need checkpointing. A comparison is

[1] We use the terminology VM because we design TR-Spark specifically for Azure Batch, where the unit of computation are VMs. But TR-Spark can also run on other compute resource units, such as nodes, servers, machines, containers, etc...

made in Figure 2, suppose VM 1 is not stable, then Task 1's output data blocks on VM 1 should be checkpointed. However, using the stage level checkpointing strategy (e.g[30]), all the tasks in Task 1's current stage will be checkpointed although VMs 2 through *m* are all sufficiently stable. Usually, in big data analytics applications, the number of VMs or nodes *m* is very big. Therefore, stage level checkpointing incurs excessive checkpointing overhead. In contrast, TR-Spark only checkpoints the result of Task 1. The tasks in one stage are running on different VMs with different stabilities, thus finer grained checkpointing offers more flexibility to achieve the best checkpointing plan. To further reduce the checkpointing cost, TR-Spark also co-optimizes the scheduling policy as we discuss in Section 3.1.

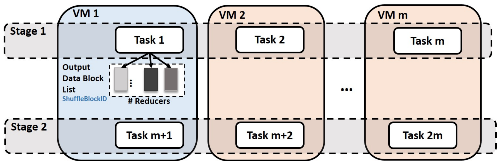


**Figure 2.** Checkpointing granularity: stage v.s. task level

### 2.3 TR-Spark: Design Overview

The system architecture of TR-Spark is shown in Figure 3. We implement TR-Spark by modifying Spark's *Task Scheduler* and *Shuffle Manager*, and introduce two new modules *Checkpointing Scheduler* and *Checkpoint Manager*. Details of the implementation are given in Section 5.

**Task Scheduler** implements our TR-Scheduling algorithm for task scheduling (Section 3.1).

**Checkpointing Scheduler** implements our TR-Checkpointing algorithm (Section 3.2). It first collects the necessary resource instability information, computes the best checkpointing plan, and then sends the checkpointing plan as a triple <ShuffleID, TaskID, Backup plan> to the Checkpoint manger on each worker. Finally, it pushes the information into a backup stack. We use a stack here for the reason that when the stack is not empty, it is always more efficient to backup the top data blocks which are newly generated. In this way, if the tasks in the next stage finish the backup, the current stage's tasks do not need to be backed up any more. Through prioritizing the backup order we may save backup cost as much as possible. After receiving the checkpointing status from the Checkpoint Manager, the Checkpointing Scheduler updates the new data location in the mapstatus table.

**Checkpoint Manager** is responsible for maintaining the backup plan stack and executing the backup operation. It also periodically sends back status information to the Checkpointing Scheduler in the master.

**Shuffle Manager** supports checkpointing data to remote reliable storage.

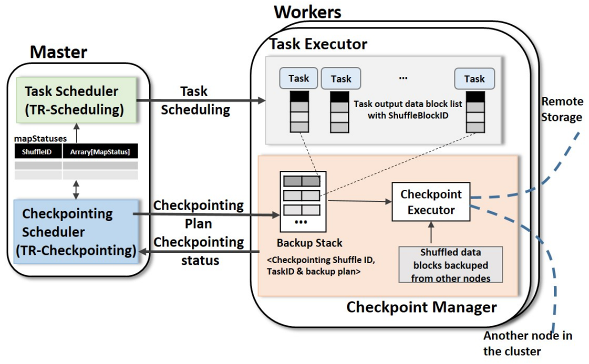

Workers
Master
Task Executor
Task Scheduler (TR-Scheduling)
Task Scheduling
Task
Task
...
Task
Task output data block list with ShuffleBlockID
Remote Storage
mapStatuses
ShuffleID
Arrary[MapStatus]
Checkpointing Scheduler (TR-Checkpointing)
Checkpointing Plan
Checkpointing status
Checkpoint Executor
Backup Stack
<Checkpointing Shuffle ID, TaskID & backup plan>
Shuffled data blocks backuped from other nodes
Checkpoint Manager
Another node in the cluster

**Figure 3.** TR-Spark Architecture

## 3. Transient Computing

At the heart of TR-Spark are the novel resource-stability and data-size reduction-aware scheduling (*TR-Scheduling*) policy and the lineage-aware checkpointing (*TR-Checkpointing*) strategy. We introduce these paradigms in this section.

### 3.1 TR-Scheduling

The task scheduler's job is to assign a task from the pending list whenever there is a resource available (i.e., a free core). The task selection is done through scanning the non-blocking tasks (all their dependencies are finished) ordered according to a certain priority rule (e.g. FIFO or Fair in original Spark [8]). In order to reduce re-computation cost in the face of unstable resources, we should give prioritize tasks that output the least amount of data for scheduling. In this way, the downstream stages contain less output data, which makes back-up and checkpointing relatively cheaper. Based on this intuition, we identify two important factors that govern scheduling decisions when computing on transient resources:

**Task's output and input data sizes**: A task which generates less output data than it takes in as input, effectively reduces the size of data for checkpointing. Such tasks should have higher priority for scheduling.

**Task's execution time**: A task with shorter execution time has less risk of being impacted by a failure, and thus has higher probability of reducing the number of data blocks waiting to be processed.

*TR-Scheduling* co-considers the above two factors and combines them with Spark's other existing scheduling factors. Specifically, as illustrated in Algorithm 1, for an available resource on VM $v$, TR-Scheduling prioritizes the tasks according to:

- The task whose success probability on that VM exceeds a probability threshold $\gamma$. When the VM's lifetime can be accurately obtained (e.g. Amazon spot block), $\gamma$ is set to 1.
- The task with the biggest data size reduction rate is selected with higher priority. For a task $t$, its data reduction rate is calculated as $reduceSizeRate = reduceSize/ET$,

where $ET$ is the estimated execution time of task $t$. $reduceSize$ can be calculated from $Size(OutputData) - Size(InputData)$. If the data is located not on the current VM, then $Size(InputData) = 0$.

**Algorithm 1** TR-Scheduling

```
1:  Input: VM $v$ with a free core, $\gamma$, K, candidateStages
2:  Output: Task Id
3:  Initialize currentBestTaskId, candidateTasks, stage to null
4:  if (stage is null) then
5:      stage ← selects max reduceSizeRate stage from candidateStages
6:  end if
7:  for each task Id $i$ in stage do
8:      if $\frac{taskSet[i].ET}{v.E(t,T)} < \gamma$ then
9:          push $i$ to candidateTasks
10:     end if
11: end for
12: for each task Id j in candidateTasks do
13:     if currentBestTaskId is null then
14:         currentBestTaskId = j
15:     else
16:         currentBestTaskId ← max(
17: candidateTasks[currentBestTaskId].reduceSizeRate,
18: candidateTasks[j].reduceSizeRate)
19:     end if
20: end for
21: return currentBestTaskId
```

### 3.2 TR-Checkpointing

As motivated above, checkpointing is crucial when operating in an environment with transient resources, but on the other hand, we do not want to checkpoint everything as checkpoints are expensive. A checkpointing strategy determines for each data block 1) whether it should be backed up and 2) if so, to where. The optimal checkpointing strategy backs up the right data blocks to the right location, so that the efficiency of the job is optimized. From our investigation, the ideal target data blocks for backup are those data blocks,

(1) whose VM will fail before they are consumed (processed by the next task), and

(2) whose re-computation cost is larger than their backup cost.

Principle (1) ensures that we only consider a minimal set of data blocks as checkpointing candidates. Given a data block, if it can be read and processed by its next stage before the VM fails (or is likely to fail), it does need to be backed up. Unfortunately, the next stage's starting time is typically difficult to be accurately predicted, and sometimes the VM failure time may also not be estimated accurately. In practice, we therefore take a probabilistic approach and give the data blocks with higher probability of being consumed a lower priority in the backup candidate list. For every backup candidate, Principle (2) further compares their backup cost to the hypothetical re-computation cost for re-generating this data block so that the most efficient decision can be made to ensure the efficiency of the entire job. There are different options for the backup location, and these options have different cost: *Backup to remote reliable storage* (e.g. Azure Blob or a dedicate HDFS cluster) has typically higher backup cost than *backup to another local VM* due to the differences in bandwidth. On the other hand, backing up data blocks to local VMs in a transient resource environment has the downside that these local VMs themselves may also be unstable. Thus, when backing up to a local VM, we need to take into account the additional hypothetical re-computation cost that is incurred if the local backup destination VM fails before the data block is consumed.

Thus, making the right backup decision is complex and should be made based as much as possible on an accurate cost estimation and calculation. In a real-world production environment, all parameters can vary dynamically. Having sufficiently accurate estimates for the key algorithm parameters and cost models (for both Principles (1) and (2)) is important for generating an optimal checkpointing strategy. We will discuss cost estimation in TR-Spark in detail in Section 4.

Following the above principles, *TR-Checkpointing* works as shown in Algorithm 2. The algorithm optimizes checkpointing decisions by taking into consideration both the lineage information from the DAG of each job and the environment stability distribution of the runtime environment.

**Algorithm 2** TR-Checkpointing

```
1:  Input: Data Block Set $b$
2:  $C_{BR}$ = BackupRemoteCost();
3:  $C_{Redo}$ = RecomputationCost();
4:  VM$_{id}$ = FindLocalBackupDestination();
5:  if VM$_{id} > -1$ then // Find a proper local backup destination VM.
6:      $C_{BL}$ = BackupLocalCost(VM$_{id}$);
7:  else
8:      $C_{BL} = double.max$ // Big cost value;
9:  end if
10: if $C_{BL} <= C_{Redo}$ then
11:     if $C_{BL} <= C_{BR}$ then
12:         Backup to local VM VM$_{id}$;
13:     else
14:         Backup to remote;
15:     end if
16: else
17:     if $C_{Redo} >= C_{BR}$ then
18:         Backup to remote;
19:     end if
20: end if
```

TR-Checkpointing is triggered by either a new coming event (such as a task accomplishment) or periodically. The estimation of backup cost $C_{BR}$, $C_{BL}$, re-computation cost $C_{Redo}$ and VM's failure probability before each stage is explained in the Section 4.

## 4. Cost Estimation

The effectiveness of both *TR-Checkpointing* and *TR-Scheduling* (and thus TR-Spark as a whole) is largely determined by its estimation of the costs of the various decision options. In this section, we introduce TR-Spark's VM instability model, as well as backup and re-computation cost estimation.

### 4.1 Transient Resource Instability

Suppose we know the distribution of a VM's lifetime (time until failure). Given this VM's failure PDF $f(x)$, $\int_0^\infty f(x)dx = 1$. Assume that a VM $v$ has been running for time $\tau$. Under this condition, the probability that $v$ will fail at time $t$ is $f(\tau,t) = \frac{\int_\tau^t f(x)dx}{\int_\tau^\infty f(x)dx}$. The expected lifetime of $v$ between time $t_i$ and $t_j$ is computed as $E(t_i,t_j) = \int_{t_i}^{t_j} f(\tau,t)tdt$ .

Different types of VMs will naturally have different distributions from which we can calculate the probabilities of failure at any specific time.[2] The two extreme cases are 1) we know the exact VM lifetime distribution (e.g., Amazon spot block), and 2) we know nothing at all about the distribution. In the latter case, the best we can do is to collect historical VM lifetime distribution information to get a statistical distribution function.

### 4.2 Re-computation Cost $C_{Redo}$

The re-computation of a task $k$'s output block $b_k$ is a cascading process, whose cost is estimated by the cost of the current task $k$ together with all of $k$'s parent tasks if their input data is not available due to a VM failure. Let $\tau_i$ be the existing running time of the VM which has data block $b_i$. Given the VM lifetime distribution, we can calculate the expected re-computation cost $C_{Redo}$ of data block $b_k$ from the equation as follows:

$$C_{Redo}(b_k) = \int_{t_c}^{T_{max}} f(\tau_k,t)Er(t,k)dt.$$

$Er(t,k)$ is the expected re-computation cost of task $k$ if the VM fails at time $t$. If $b_k$ is not consumed (there exists some task that depends on $b_k$), task $k$ which generates this block $b_k$ needs to be re-computated, then $Er(t,k) \neq 0$. Otherwise, $Er(t,k) = 0$. $Er(t,k)$ can be calculated as follows:

$$Er(t,k) = C_k + \sum_{i \in set_N} Er(t,i) + \sum_{j \in set_A} f(\tau_j,t)Er(t,j).$$

$k$'s expected re-computation cost $Er$ consists of three components:

- The re-computation cost of $k$: $C_k$ is the running time (cost) of task $k$ which needs to be recomputed.
- The re-computation cost of $k$'s dependent tasks which also need to be re-computed: $Set_N$ is the set of $k$'s dependent tasks whose result data blocks are lost due to VM's failure. If $k$ requires re-computation, these tasks in $Set_N$ also require re-computation.
- The re-computation cost of $k$'s dependent tasks which may require re-computation at a near future time $t$: $Set_A$ is the set of $k$'s dependent tasks whose result data blocks are available now on some other VM, but will be lost at time $t$ due to a future VM failure. If $k$ requires re-computation, some of the tasks in $Set_A$ require re-computation the cost of which can be calculated according to its VM's duration probability distribution.

The calculation of $Er$ is thus recursive. In practice, a recursion depth limit can be applied to control the computation overhead for scheduling efficiency. Note that the task's running time is always collected when it is accomplished, so we can estimate this task's re-computation cost when it requires re-computation.

### 4.3 Backup Cost $C_{BR}$ and $C_{BL}$

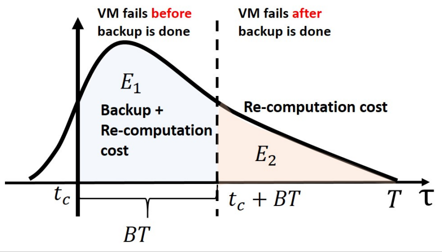

VM fails before backup is done
VM fails after backup is done
$E_1$
Backup + Re-computation cost
Re-computation cost
$E_2$
$t_c$
$t_c + BT$
$T$
$\tau$
$BT$

**Figure 4.** Backup cost components

When checkpointing a data block, there are two options: backup to remote reliable storage or backup to a more stable local VM. Let $BT(x)$ = Data Size / IO Cost($x$), $x$ =*local* for local backup time and $x$ =*remote* for remote backup time. The expected backup cost consists of three main components as illustrated in Figure 4:

- The backup cost when the VM fails before the backup is finished: $C_{B1}(x) = \int_{t_c}^{t_c+BT(x)} f(\tau,t)(t-t_c)dt.$
- The re-computation cost when the VM fails before the backup is finished: $C_R(x) = \int_{t_c}^{t_c+BT(x)} f(\tau,t) * Er(t,k)dt$
- The backup cost when the VM fails after the backup operation is finished: $C_{B2}(x) = BT \int_{t_c+BT(x)}^{T} f(\tau,t)dt$

Therefore, the backup costs are calculated as $C_B(x) = C_{B1}(x) + C_R(x) + C_{B2}(x)$ and $C_{BR} = C_B$(remote) and $C_{BL} = C_B$(local).

The cost estimation above is based on a parameter that characterizes the next stage's starting time, which is the earliest time for a data block to be consumed. It is non-trivial to accurately estimate this value due to the different starting and execution times of the tasks in the current stage. In the presence of failures, estimation becomes even more

[2] We do observe this very clearly in our implementation in Azure Batch.

inaccurate. In practice, we use $\frac{\sum_{PS} N_i * T_i}{\#core} * \alpha$ for estimating the parameter, where $N_i$ is the number of tasks in stage $i$ that have not yet finished. $T_i$ is the average running time of tasks in stage $i$ and $\alpha >= 1$ is an inverse function of the VMs' instability. That means that as the VMs become more unstable, we obtain a longer stage execution time estimation.

The above cost estimation models are applicable to all VM instability settings (deterministic and indeterministic VM lifetime distributions). As mentioned, it applies well to the case of running TR-Spark on Azure Batch, as we have detailed lifetime prediction models at the granularity of individual underlying VMs. In the case where no explicit information is available with high confidence, and the average failure rates are extremely high, our general backup strategy would naturally reduce to an "Always Checkpointing" strategy which backs up every data block to remote storage.

## 5. Implementation

We implement TR-Spark, a new version of Spark based on the checkpointing and scheduling algorithms described in the previous sections. The detailed implementation is shown in Table 1 with several practical optimizations to further improve efficiency, as well as to guarantee worst case system performance in transient resource environments (the latter being important in many production environments).

**Safety Belt:** As discussed in Section 4, the cost estimation is based on the transient resource's stability. In practice, one concern is the behavior of TR-Spark in case the actual resource stability in the cluster behaves radically different from the assumed distributions. In this case, the system should have the ability self-adjust to guarantee an acceptable performance. For this reason, our implementation of the TR-Checkpointing algorithm adds one extra rule: when the current task's total re-computation cost exceeds $\beta < 1$ times the job's execution time before re-computation is started, then we forcibly trigger an 'always checkpointing' policy.

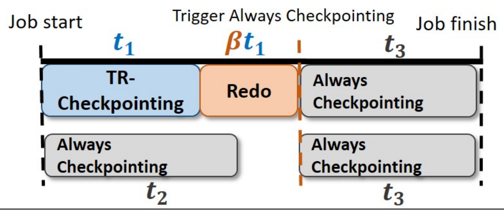


**Figure 5.** Live-site checkpointing plan

Figure 5 illustrates this plan's updates. Suppose the job execution time before re-computation is $t_1$ and the re-computation time is $\beta t_1$, while when always checkpointing from the beginning we need only $t_2$. We know that $t_1 <= t_2$, and $t_1 + \beta t_1 <= 2t_2$. Therefore, the total job running time can be guaranteed to be less than twice the cost of always checkpointing. Thus, we can guarantee that in the worst case, TR-Spark performs at most 2x as efficient as "always checkpointing" strategy. Notice that this bound holds even if our estimates are completely wrong. Of course, as long as our estimates are correct, then TR-Spark is always at least as good as the "always checkpointing" strategy.

**Increasing resource utilization**: In the checkpointing process, the data block which is assigned to be backed up locally or remotely should be checkpointed immediately in order to avoid data loss caused by resource instability. During the backup execution, if this is a free CPU resource, we schedule the next task to maximize the concurrency of checkpointing and task execution.

**Dynamic task splitting & duplication**: Relatively long running tasks (LRTs) significantly prolong job latency. In a transient resource environment, LRTs have even more negative impact: firstly, LRTs have higher failure probability on the VMs and secondly, LRTs increase the re-computation and backup cost. Usually, when LRT delays the job response time seriously, the system will trigger task splitting. Some related work [23] has summarized the reasons for LRT and proposed some practical solutions. In our TR-schedule, we can leverage the existing strategies through splitting the tasks to reduce the skewness. For example, when LRT tasks are longer than every VM's expected lifetime, it implies that those tasks cannot be successfully run with high probability. In this case, the system triggers a task splitting.

In the transient resource environment, sometimes there are idle free resources in the VM cluster and no task is pending for execution. To fully utilize the free resources for improving the job's efficiency, task duplication [40] can be applied. When a VM $v$ is waiting for task assignment for time $t$, duplication is triggered. To perform duplication, we first sort the running tasks by their duplication rates, the top task is duplicated from its current VM $s$ to $v$. Task $i$'s duplication rate can be computed as $duplicationRate_i = \frac{reduceSizeRate_i * s.E(t, t+ET_i)}{v.E(t, t+ET_i)}$

where $ET_i$ is task $i$'s estimated execution time and $s.E(t_x, t_y)$ is used to denote the expected lifetime of VM $s$ between times $t_x$ and $t_y$. After duplication, some tasks may have multiple replicas running concurrently. Whenever one of these replicas is successfully completed, the other replicas can be killed. Also, when there is no free resource available (waiting task queue is not empty). The duplicated task with the most remaining time should be killed.

## 6. Evaluation

Our evaluation consists of two parts. First, to study the effectiveness of the scheduling and checkpointing algorithms, we conduct evaluations on a home-built TR-Spark simulator with different parameters and environment settings. Then, to evaluate the robustness and scalability of our implementation, we deploy and conduct evaluations of TR-Spark on Azure Batch with benchmark as well as production workloads. TR-Spark is implemented on Spark 1.5.2.

We compare TR-Spark to the original *Spark* as well as Spark with *Always Checkpointing*, which checkpoints every task's output whenever it is generated. The efficiency

**Table 1.** TR-Spark design

| Module | Class | Base Class | Functionality | Method |
|---|---|---|---|---|
| Task Scheduler | TRDAGScheduler | DAGScheduler | Collect runtime stages and task information, launch checkpointing scheduler and adapt old logic to current strategies | handleTaskCompletion() handleExecutorLost() getCheckpointingStages() |
| | TRTaskSetManager | TaskSetManager | Task-level TR-Scheduling | dequeueTask() |
| | TRSchedulingAlgortihom | SchedulingAlgortihom | Stage-level TR-Scheduling | comparator() |
| | TRShuffleMapTask | ShuffleMapTask | Update meta data and trigger checkpoiting | runTask() |
| | TRExecutorAllocationManager | ExecutorAllocationManager | Update resource status | schedule() |
| | TRScheduleMetricsListener | ScheduleMetricsListener | Getting the shuffle metrics | onTaskEnd() |
| Checkpointing Scheduler | TRBlockManagerMasterEndpoint | BlockManagerMasterEndpoint | Sending checkpointing RPC | receive() |
| | TRCheckPointingScheduler | | TR-Checkpointing | generateCheckpointPlan() |
| Checkpointing Manager | TRBlockManager | BlockManager | Launch checkpointing Manager, implement checkpointing primitives, adapt old logic to current strategies | getBlockData() removeBlock() replicate() |
| | TRBlockManagerSlaveEndpoint | BlockManagerSlaveEndpoint | Getting RPC messages from Driver and calling backup primitive | receive() |
| | AzureBlobBlockManager | ExternalBlockManager | Checkpointing to Azure Blob | putBytes() getBytes() |
| | HDFSBlockManager | ExternalBlockManager | Checkpointing to HDFS | putBytes() getBytes() |
| Shuffle | TRNettyBlockRpcServer | NettyBlockRpcServer | Control the network traffic and requests priority | receive() |
| | TRShuffleBlockFetcherIterator | ShuffleBlockFetcherIterator | Adapt shuffle reader to support fetch from reliable storage | fetchLocalBlocks() sendRequest() |

results are expressed as the slowdowns compared to the performance of running original Spark on stable resources.

We choose four representative workloads: (1) TPC-DS Benchmark [9] contains SQL-like job workloads with input data size 2TB; (2)-(3) Production workloads for Bing API and Cortana session analysis with input data sizes 1TB and 950GB, respectively; (4) PageRank with input data size 250GB. Example job DAGs are illustrated in Figure 6.

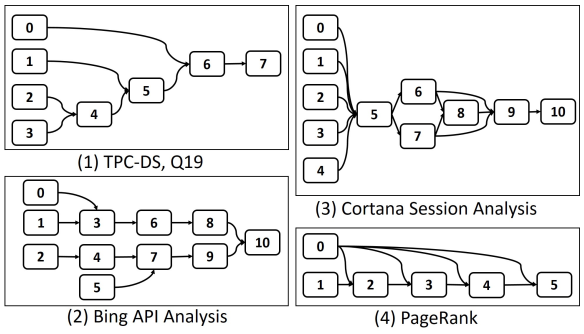


**Figure 6.** Job DAG example

Evaluations are conducted at with different degrees of resource stability, defined as the ratio between a VM's mean lifetime and the average task's execution time: VM stability = $\frac{AVG(VM\ lifetime)}{AVG(task\ latency)}$. VM lifetime distribution is approximated as an exponential distribution [13, 30], $y = \frac{1}{\lambda}e^{-\lambda x}$ with $\lambda$ varying from 0.005 to 0.5. VM mean lifetime is $\frac{1}{\lambda}$, and variance is $\frac{1}{\lambda^2}$.

Each performance result is reported as the average results of 50 runs. The parameters and their ranges considered in the experiments are listed in Table 2. In each group of evaluations, we vary some parameters within their ranges. For the remaining parameters, unless mentioned otherwise, the default values are selected.

**Table 2.** Parameter Setting

| Parameter | Range | Default |
|---|---|---|
| Number of VM | 50-200 | 100 |
| Number of cores per VM | 2-16 | 4 |
| VM Lifetime - Exponential distribution $\lambda$ | 0.005-0.5 | 0.15 |
| Local per VM bandwidth [2] | 200-400 MB/s | 400 MB/s |
| Remote per VM bandwidth | 50 - 200 MB/s | 100 MB/s |
| Remote total bandwidth limit | - | 10 GB/s |

## 6.1 Effectiveness of the Algorithms

We implement a TR-Spark simulator that accurately implements different Spark schedulers, including TR-Spark. The principle events such as task submission, task completion, VM failure with representative workloads over different scalability of clusters are modeled in the simulator, as well as different settings like DAG workflows, network bandwidth, VM stability and input data size. The evaluations are done using real-world representative workload traces.

### 6.1.1 Effect of Resource Instability

In the first group of evaluations, we seek to understand the robustness of TR-Spark on resources with different stabilities. Performance results across four different workloads are illustrated in Figures 7 and 8. For TR-Spark, we also consider different settings to profile its effectiveness. More specifically, we use **TR-Spark (deterministic, remote)** to represent the setting where each VM's lifetime is known [1]

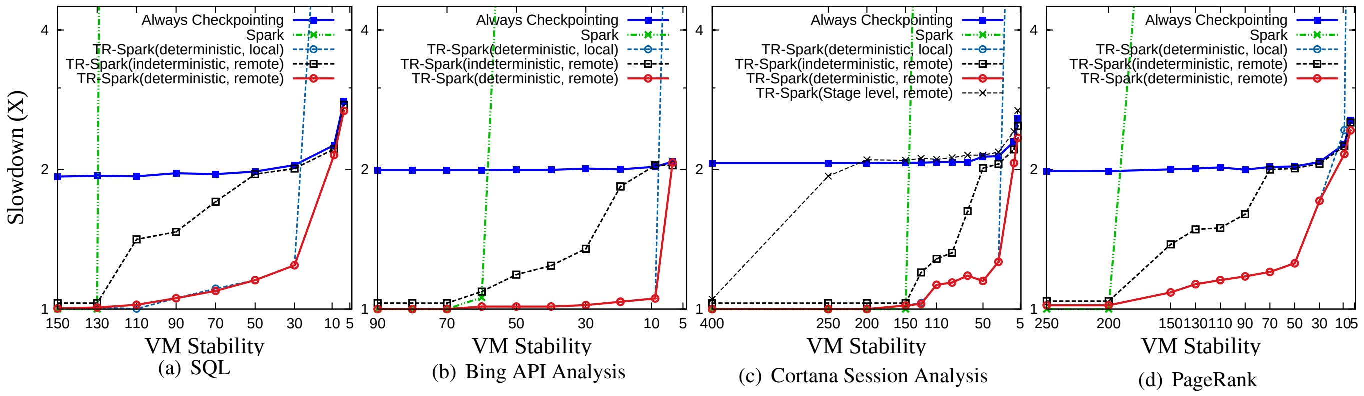


**Figure 7.** Performance with different resource instabilities

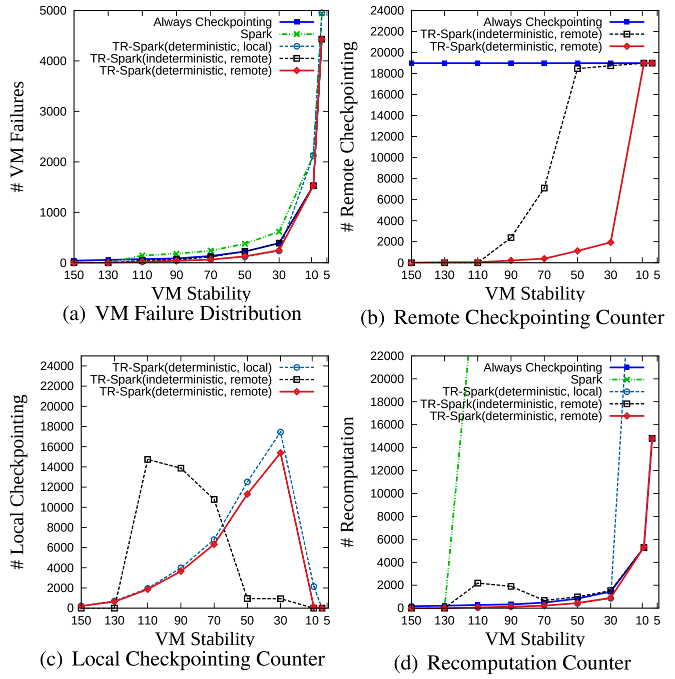


**Figure 8.** Perf Counters of SQL Workload

and remote reliable storage is available. We use **TR-Spark (indeterministic, remote)** to represent the setting where only a statical (historical) lifetime distribution is known and remote reliable storage is available. **TR-Spark (deterministic, local)** is used to represent the case when each VM's exact lifetime distribution is known but there is no remote storage available. With the above settings, we gain insight into the effects of resource instability and the importance of reliable storage in our algorithms.

From the results in Figure 7, it is clear that original Spark fails to work on transient resources. As the VMs stability decreases, the performance of Spark decreases exponentially. Most of the jobs cannot even get completed within a reasonable amount of time. The fundamental reason is the excessive cost of cascading re-computations (see counters in Figure 8(d)). *Always checkpointing* fails to adapt to the instability of the environment. It performs well when VMs are highly unstable, but it is inefficient in less unstable environments as it backs up every intermediate result to reliable storage. When the environment is relative stable, usually, not all data needs to be backed up. As shown in Figure 8(b), its backup count is much bigger than *TR-Spark* with different settings. The latency of *Always checkpointing* is also affected by the remote bandwidth cost which will be discussed in 6.1.4.

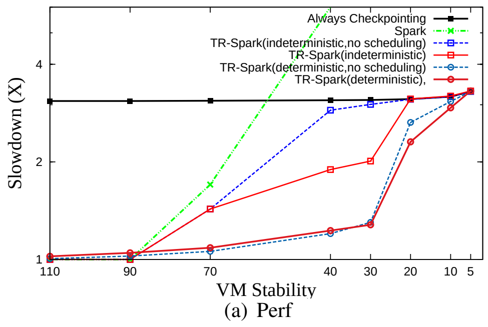
*TR-Spark (deterministic, local)* performs poorly when the resources become more unstable because in that situation, it is almost impossible to find a local reliable backup destination. *TR-Spark (deterministic, remote)* addresses this problem by checkpointing more intermediate results to remote reliable storage. It gracefully degrades to always checkpointing when the resource instability is extremely high. Importantly, *TR-Spark (indeterministic, remote)* without accurate VM lifetime information is still much better than both the original Spark and Spark with always checkpointing. It does not perform as well as *TR-Spark (deterministic, remote* because the failure of VMs can not be predicted accurately. To reduce the number of re-computations, it prefers to checkpoint more often (Figure 8(c)). The results from different workloads confirm the effectiveness of TR-Spark.

From the backup to local counters in Figure 8(c), we see how the checkpointing plan adapts to the instability of the resources. When the resources are stable, the VM failure probability is small, therefore, the re-computation cost is small. There are no checkpointing decisions to be made. As the resources become more unstable, re-computation costs start to increase and checkpointing decisions should be made. When there exist local VMs suitable to be a backup destination, a local backup plan is chosen. As the resources become even more unstable, there are fewer qualified local backup destination VMs and therefore, a remote backup plan is chosen. This also explains why our checkpointing algorithm automatically reduces to Always Checkpointing strategy in the limit.

To analyze the performance of stage level checkpointing (similar granularity to [30]) in our VM distribution setting, we conduct one more evaluation over Cortana workload. As shown in Figure 7(c), when the VM becomes less stable, in every stage, there is always some VMs which are not stable, which triggers checkpointing. Therefore, the checkpointing overhead increases.

### 6.1.2 Effectiveness of TR-Scheduling

In this group of evaluations, we want to examine the effectiveness of our TR-Scheduling algorithm. We compare the performance of TR-Spark *with* and *without* TR-Scheduling on workload (2). The number of VMs is 50. The local and remote bandwidth is 400 and 100 MB/S. The result is illustrated in Figure9(a). We can see that with TR-Scheduling, the performance of TR-Spark improves when the resources are less stable. This is because TR-Scheduling can reduce the number of costly checkpointings, thus reducing the total number of re-computations (Figure 9(b)) through prioritizing tasks that can reduce the output data size the most.



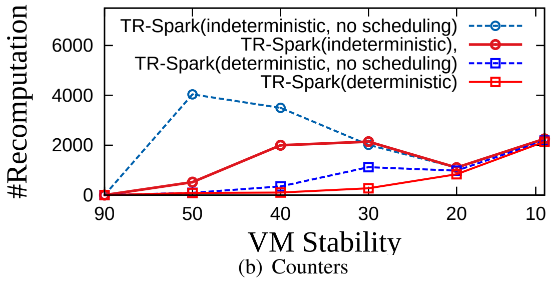


**Figure 9.** Impact of TR-Scheduling

### 6.1.3 Scalability

We study the scalability of our proposed solution in this group of evaluation. We use workload (1) TPC-DS Benchmark. When increasing the number of VMs from 50 to 200, with fixed input data size of 2T, the job latency is reduced. The input data is read from local HDFS and checkpointed to Azure blob. Figure 10(a) shows the changes of the VM instability. Spark fails to get reasonable performance as the number of VMs increases. As shown in Figure 10(b), the always checkpointing strategy faces a dramatic slowdown when the job runs on 200 VMs. This is because when there are 200 VMs, the available bandwidth per VM becomes small due to the limitation of the total remote bandwidth. Therefore, the relative remote checkpointing cost increases. In contrast, as shown in Figures 10(c) and 10(d), TR-Spark is highly effective at different cluster sizes.

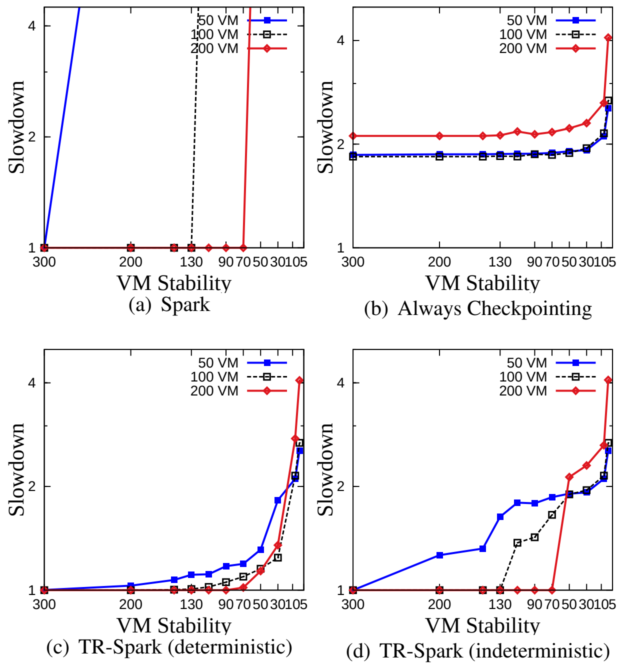


**Figure 10.** Scalability

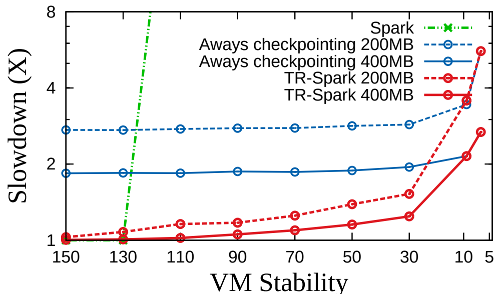


**Figure 11.** Effect of Bandwidth

### 6.1.4 Effect of Bandwidth

In this group of evaluations, we change the local and remote bandwidth with fixed local : remote bandwidth ratio to examine the effect of bandwidth to our solution. The number of VMs is set to 100. As the local bandwidth increases from 200 to 400 MB, (Figure 11), the remote bandwidth is also increased. The checkpointing cost thus becomes cheaper and TR-Spark becomes more efficient. Although we might expect that with the increase of bandwidth, the original Spark should be more efficient because of the cheaper shuffle cost, it still performs poorly when the resource become unstable. Indeed, it cannot be used at all on transient resources.

#### 6.1.5 Robustness to Imprecision of Resource Stability Estimation

The cost calculation in the scheduling and checkpointing algorithms are based on information about instability of the underlying resources. In this group of experiments, we examine the robustness of our algorithm to imprecisions in these estimations. Specifically, we manually add an error to the estimation of the mean lifetime and report the resulting latency slowdown. The evaluation is done on workload (1) with the error ranging from 20% to 50% over different numbers of VMs. VM instability is 90. The setting of TR-Spark we use is TR-Spark (indeterministic, Remote). From the results shown in Figure 12 we see that as imprecision increases slowdowns also increase. However, importantly, even when the estimation error is 50% over 200 VMs, the slowdown is still no bigger than 30% which confirms the robustness of our proposed solution.

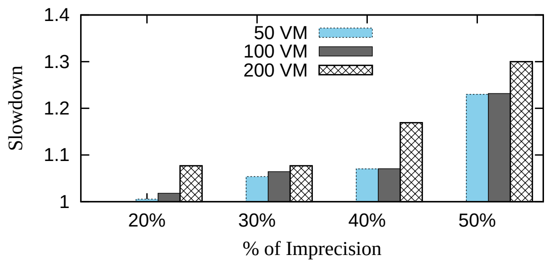


**Figure 12.** Robustness

### 6.2 TR-Spark on Azure

Finally, we deploy our implementation of TR-Spark on Azure Batch D3 cluster (4 Intel(R) Xeon(R) CPU E5-26600@2.20GHz, 14GB memory) with local and remote bandwidth 200 and 60 MB/s. TR-Spark is built on and compared to Spark 1.5.2. To simulate different VM stabilities, we kill VMs according to the VM lifetimes modeled as exponential distributions following workload data from the underlying Azure Fabric. We evaluate the performance and scalability of TR-Spark with TPC-DC (400GB input), PageRank (50GB input) and production Cortana session (100GB input) workloads. Performance is measured by the slowdown compared to Spark on stable resources. First, we fix the number of VMs to be 20 and examine the performance of TR-Spark over different workloads. As shown in Figures 13(a) - 13(c), the original Spark does not work when the resources become less stable. In contrast, TR-Spark always performs very well with different resource stabilities. Then, when we vary the number of VM from 5 to 50, as shown in Figure 13(d), the performance gain compared to original Spark on stable resources over SQL workload is stable across different scales.

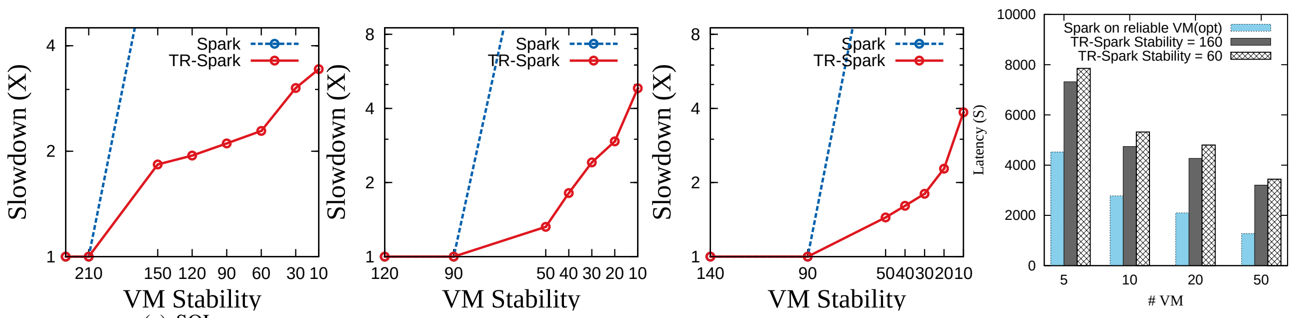
In summary, the above evaluations in both simulator and real cloud environments confirm the efficiency and effectiveness of TR-Spark on transient resources.

## 7. Related Work

**Transient resource aware checkpointing**: Many existing works [10, 13, 14, 18, 19, 21, 35, 36, 41] propose solutions to minimize the cost and optimize the performance through checkpointing based fault-tolerance. Specifically, [35, 36] proposes job level checkpointing and migration planning specific to Amazon EC2 spot instance based on price prediction. [14, 21] provides interval-based checkpointing solutions. [19] proposes monetary cost optimizations for MPI-based applications with deadline constraints on Amazon EC2 spot instance. [41] presents a solution to maximize the revenue through a dynamic resource allocation plan. [10] further considers Map-Reduce jobs: The solution is to split an original reduce task into fine-grained tasks and checkpoint intermediate data at key boundaries. [18] provides an efficient checkpointing implementation. The above two works are complementary to our work in this paper. [13] is maybe most closely related to our work. It leverages Amazon EC2 spot instance pricing as an estimation of resource unavailability and proposes MapReduce job's checkpointing decisions, however without co-considering scheduling plan and complex job DAG structure. [28] proposed a cost-based materialization (checkpointing) scheme for fault-tolerance with the assumption that intermediates are not lost by failure which is different from the VM failure settings we consider. The recent effort Flint [30] has a similar optimization goal as our work. Flint is based on RDD-level checkpointing, while TR-Spark provides finer granularity - task-level checkpointing and together with its scheduling algorithm can perform well over resources with different instability. The work of [11] provides checkpointing to remote scheme for Amazon spot instance, while our work considers more checkpointing location options. In contrast to these works, the solution in this paper is the first comprehensive solution to explicitly consider the performance of Spark on transient resources, co-optimizing scheduling and checkpointing strategies. Our implementation and evaluations are based on a real system implementation that is deployed on Azure Batch.

**Large scale job scheduling**: Scheduling has been studied almost ad infinitum. [22] presented a framework to optimize dataflow scheduling in the Cloud. [26] provides heuristics for task scheduling along the DAG with budget constraint. [15] designed online heuristics for optimizing job latency and cost. [42] provides solutions to scientific workflow scheduling on the cloud with user-defined optimization goals and constraints. [37] proposed a delay-scheduling algorithm to increase the job's throughput. However, these works are based on the assumption that the underlining resources are stable. The intermediate states are always available even after pausing and restarting.

**Big data analytic systems and fault-tolerance**: Efficient big data analytic systems [6, 12, 17, 20, 27, 31, 33, 39] have fault-tolerance mechanisms. As described, these systems are designed to run on dedicated clusters instead of the cloud



(a) SQL (b) Cortana (c) PageRank (d) Scalability

**Figure 13.** TR-Spark on Azure

transient resources. Our solution and principles are applicable to all these systems.

## 8. Conclusion

A large-scale public cloud provider like Microsoft and Amazon operates hundreds of thousands of servers across the globe. Running big data analytic jobs en masse on *transient resources* (possibly at lower price) could be a key tool to increase utilization. In reality, however, modern distributed data processing systems such as MapReduce or Spark are designed to run on dedicated hardware, and they perform badly on transient resources because of the excessive cost of cascading re-computations. In this work, we propose a new framework for big data analytics on transient resources, based on two principles: resource-stability and data-size reduction-aware scheduling and lineage-aware checkpointing. We implement our principles into Spark, producing TR-Spark (Transient Resource Spark). Intensive evaluation results confirm the efficiency of TR-Spark on transient resources. While original Spark is often unable to complete a task within a reasonable amount of time on even moderately transient resources, TR-Spark adaptively adjusts to the cluster environment, and completes all jobs within a near-optimal execution time.

## References

[1] http://aws.amazon.com/about-aws/whats-new/2015/10/introducing-amazon-ec2-spot-instances-for-specific-duration-workloads/.

[2] http://docs.aws.amazon.com/awsec2/latest/userguide/ebs-ec2-config.html/.

[3] https://aws.amazon.com/ec2/spot/.

[4] https://azure.microsoft.com/en-us/services/batch/.

[5] https://cloud.google.com/compute/docs/instances/preemptible.

[6] https://flink.apache.org/.

[7] http://spark.apache.org/.

[8] http://spark.apache.org/docs/latest/job-scheduling.html.

[9] http://www.tpc.org/tpcds/.

[10] G. Ananthanarayanan, C. Douglas, R. Ramakrishnan, S. Rao, and I. Stoica. True elasticity in multi-tenant data-intensive compute clusters. In *ACM Symposium on Cloud Computing, SOCC '12*, page 24, 2012.

[11] C. Binnig, A. Salama, E. Zamanian, M. El-Hindi, S. Feil, and T. Ziegler. Spotgres - parallel data analytics on spot instances. In *ICDE Workshops*, pages 14–21, 2015.

[12] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: easy and efficient parallel processing of massive data sets. *PVLDB*, 1(2), 2008.

[13] N. Chohan, C. Castillo, M. Spreitzer, M. Steinder, A. N. Tantawi, and C. Krintz. See spot run: Using spot instances for mapreduce workflows. In *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud'10*, 2010.

[14] S. Di, Y. Robert, F. Vivien, D. Kondo, C. Wang, and F. Cappello. Optimization of cloud task processing with checkpoint-restart mechanism. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC'13*, pages 64:1–64:12, 2013.

[15] S. K. Garg, R. Buyya, and H. J. Siegel. Scheduling parallel applications on utility grids: Time and cost trade-off management. In *Computer Science 2009, Thirty-Second Australasian Computer Science Conference (ACSC 2009)*, pages 139–147, 2009.

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

[17] I. Goiri, R. Bianchini, S. Nagarakatte, and T. D. Nguyen. Approxhadoop: Bringing approximations to mapreduce frameworks. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15*, pages 383–397, 2015.

[18] I. Goiri, F. Julià, J. Guitart, and J. Torres. Checkpoint-based fault-tolerant infrastructure for virtualized service providers. In *IEEE/IFIP Network Operations and Management Symposium, NOMS 2010*, pages 455–462, 2010.

[19] Y. Gong, B. He, and A. C. Zhou. Monetary cost optimizations for mpi-based HPC applications on amazon clouds: checkpoints and replicated execution. In *Proceedings of the International Conference for High Performance Computing, Net-*

*working, Storage and Analysis, SC 2015*, pages 32:1–32:12, 2015.

[20] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2007 EuroSys Conference*, pages 59–72, 2007.

[21] D. Jung, J. Lim, H. Yu, and T. Suh. Estimated interval-based checkpointing (EIC) on spot instances in cloud computing. *J. Applied Mathematics*, 2014:217547:1–217547:12, 2014.

[22] H. Kllapi, E. Sitaridi, M. M. Tsangaris, and Y. E. Ioannidis. Schedule optimization for data processing flows on the cloud. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011*, pages 289–300, 2011.

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

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

[25] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Improving resource efficiency at scale with heracles. *ACM Trans. Comput. Syst.*, 34(2):6, 2016.

[26] A. Ramakrishnan, G. Singh, H. Zhao, E. Deelman, R. Sakellariou, K. Vahi, K. Blackburn, D. Meyers, and M. Samidi. Scheduling data-intensiveworkflows onto storage-constrained distributed resources. In *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2007)*, pages 401–409, 2007.

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

[28] A. Salama, C. Binnig, T. Kraska, and E. Zamanian. Cost-based fault-tolerance for parallel data processing. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 285–297, 2015.

[29] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *SIGOPS European Conference on Computer Systems (EuroSys)*, pages 351–364, Prague, Czech Republic, 2013.

[30] P. Sharma, T. Guo, X. He, D. Irwin, and P. Shenoy. Flint: Batch-interactive data-intensive processing on transient servers. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2016.

[31] P. Upadhyaya, Y. Kwon, and M. Balazinska. A latency and fault-tolerance optimizer for online parallel query plans. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, pages 241–252, 2011.

[32] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[33] C. Xu, M. Holzemer, M. Kaul, and V. Markl. Efficient fault-tolerance for iterative graph processing on distributed dataflow systems. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*, pages 613–624, 2016.

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

[35] S. Yi, A. Andrzejak, and D. Kondo. Monetary cost-aware checkpointing and migration on amazon cloud spot instances. *IEEE Trans. Services Computing*, 5(4):512–524, 2012.

[36] S. Yi, D. Kondo, and A. Andrzejak. Reducing costs of spot instances via checkpointing in the amazon elastic compute cloud. In *IEEE International Conference on Cloud Computing, CLOUD 2010*, pages 236–243, 2010.

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

[38] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012*, pages 15–28, 2012.

[39] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012*, pages 15–28, 2012.

[40] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008*, pages 29–42, 2008.

[41] Q. Zhang, E. Gürses, R. Boutaba, and J. Xiao. Dynamic resource allocation for spot markets in clouds. In *USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services, Hot-ICE'11*, 2011.

[42] A. C. Zhou, B. He, X. Cheng, and C. T. Lau. A declarative optimization engine for resource provisioning of scientific workflows in iaas clouds. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2015*, pages 223–234, 2015.