Pado: A Data Processing Engine for Harnessing Transient Resources in Datacenters

Youngseok Yang[1] Geon-Woo Kim[1] Won Wook Song[1] Yunseong Lee[1]
Andrew Chung[2] Zhengping Qian[3] Brian Cho[4] Byung-Gon Chun[1] *

[1]Seoul National University [2]Carnegie Mellon University [3]Alibaba Group [4]Facebook

Abstract

Datacenters are under-utilized, primarily due to unused resources on over-provisioned nodes of latency-critical jobs. Such idle resources can be used to run batch data analytic jobs to increase datacenter utilization, but these *transient* resources must be evicted whenever latency-critical jobs require them again. Resource evictions often lead to cascading recomputations, which is usually handled by checkpointing intermediate results on stable storages of eviction-free *reserved* resources. However, checkpointing has major shortcomings in its substantial overhead of transferring data back and forth. In this work, we step away from such approaches and focus on observing the job structure and the relationships between computations of the job. We carefully mark the computations that are most likely to cause a large number of recomputations upon evictions, to run them reliably using *reserved* resources. This lets us retain corresponding intermediate results effortlessly without any additional checkpointing. We design Pado, a general data processing engine, which carries out our idea with several optimizations that minimize the number of additional reserved nodes. Evaluation results show that Pado outperforms Spark 2.0.0 by up to $5.1\times$, and checkpoint-enabled Spark by up to $3.8\times$.

1. Introduction

Companies like Amazon, Facebook, Google, and Microsoft are continuously investing billions of dollars to increase the size and the capability of their datacenters to keep up with the ever-increasing demand in popular online services and complex data analytic workloads. Although the total amount of computing resources are greatly increasing with the investments, a large portion of resources in datacenters such as CPU and memory are left unused. A major reason is that latency-critical (LC) jobs, such as user-facing search engine services, are over-provisioned with excess resources in order to be responsive even at load spikes. However, the resources are actually left idle at most of the times [19, 25].

To increase datacenter utilization, researchers have developed runtime resource isolation and monitoring mechanisms to run batch jobs, such as data analytic jobs, on the unused idle resources of the LC jobs [19, 25, 30]. However, when LC jobs require resources again, the tasks of batch jobs running on these resources need to be evicted. From this property, we categorize such eviction-prone resources as *transient resources*. Although it is most ideal to use transient resources most aggressively, it leads to frequent evictions with the fluctuation of LC jobs. Indeed, based on the assumption that transient resources run on the unused resources of LC jobs, our analysis of a Google datacenter trace [27] shows that evictions can occur only a few minutes after the batch jobs are newly allocated with transient resources.

Many distributed data processing engines [12, 15, 31] have been introduced to run various data analytics jobs, but they were not designed to handle such high rates of evictions. Most engines handle evictions through recomputing from the last available intermediate result of previous computations. However, such fault-tolerance mechanisms are ineffective with transient resources, as intermediate results are repeatedly lost under frequent evictions, and requires numerous cascading recomputations to recover the lost data. This notably delays jobs from completion and causes a great deal of inefficiency in resource usages.

As a solution, recent works like Flint [23] and TR-Spark [29] focus on using additional nodes of eviction-free *reserved resources* as storages to checkpoint intermediate results. This allows computations to resume the work from the last checkpointed data. Nonetheless, checkpointing is very expensive for data-intensive workloads, since checkpointing requires transferring large amounts of data back and forth, and incurs substantial network and disk overhead. To allevi-

* Corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

EuroSys '17, April 23 - 26, 2017, Belgrade, Serbia

© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ISBN 978-1-4503-4938-3/17/04…$15.00

DOI: http://dx.doi.org/10.1145/3064176.3064181

ate the overhead, such systems introduce various techniques to decide the optimal amount of checkpointing. They predict eviction costs to selectively checkpoint where evictions incur high recomputation costs. However, if evictions of transient resources occur frequently, it forces checkpointing to be done repeatedly even with such techniques.

In this work, we step away from such approaches and focus on observing the job structure and the relationship between the computations of the job. Generally, data processing engines take an arbitrary DAG (Directed Acyclic Graph) of computations as its workload, where each vertex represents an operator or an execution, and each edge represents the dependency of data flow. Instead of checkpointing intermediate results, we focus on observing the DAG of computations to use the additional *reserved resources* to selectively run the computations that are most likely to cause high recomputation costs once evicted. For example, as many tasks are involved in a shuffling edge, a single eviction of a task can lead to a large number of recomputations of its dependent tasks, and we choose to run such operators reliably. As a result, our approach reliably retains corresponding intermediate results effortlessly on reserved resources.

This idea is embodied in a general-purpose data processing engine called Pado. Pado consists of two main components: the Pado Compiler and Runtime. The compiler takes input programs and analyzes their derived logical DAGs to select and place a set of operators that are more likely to cause high recomputation costs on reserved resources, and the rest on transient resources. Then, the logical DAG is partitioned into stages based on the placement information, each of which acts as a unit of execution. Using the DAG of stages, the runtime generates physical execution plans and schedules the generated tasks across combinations of reserved and transient resources. During the execution, the outputs of the tasks placed on transient resources are transferred as soon as they are completed to the tasks allocated on reserved resources so that they can quickly escape from the threat of evictions. The runtime also provides several optimizations, such as task input caching and task output partial aggregation, to reduce the load and to minimize the amount of additional reserved resources.

We have integrated Pado with several big data open source projects, in order to facilitate real-world deployments. Our implementation supports programs written with Beam [1], a programming model initially developed by the Google Dataflow [2] team, and runs on various datacenter resource managers like Mesos [14] and Hadoop YARN [5] by using the REEF library [26].

We evaluated Pado with several real-world applications on a cluster of Amazon EC2 instances, which we set up to simulate a datacenter environment with transient resources. We obtained the transient container lifetimes by analyzing a Google datacenter trace [27]. The results show that under a high rate of evictions, Pado outperforms Spark 2.0.0 [8] by up to $5.1\times$ and checkpoint-enabled Spark, which encompasses ideas proposed by Flint [23], by up to $3.8\times$. Using Pado, datacenters can greatly increase utilization through effectively running batch jobs using wasted idle resources aggressively collected from datacenters.

2. Background

In this section, we introduce how transient resources are used in datacenter environments that we assume, and the behavior of different data processing engines in them.

2.1 Resources on Datacenters

We target datacenters in which resource managers [14, 24, 25] manage computing resources such as CPU, memory, network, and disk on a large number of nodes. The resource manager collects and allocates *containers*, each of which is a slice of resources of a node, to set up an environment for running heterogeneous jobs. Normally, each of the containers is *reserved* for a job until the job voluntarily releases it to be collected by the resource manager.

Latency-critical (LC) jobs, which have strict service-level objective (SLO) latency bounds, use containers with overprovisioned resources to meet the SLOs at all times, even at load spikes. An example LC job is a user-facing search engine service that needs to responsively return search query results to its customers at any time of the day. However, as the average load is much smaller than at load spikes, a large portion of resources are regularly left unused, making the datacenter under-utilized [21, 25].

To address this problem, resource managers like Borg and Mesos borrow the regularly unused resources from LC jobs, and use the resources to run new containers [14, 25]. However, such containers differ from the usual *reserved containers* which are guaranteed to be available for the job. Once LC jobs require the resources again, they must be yielded to the original LC jobs to meet their SLOs. Although resource managers allow some types of resources, like CPUs, to be throttled in such situations, other types of resources, like memory, have to be *evicted* [14, 19, 25]. In this paper, we focus on the eviction aspect and call those containers vulnerable to evictions as *transient containers*. We assume all state in transient containers, including those saved on their local disks, gets destroyed upon evictions [14].

To obtain transient container lifetimes and their eviction rates in real-world datacenters, we analyzed a Google datacenter trace of average memory usage records given in 5-minute intervals [27]. However, as we found 5-minute intervals overly coarse-grained compared to real-world environments, where resources are immediately returned to LC tasks as soon as they are needed, we have applied the B-spline function to acquire memory usage records in a more fine-grained 1-minute intervals, which is commonly used for curve-fitting of experimental data [11]. Considering LC jobs as those tagged as the most latency-sensitive and the highest-

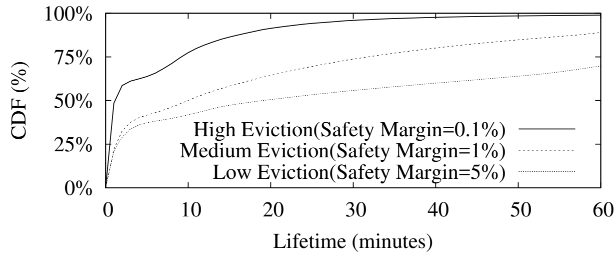

Figure 1. CDFs of transient container lifetimes over different safety margins.

Safety Margin	0.1%	1%	5%
10th Percentile	1 min	1 min	1 min
50th Percentile	2 mins	10 mins	20 mins
90th Percentile	19 mins	64 mins	276 mins

Table 1. Time to different percentiles of transient container lifetimes over different safety margins.

Safety Margin	Baseline	0.1%	1%	5%
Collected Mem	26.0%	25.9%	25.3%	22.7%

Table 2. Collected idle memory from total memory allocated to LC jobs over different safety margins. Baseline indicates collection of all idle memory.

priority jobs, we observed the containers of the LC jobs. Assuming that *transient containers* run on the unused resources of the LC job containers, we were able to figure out when the transient containers were evicted, by applying the technique introduced in Borg using *safety margins* [25]. Here, we set up transient containers with the unused memory in each of the LC job containers, while leaving a portion, the buffer memory, untouched to prevent evictions from negligible LC job fluctuations. The maximum size of the buffer memory is given by $(total_LC_mem \times safety_margin)$, thus the *safety margin* indicates the percentage of the memory that we try to leave intact. Under this condition, once the memory usage of a LC job decreases, the *transient container* on the same LC job container is additionally reallocated with the increased unused memory. On the other hand, if the LC job requires more memory, exceeding the value of the buffer memory, the transient container has to be evicted, as it indicates resource conflict.

With this assumption, we derived cumulative distribution functions (CDF) of transient container lifetimes and their eviction rates with three different safety margins, as depicted in Figure 1 and Table 1. Here, lower safety margin indicates aggressive resource collection, which leads to higher datacenter utilization. The 0.1% safety margin indicates that we aggressively use almost all the available idle resources, consisted of around 25.9% of the memory allocated to LC jobs as shown in Table 2. However, the 0.1% safety margin results in a high eviction rate, where most transient containers are evicted within half an hour. This implies that evictions occur much more frequently with transient containers compared to other environments that previous works assume [23, 29]. Such environments are mainly spot instances, which are revocable virtual machines that cloud providers like Amazon Web Services (AWS) provide at a lower cost compared to regular on-demand instances. Unlike transient containers, spot instances are usually revoked at an hourly or at a more moderate basis. Consequently, to effectively use transient containers and increase datacenter utilization, it is crucial for data processing engines to handle frequent evictions and complete their workloads with minimum delays.

2.2 Data Processing Engines

A plethora of distributed data processing engines [12, 15, 31] have been introduced to run batch data analytics jobs. They allow users to write dataflow programs with high level languages. In general, dataflow programs can be represented as *logical DAGs* of computations, in which each vertex represents an operator that processes data, and each edge represents the dependency of data flow between the operators. Data processing engines transform and run the logical DAGs as *physical DAGs* in which each operator is expanded into multiple parallel tasks to be distributed and run on containers, and each dependency is converted into a physical data transfer between the corresponding tasks.

In logical DAGs, we define four types of dependencies: **(1)** *one-to-one*, **(2)** *one-to-many*, **(3)** *many-to-one*, and **(4)** *many-to-many* dependency. **(1)** First, the *one-to-one* dependency describes a relation where each of the parent tasks only has a single child task and vice versa. **(2)** The *one-to-many* dependency describes a relation in which the results of the parent tasks are transferred to all tasks of the child operator. **(3)** The *many-to-one* dependency describes a relation where the results of the parent tasks are collected in a task of the child operator. **(4)** Lastly, the *many-to-many* dependency describes a relation where parent tasks and their children tasks are co-related to each other.

Figure 2 illustrates the logical and physical DAG representations of a simple Map-Reduce program in different data processing engines. We have selected Map-Reduce as our example workload for the ease of explanation, but this can be applied to any kind of programs expressed as a DAG of computations. In our example, the `Map` operator is expanded into tasks of the upper row, and the `Reduce` operator is expanded into tasks of the bottom row, both running on containers as shown in the figure. We assume that containers can run both `Map` and `Reduce` tasks. Figure 2(a) shows the logical DAG representation of the program, in which the `Reduce` operator depends on the `Map` operator with a *many-to-many* dependency. Due to the *many-to-many* dependency, each of the `Reduce` tasks needs the outputs of all `Map` tasks as its input. (b) and (c) each shows the physical DAG interpretation of the logical DAG in current data processing engines without

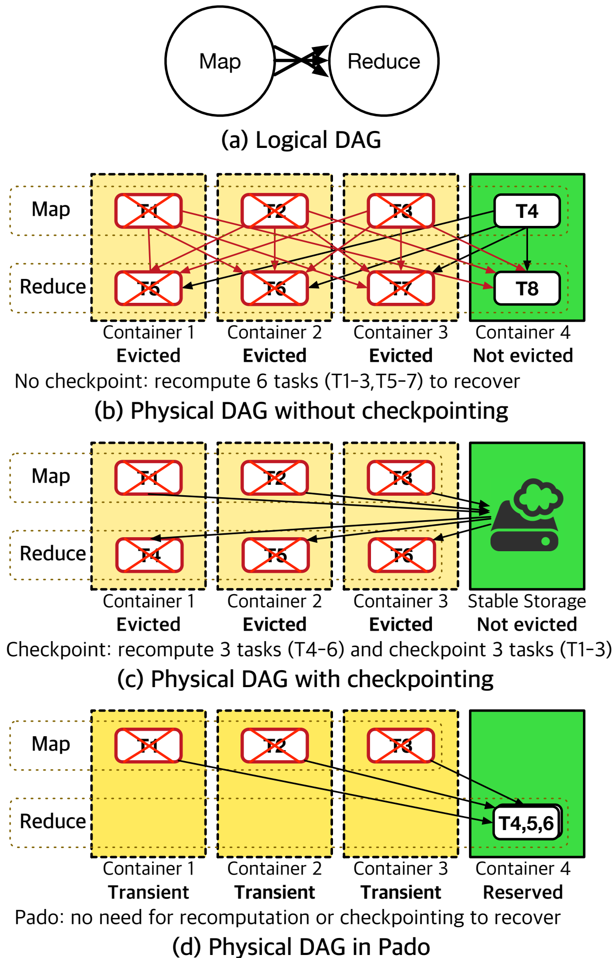

Figure 2. A Map-Reduce job's logical(a) and physical DAG representation in existing data processing engines, without(b) and with(c) checkpointing, as well as in Pado (d). We consider a case where transient containers 1 to 3 are evicted while running the `Reduce` operator. The arrows indicate dependencies of tasks, and red arrows indicate those of the tasks that must be relaunched upon evictions.

and with checkpointing enabled. Lastly, (d) shows the physical DAG that Pado generates. With this setup, we explore a case where container 4 is reserved and free from evictions, and containers 1-3 are transient and evicted at arbitrary time. As an eviction while executing the `Map` operator simply results in recomputations of evicted `Map` tasks, we focus on the effect of an eviction while executing the `Reduce` operator in this subsection.

In the case of general data processing engines, illustrated in Figure 2(b), when an eviction occurs, the engines first check whether the outputs of the parent tasks (1-4) of the evicted tasks (5-7) are available to be reused, and see which tasks need to be relaunched. Such engines, like MapReduce and Spark, maintain `Map` task outputs on local disk, for them to be *pulled* by the following `Reduce` tasks when needed. Therefore, the outputs of the `Map` tasks 1-3 are destroyed upon the container eviction, and they need to be recomputed along with the evicted `Reduce` tasks 5-7. This requires recomputations of a total of 6 tasks (1-3, 5-7) to recover from the eviction, delaying the job from completion. For more complex jobs, such as iterative algorithms, the delay is even more amplified. For example, if tasks 1-3 also had parent tasks that ran on transient containers, those parent tasks need to be recomputed as well, and the same applies for their parent tasks recursively. Such chain of cascading recomputations are called as a critical chain [16, 17].

To address the critical chain problem and to provide more fault-tolerance, data processing engines usually provide techniques to *checkpoint* intermediate results in remote stable storages placed on reserved containers. The idea is to checkpoint the outputs of the `Map` operator to remote storages to prevent recomputations of the evicted `Map` tasks. As shown in our example case (c), we would only need to recompute 3 tasks (5-7) to recover from the eviction and complete the job, as the outputs of the evicted Map tasks (1-3) are already checkpointed on the remote stable storages (container 4). However, the problem of checkpointing is that checkpointing incurs a considerable amount of additional network and disk I/O costs, which hinder jobs from completions. This overhead can become much larger depending on the amount of intermediate results that have to be sent back and forth with remote stable storages. Consequently, works like Flint [23] and TR-Spark [29] explore methods to checkpoint only when it is needed, by making predictions about task durations and container lifetimes to calculate recomputation costs. Nevertheless, recomputation cost rises under frequent evictions, and checkpointing has to be done frequently with those engines as well. Indeed, the mentioned works report that under frequent evictions, jobs can face severe performance degradation even with their sophisticated checkpointing mechanisms.

We propose a novel solution to overcome such limitations of current data processing engines, as briefly illustrated in (d). Here, we first compute the `Map` tasks on transient containers, and *push* the mapped data to reserved containers immediately upon completions, for them to quickly escape the risk of evictions. As the eviction occurs while performing `Reduce` tasks, the lost data on transient containers are already transferred to reserved containers at this point, hence there is no need for any recomputations or any checkpointing upon the eviction. This idea can be generalized for DAGs of arbitrary length and complexity. We observe the job structure and the relationships between the operators to sort out computations that are more likely to cause higher number of recomputations upon evictions, like the `Reduce` tasks in our example, to run them reliably on reserved containers. This is implemented with a simple algorithm (§3.1), that observes and processes the DAG prior to the execution, along with a runtime (§3.2) specifically tailored for our requirements.

With our idea, intermediate results then can be effortlessly preserved on *reserved* containers without the overhead of checkpointing or cascading recomputations.

3. Pado Design

Pado is our general-purpose distributed data processing engine tailored to harness transient resources in datacenters. Pado can be largely divided into the Compiler and the Runtime. The Compiler translates and processes dataflow programs into a DAG of Pado Stages, each of which is a unit of execution. The Runtime executes the processed DAG efficiently under frequent evictions using a combination of transient containers and a small number of reserved containers.

3.1 Compiler

The Pado Compiler receives and processes dataflow programs, represented as logical DAGs, through two major steps. First, the compiler *places* the operators in the logical DAG of the given program on transient or reserved containers. The compiler marks a set of operators that are more likely to cause larger numbers of recomputations upon evictions to run them reliably on reserved containers and the rest on transient containers. Next, leveraging the placement information, it partitions the logical DAG into Pado Stages, each of which serves as a basic execution unit in Pado. These subgraphs are later received by the Pado Runtime to be transformed into physical execution plans and run in distributed tasks. We describe the compilation process in detail and show how it is applied to a number of real-world data processing applications.

3.1.1 Operator Placement

Computations and their outputs placed on transient containers are vulnerable to data loss and recomputations due to container evictions, whereas those on reserved containers are free from evictions. However, as reserved containers are consisted of expensive resources that cannot be yielded to any other jobs, we need to keep the size of reserved containers as small as possible to maximize datacenter utilization. As a simple solution, the compiler observes the logical DAG and carefully selects the operators that are most likely to have the highest recomputation costs once evicted by observing their dependencies.

In the case of a child operator with a *many-to-many* or a *many-to-one* dependency from its parent operator, eviction of a *single* task can result in recomputations of *multiple* tasks of the parent operator, as it requires outputs of multiple parent tasks, similar to Reduce tasks in the Map-Reduce example in Section 2.2. In contrast, in the case of a child operator with a *one-to-one* or a *one-to-many* dependency with its parent operator, eviction of a *single* task only results in a recomputation of a *single* additional task of the parent operator, as it only requires the output of its single parent task.

Algorithm 1 Operator Placement Algorithm

```
 1: Input: Logical DAG dataflow-dag
 2: Output: Logical DAG op-placed-dag
 3: for op ∈ TOPOLOGICALSORT(dataflow-dag) do
 4:     if op.inEdges ≠ ∅ then                      ▷ Computational Operator
 5:         if op.inEdges.ANYMATCH(m-m or m-o) then
 6:             op.MARK(reserved)
 7:         else if op.inEdges.ALLMATCH(o-o) and
 8:                 op.inEdges.ALLFROM(reserved) then
 9:             op.MARK(reserved)
10:         else
11:             op.MARK(transient)
12:     else if op.inEdges=∅ then                   ▷ Source Operator
13:         if op.ISREAD then
14:             op.MARK(transient)
15:         else if op.ISCREATED then
16:             op.MARK(reserved)
```

Based on this simple intuition, the compiler places operators with complex dependencies with parent operators on reserved containers, and the rest on transient containers, while being aware of data locality. The placement algorithm is illustrated in Algorithm 1. The semantics of algorithms are explained in parenthesis throughout the section.

First of all, we sort the DAG in a topological order and observe each operator. As described in the algorithm, each operator is placed by the following policy:

- Operators with *any* (ANYMATCH) incoming *many-to-many* (`m-m`) or *many-to-one* (`m-o`) dependencies from parent operators are placed on reserved containers. This prevents such tasks from being evicted and prohibits multiple recomputations of parent tasks.
- Operators with *all* (ALLMATCH) incoming edges that have *one-to-one* (`o-o`) dependency from parent operators and that *all* come from (ALLFROM) operators placed on reserved containers are also placed on reserved containers. This lets us exploit data locality on reserved containers.
- All operators that do not fall under the previous two conditions are placed on transient containers. This allows us to aggressively utilize transient containers where the risk of cascading recomputations are not as large.

Source operators, which do not have any incoming edges, are handled differently. Those that read their input from a storage, such as a distributed filesystem or a disk (ISREAD), are placed on transient containers to load large amounts of input data using many containers. On the other hand, those that newly create their data in memory (ISCREATED) are placed on reserved containers as the relatively lightweight created data can be kept on a small number of reserved containers. Our algorithm can be applied to a logical DAG with any length and complexity, and we can get a logical

Algorithm 2 Logical DAG Partitioning Algorithm

1: **Input**: Logical DAG *op-placed-dag*
2: **Output**: DAG of Pado Stages $stages$
3: **for** $op \in$ TOPOLOGICALSORT(*op-placed-dag*) **do**
4:     **if** op.ISMARKED(reserved) **or** $op.outEdges$=$\emptyset$ **then**
5:         $currStage := stages$.NEWSTAGE()
6:         RECURSIVEADD($currStage$, op)
7:
8: **function** RECURSIVEADD($currStage$, op)
9:     $currStage$.ADD(op)
10:     **for all** $parentOp \in op.inEdges$ **do**
11:         **if** $parentOp$.ISMARKED(transient) **then**
12:             RECURSIVEADD($currStage$, $parentOp$)
13:         **else if** $parentOp$.ISMARKED(reserved) **then**
14:             $parentOp.stage$.ADDCHILD($currStage$)

DAG in which every operator is marked to run on either a transient or a reserved container as a result.

3.1.2 Partitioning

To facilitate the execution and to easily keep track of the job progress, the compiler partitions the marked logical DAG into subgraphs called *Pado Stages*, each of which acts as a basic unit of execution. The idea of partitioning is also widely used by existing data processing engines, as it simplifies the implementations of task execution and fault tolerance mechanisms. Nevertheless, unlike the stages in general data processing engines, which are partitioned in the shuffle boundaries, Pado partitions stages based on the *operator placement information* that we have previously discussed.

The algorithm traverses the logical DAG in a topological order and creates a new stage at each of the operators placed on reserved containers or without any outgoing edges. At each of such operators, its parent operators placed on transient containers are recursively added to the stage. If the parent operator is placed on reserved containers, this indicates that it belongs to a previously created stage. Algorithm 2 shows how a DAG of Pado Stages is generated.

As the result of the partitioning algorithm, computations of a stage start on transient containers, if any exists, and finish on reserved containers, unless the DAG ends on a transient container. Also, as stages finish on reserved containers or at the end of the DAG, it ensures that all stage outputs are reliably conserved on reserved containers or written to sink, minimizing the risk of data loss. With this characteristic, following children stages can steadily fetch the intermediate results stored on reserved containers. This enables us to simply relaunch the evicted tasks of the stage that is running at the time of evictions, without having to recompute previous parent stages.

3.1.3 Application on Different Workloads

We use three real-world example workloads to show how our algorithms can be applied on different cases. We use Figure 3 to visualize each of the examples.

Map-Reduce: Map-Reduce is used for various large-scale Extract-Transform-Load (ETL) types of applications. The compilation result is illustrated in Figure 3(a). Following the placement algorithm 1, the `Read` operator and the following `Map` operator are placed on transient containers. Then, the next operator is placed on reserved containers, as it has a many-to-many incoming edge. For partitioning, the algorithm finds the `Reduce` operator on reserved containers while traversing the logical DAG, and adds up the in-edges placed on transient containers recursively to Stage a-1.

Multinomial Logistic Regression: Multinomial Logistic Regression is a machine learning application for classifying inputs, like classifying tumors as malignant or benign and ad clicks as profitable or not [13]. Such iterative workloads compute gradients to update the regression model, which is used to classify results and predict outcomes for arbitrary inputs. Its compilation result is illustrated in Figure 3(b). As illustrated, `Aggregate Gradients` has a many-to-one incoming edge, and `Compute 2nd Model` only has one-to-one relations from operators on reserved containers, thus both are placed on reserved containers. For source operators, `Create 1st Model` newly creates its data and hence is placed on reserved containers, and `Read Training Data` reads its data from a source, thus is placed on transient containers. The rest are placed on transient containers following the algorithm. As a result of partitioning, we can observe that there are three stages for the three operators on reserved containers.

Alternating Least Squares: Alternating Least Squares is another machine learning application used for recommendation services, such as for shopping or movie recommendation sites [18]. It alternates its computation and aggregation between user and item factors, and its compilation result is illustrated in Figure 3(c). The `Read` operator is placed on transient containers, then operators with many-to-many in-edges are placed on reserved containers. `Compute 1st Item Factor` operator only has a single one-to-one incoming edge from reserved containers and is placed on reserved containers to ensure data locality. The rest of the operators are placed on transient containers. By the partitioning algorithm, operators placed on transient containers are recursively added by the four operators on reserved containers.

Now, we describe how the Pado Runtime actually executes the DAGs of Pado Stages.

3.2 Runtime

The Pado Runtime receives and efficiently executes the DAG of Pado Stages with several techniques. As illustrated in Figure 4, the runtime consists of the *Pado master* that orchestrates the distributed workload, and multiple *Pado executors*

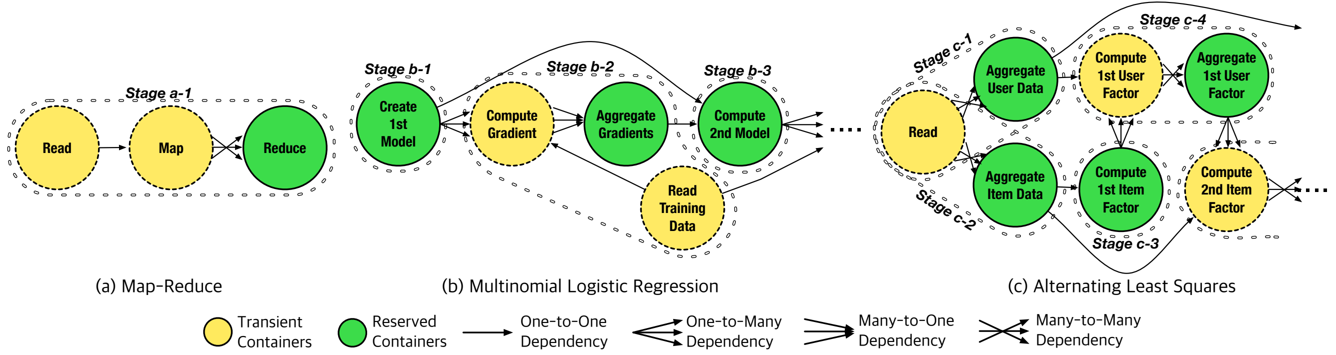

Figure 3. Compilation results of different workloads. Operators with complex dependencies with parent operators are placed on reserved containers, and all stages finish on operators placed on reserved containers.

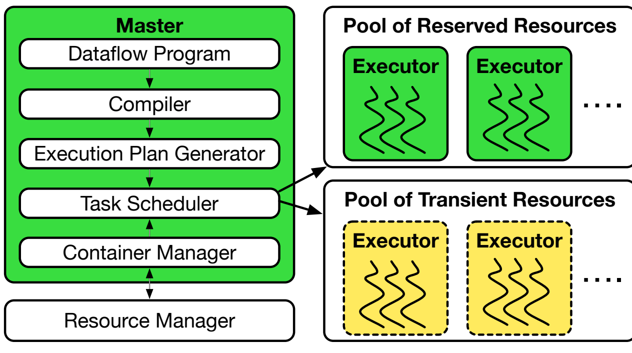

Figure 4. Pado Runtime

that carry out the actual execution. For every submission of a dataflow program, a *master* is launched by the resource manager that manages computing resources of the cluster (§2.1). Then the *container manager* in the master obtains a number of transient and reserved containers from the resource manager and launches them as *Pado executors*. The *execution plan generator* generates execution plans from the physical DAG of tasks, in which each operator of the stages is expanded into multiple parallel tasks, and each of the edges is translated into physical data transfers between the tasks. The *task scheduler* in the master then schedules and launches tasks of the generated execution plan in the Pado executors by each of the partitioned stages. Here, unlike existent runtimes that assume reserved containers, tasks are scheduled across a combination of reserved and transient containers. Finally, the scheduled tasks are executed in multiple threads of the executors and their intermediate results are shuffled across executors and *pushed* into reserved containers to quickly escape the threat of evictions. In the meanwhile, the runtime efficiently handles evictions, and provides several optimizations to minimize the load on the small number of reserved containers. We explain each of the components of Pado Runtime in more detail throughout the rest of the section.

3.2.1 Container Manager

The container manager in the *master* interacts with the resource manager to obtain, classify, and manage transient and reserved containers. It obtains a user-configured number of reserved and transient containers from the resource manager and launches *executors* on them. We call the executors running on transient containers as *transient executors*, and those on reserved containers as *reserved executors*. The container manager keeps track of the executors on different types of containers and notifies the task scheduler whenever a new executor comes online, so that the executor can be utilized. It also delivers container eviction notifications from the resource manager to the task scheduler to handle them accordingly.

3.2.2 Execution Plan Generator

The execution plan generator converts the DAG of Pado Stages created by the compiler into a physical DAG of tasks. For each stage, neighboring operators placed on identical types of containers are fused into a single operator to exploit data locality. For example, a chain of `Map` operators placed on transient containers are fused as a single `Map` operator. Such operators are expanded into a number of multiple parallel tasks, which is configured by the user or determined by the number of input data partitions. Then, edges between the operators are converted into physical data transfers between the tasks. For example, a *many-to-many* dependency can be converted into a hash-partitioned data shuffle. The tasks of the initial operators of each stage fetch data from reserved executors or storages. They can then process the data and *push* their outputs to their children operators. At the final operator of each stage, the outputs are preserved on reserved executors, and they can be later pulled by the following children stages.

3.2.3 Task Scheduler

The task scheduler in the *master* schedules and distributes tasks in the generated execution plan to reserved and transient executors. The DAG of tasks is executed stage-by-stage

in a topological order. For each stage, the task scheduler first schedules and sets up the tasks placed on reserved executors, so that they can be prepared to receive task outputs pushed from transient executors. Once they are set up, the tasks placed on transient executors are scheduled and run. Here, each of the executors is allocated with task slots, the size of which can be configured by the user. With a pluggable scheduling policy, the user can schedule each task on a particular executor with an available task slot. By default, the policy schedules tasks in a round-robin manner, while utilizing data locality information as much as possible. The policy first tries to pick an executor with the input data of the task *cached*, which we will discuss further in Section 3.2.7. If not applicable, it picks an available executor in a round-robin manner, and waits until a task slot becomes available if none of the executors are available.

3.2.4 Executor

Each executor has a user-configured number of threads for executing scheduled tasks, and thus can execute multiple tasks in parallel on separate threads. When a task on a transient executor finishes execution, it immediately notifies the master for the task scheduler to schedule a new task to the executor, without having to wait for the task output to be sent. In the meanwhile, on a separate thread, the task's output is partitioned and pushed to reserved executors that are dependent on the task. The tasks scheduled on reserved executors receive and process it, and finally preserve their outputs on its local disk for their following children stages.

3.2.5 Eviction Tolerance

Transient executors are expected to be frequently evicted during execution, which raises the following issues. First, task outputs can be partially pushed to only some of the reserved executors. To address this issue, transient executors send task output *commit* messages to the destination reserved executors through the *master* to acknowledge that all outputs are sent to them. Only after receiving the commit messages, the tasks on the reserved executors can process the outputs. This ensures that the outputs are processed exactly once. Second, we have to determine the tasks that need to be re-executed to recover lost data. As discussed previously, an eviction of a task of a particular stage never leads to recomputations of the tasks of its parent stages. Thus, the tasks of the evicted stage can be rescheduled independently and immediately upon evictions. Exploiting this property, the task scheduler reschedules only the tasks that were scheduled in the evicted executors, whose outputs were not transferred and committed to their destinations.

3.2.6 Fault Tolerance

Any container can fail due to various reasons such as hardware failures, which are very rare compared to container evictions. In case of transient executor failures, the runtime can simply use the eviction tolerance mechanisms we have just described. However, in case of reserved executor or master failures upon machine faults, the runtime needs to handle them differently. First, failures of reserved executors result in a loss of the intermediate results that were preserved on their local disk. The runtime handles this by pausing the currently executing stage, and observing its parent stages to recompute those that are necessary. Specifically, it observes the parent stages in a topological order to identify stages whose intermediate results are unavailable, to relaunch the corresponding tasks. Second, failure of the master results in a loss of the execution progress record, which includes the record of finished stages and tasks. This can be resolved by periodically replicating the progress metadata. Then, a new master can be launched to resume from the last available progress information upon machine failures.

3.2.7 Optimizations

Pado tries to keep the number of additional reserved containers as small as possible, since reserved containers are expensive as they cannot be yielded for other jobs. However, the small number of reserved executors and their limited computational resources can become a bottleneck in job executions, if they cannot handle the load that they receive. To mitigate this potential bottleneck issue, the runtime provides optimizations to reduce the load on reserved executors.

Task input caching: Tasks of operators specified by the user can cache their input data in their executor memory once the data becomes available. When the cache memory space gets full, evictions occur by the LRU policy. Moreover, as mentioned earlier, the runtime provides the cache-aware scheduling policy that distributes tasks on specific executors, in which the input data are cached. This lets tasks scheduled on transient executors to read the cached data instead of incurring data transfer from the reserved executors that they depend on or the storage that it reads from. For example, the transient tasks of the `Compute Gradient` operator in Figure 3(b) takes the latest model residing on reserved executors as their input. Without caching, the reserved executors need to send the model for every task of the operator. However, with caching, it only needs to be sent once to the executors that the tasks are allocated to.

Task output partial aggregation: Task outputs can be partially aggregated if the aggregation logic is commutative and associative [12]. Exploiting this property, on Pado, partial aggregation occurs on the outputs of the tasks allocated on the same transient executors and on the pushed data that arrive on identical reserved executors. This optimization reduces the amount of data that reserved executors receive and maintain. For example, the `Aggregate Gradients` operator in Figure 3(b) computes the sum of gradients, each of which is a vector. With partial aggregation, the number of vectors reserved executors receive is reduced, as multiple vectors computed on transient executors can be partially aggregated into a single vector before getting sent. Moreover, reserved executors only need to maintain a single vec-

tor by partially aggregating it with vectors getting pushed to them on the fly. A downside of aggregation is that data stay on transient executor for a longer time before getting sent, which increases the risk of evictions. To solve this issue, Pado can configure an upper limit for the time and the number of aggregated tasks, so that data escapes once it reaches a certain point.

4. Implementation

We have implemented Pado with around 7,000 lines of code in Java. We have integrated our implementation with big data open source projects to facilitate real-world deployments, and minimize boilerplate code.

First, our implementation can run dataflow programs written with Beam [1], an open source programming model also supported by other data processing engines like Google Cloud Dataflow [2], Flink [3], and Spark [8]. Beam programs are represented as logical DAGs of `Transforms`, which are operators for transforming one or more distributed data sets. Example `Transforms` are `ParDo` (Parallel-Do), which performs a parallel operation on each of input elements, and `Combine`, which groups all input elements by key. Given a Beam program, our implementation first identifies the data dependency type (e.g., many-to-many) of edges between `Transforms` and applies the compilation algorithms as described in Section 3.1. During the process, user-defined functions and output serializers for each `Transform` are extracted and saved as a part of a stage to be used by the runtime.

Second, our implementation can run on different resource managers like Mesos [14] and Hadoop YARN [5] using REEF [7, 26], an open source library for developing portable systems on different resource managers. In REEF, a job consists of a *Driver*, which interacts with a resource manager to obtain and manage containers, and multiple *Evaluator*s, each of which is a process running on a container managed by the *Driver*. Thus, in our implementation, Pado master runs as the *Driver*, and Pado executors run as the *Evaluator*s. Using REEF, we were able to reduce the boilerplate code that would otherwise be required to implement the low-level resource manager protocols, and to focus on developing the core runtime logic described in Section 3.2.

5. Experimental Evaluation

We evaluate Pado with three different experiments to answer the following questions:

- How Pado efficiently handles frequent evictions while aggressively collecting idle resources.
- How Pado performs with different ratios of transient to reserved containers.
- How Pado scales with more numbers of a fixed ratio of transient and reserved containers.

5.1 Experimental Setup

We describe the cluster environment, the data processing engines that we compare, and the workloads that we run on the engines for the experiments.

5.1.1 Cluster Environment

We set up a YARN cluster on AWS EC2 instances to simulate a datacenter environment. Each of the instances is used to run a transient or a reserved container. We use i2.xlarge instances (4 virtual cores, 30.5GB memory, 800GB local SSD) for reserved containers, and m3.xlarge instances (4 virtual cores, 15GB memory, double 40GB local SSDs) for transient containers. We chose instances with fast and large local SSDs to provide fast checkpointing and other disk operations.

Under our environment, reserved containers are never evicted, meaning that a job is able to use them until it voluntarily lets them go. On the other hand, transient containers are evicted according to different lifetime CDFs in Figure 1 that we acquired from analyzing the Google cluster trace. As we assume that each job in our experiments uses a small portion of total resources of the cluster, whenever an eviction occurs on a transient container, we immediately provide a new transient container with a new container lifetime.

5.1.2 Data Processing Engines

The specifications of the data processing engines we evaluate in this cluster are as follows:

Spark: Spark 2.0.0 that runs executors on both transient and reserved containers.

Spark-checkpoint: Our modified checkpointing-enabled version of Spark 2.0.0. We modified the Spark internal task scheduler and shuffle manager to implement task-level asynchronous checkpointing, in which compressed map outputs, preserved on local disks, are checkpointed by separate threads. Based on the checkpointing policy introduced in Flint [23], Spark-checkpoint selectively checkpoints intermediate data. As mentioned, works like Flint usually assume spot instances which are evicted on an *hourly or daily* basis, whereas we assume transient containers which get evicted on a *minutewise* basis. With this assumption and as data shuffle boundaries are treated as an important point to checkpoint due to the high recomputation cost, we have implemented Spark-checkpoint to checkpoint on each shuffle boundary. Spark-checkpoint runs executors on transient containers and uses reserved containers to run a non-replicated GlusterFS [4] cluster as stable storages for checkpointing. We also observed similar trends of experiment results using a non-replicated HDFS [5] cluster as stable storages.

Pado: Our Pado implementation that runs executors on both transient and reserved containers.

5.1.3 Workloads

On the engines presented above, we run three data analytics applications: Alternating Least Squares (ALS), Multino-

mial Logistic Regression (MLR), and Map-Reduce (MR). For Spark and Spark-checkpoint, we use MLlib [6] programs for ALS and MLR, and implement MR using Spark's programming API. For Pado, we implement Beam programs with the DAGs as illustrated in Figure 3. Input data are stored on AWS S3, and read by engines upon launching the job. The workloads for the applications are as shown below:

ALS: ALS is a workload with long and complex dependencies between operators, which makes it vulnerable to critical chains of cascading recomputations. We use a 10GB music ratings dataset provided by Yahoo! [10], which contains over 717M ratings of 136K songs given by 1.8M users. We set rank to 50, and run 10 iterations of the algorithm.

MLR: MLR also has long, but slightly less complex dependencies between its operators. MLR creates large amounts of intermediate data in each iteration, which can be partially aggregated into a small vector. We use a synthetic 31GB training dataset generated with a script open sourced as part of Petuum [28]. The dataset is a sparse matrix with 500K samples of 512 classes, 100K features, and 2.5B nonzero numbers. We run 5 iterations of the algorithm.

MR: MR has the shortest and simplest dependencies between operators among our workloads, and imposes the largest amount of load on reserved containers for Pado. We use a 280GB Wikipedia dump of its page view statistics [9]. The dataset consists of around a month of hourly page view counts of document. We compute the sum of page views for each of the documents over the month.

We run the experiments five times and report the averages with error bars showing standard deviations.

5.2 Eviction Rate

As discussed in Section 2.1, an effective way to increase datacenter utilization is by collecting idle resources to run transient containers. However, such containers are evicted more frequently as resources are collected more aggressively. Therefore, it is crucial for data processing engines to complete their jobs while handling frequent evictions with minimum delays. We observe the effect of different eviction rates on job completion times (JCTs) of different engines for each of the workloads.

In this experiment, we simulate datacenter environments with different *safety margins* by varying the eviction rate for transient containers with different lifetime CDFs illustrated in Figure 1 and Table 1. The CDFs show the *low*, *medium*, and *high* eviction rates. As the baseline, we also experiment without any evictions on transient containers, which is shown as the *none* eviction rate. We use 40 transient containers and 5 reserved containers to run the workloads, with an additional reserved container for the master process of the engines to run on. The numbers demonstrate the effectiveness of Pado with a relatively small number of reserved containers. We discuss the effect of different ratios of transient to reserved containers and different sizes of cluster in more depth in Section 5.3 and Section 5.4.

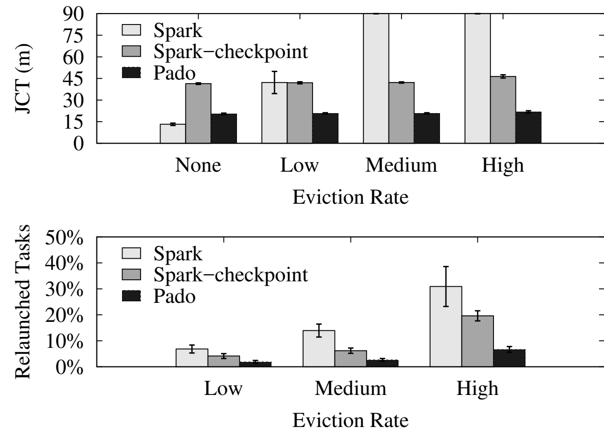

Figure 5. Job completion times, and ratio of relaunched tasks to original tasks in ALS under different eviction rates

5.2.1 Alternating Least Squares

The results of running ALS according to different eviction rates are as shown in Figure 5. Spark finishes the job in 13 minutes without any evictions, but does not finish for more than 90 minutes under the medium and high eviction rates. On the other hand, the job completion times of both Spark-checkpoint and Pado increase smoothly with higher eviction rates. Yet, Pado outperforms Spark-checkpoint at all eviction rates. Under the high eviction rate, Pado is $2.1\times$ faster than Spark-checkpoint and $4.1\times$ faster than Spark.

In Spark, task outputs are preserved on local disks and pulled by the children tasks between shuffle boundaries. Thus, an eviction of a transient container can result in a loss of intermediate results of all previous iterations. As discussed previously in Section 2.2, this creates a critical chain of cascading recomputations. For example, Spark can only relaunch the tasks of an iteration, only if it has the results of its previous iteration, and the same applies recursively. Thus, tasks of different iterations cannot be relaunched in parallel, as each of the iterations is dependent on its previous iteration. When an eviction occurs, Spark has to relaunch the tasks that output the lost data to recover from the eviction, from the initial iteration. This can delay the job greatly, as evictions can occur while running the recomputation itself, thus critical chains can repeatedly occur, further delaying the job from completion. Indeed, we found Spark recomputing identical iterations dozens of times under the high eviction rate. This makes Spark severely degrade with 31% of original tasks being relaunched under the high eviction rate.

In Spark-checkpoint, task outputs are checkpointed to stable storages on reserved containers, safe from evictions. This lets Spark-checkpoint avoid the cascading recomputations that Spark suffers from. Upon evictions, Spark-checkpoint only needs to relaunch the tasks that were running on the evicted transient containers. As a result, its job completion time marginally increases with higher eviction rates.

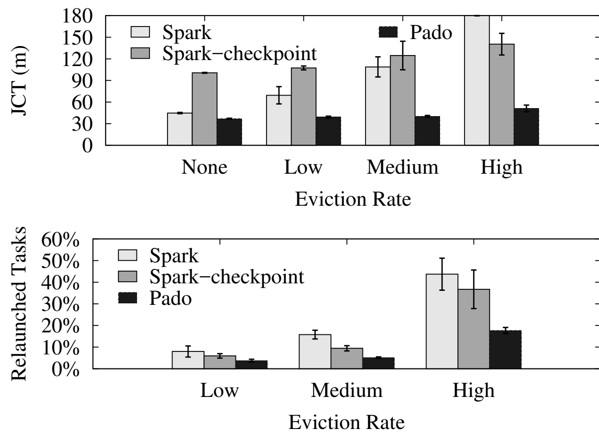

Figure 6. Job completion times, and ratio of relaunched tasks to original tasks in MLR under different eviction rates

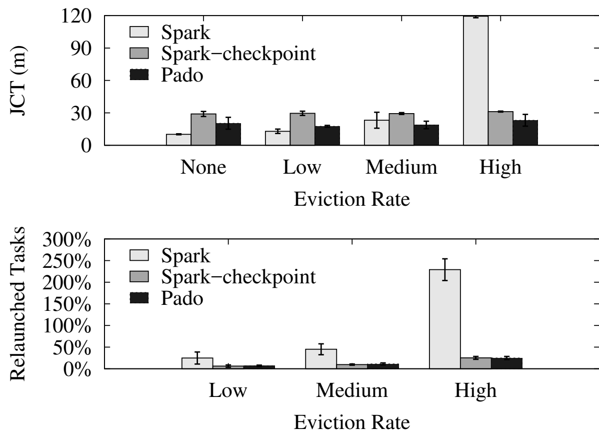

Figure 7. Job completion times, and ratio of relaunched tasks to original tasks in MR under different eviction rates

However, checkpointing incurs the overhead of transferring data back and forth with the stable storages. We found that a total of 279GB of data were checkpointed to the stable storage during the execution of the ALS workload without repetitions caused by relaunched tasks. Sending the data does not incur much overhead, since each task output can be sent independently and asynchronously. Nonetheless, fetching the data incurs a large overhead. Due to pull-based data shuffles, children tasks can only start after their parent tasks finish and checkpoint their outputs, after which the checkpointed data are pulled all at once. In Spark-checkpoint the data are served by the 5-node stable storages, whereas they are served by 45 executors in the original Spark. The reduced disk and network bandwidth slows down the data transfer and greatly increases the time to fetch the data.

In Pado, most computations are run by the tasks on transient executors, and their outputs are pushed to reserved executors to be aggregated. Thus, the aggregation occurs on reserved executors and its intermediate results are reliably retained on them, preventing cascading recomputations. For this workload, although the aggregation does not reduce the size of the data, executors can retain intermediate results within the memory. Therefore, Pado can fetch intermediate results much faster than using stable storages. This makes Pado faster than Spark-checkpoint at all eviction rates.

5.2.2 Multinomial Logistic Regression

The results of MLR are shown in Figure 6. Under the high eviction rate, Pado is $2.7\times$ faster than Spark-checkpoint and more than $3.5\times$ faster than Spark. Pado outperforms Spark-checkpoint even more compared to ALS, due to the larger amount of intermediate data created in each iteration. Each MLR iteration consists of 550 map tasks, each of which computes a gradient vector using a partition of the training data and the latest model, followed by a tree-aggregation of the vectors to update the model. The tree-aggregation is performed differently in each of the engines.

In Spark, 550 gradient vectors computed by the map tasks are preserved in the local disks of the executors. Then, each of the 22 aggregate tasks pulls $550/22$ vectors, and aggregates them into a single gradient vector. Finally, the 22 aggregated vectors are sent to the master process, which uses them to update its model. As the master process is never evicted, the critical chain does not exceed the current iteration, unlike ALS. Nonetheless, MLR iterations are much longer than ALS iterations, due to the time it takes for the gradient vector computation. Thus, the loss of preserved vectors upon evictions causes Spark to degrade severely with higher eviction rates.

In Spark-checkpoint, map task outputs are checkpointed to the stable storages on reserved containers, immediately after they are computed on transient containers. Although this prevents recomputations, each compressed map task output vector is 323MB in size, and around 173GB (323MB $*$ 550 tasks) of data have to be checkpointed in each iteration. Moreover, the data also need to be fetched back to transient containers for the following aggregate tasks. This checkpointing process requires transferring large data repeatedly, greatly delaying the work.

In Pado, gradient vectors are partially aggregated with other gradient vectors computed on the same transient container. Then, the partially aggregated vectors are pushed to aggregate tasks on eviction-free reserved containers, which prevents costly losses of the gradient vectors and task relaunches. As Pado sends less data to reserved containers with partial aggregation, it reduces the load on reserved containers. Only an average of 303 partially aggregated vectors were sent, in contrast to the 550 gradient vectors in Spark-checkpoint. Moreover, Pado does not need to transfer the data back to transient containers for aggregation. Instead, the aggregate tasks on reserved containers can directly receive the data and aggregate them into a single vector on the fly. This creates a great difference in performance since

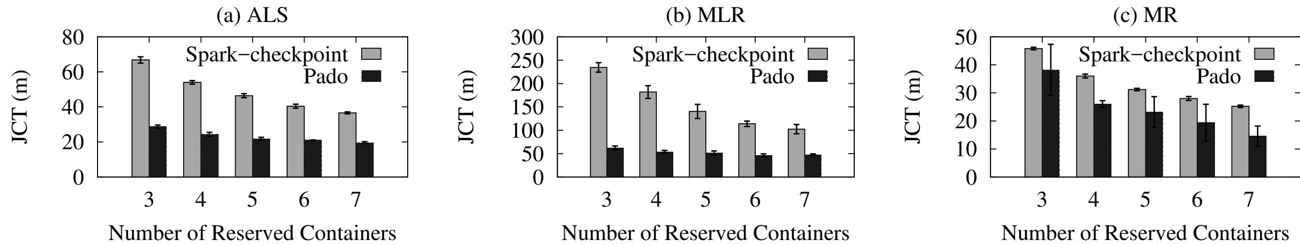

Figure 8. The job completion times of applications with different numbers of reserved containers, in addition to 40 transient containers under the high eviction rate

Spark-checkpoint has to checkpoint large amounts of data repeatedly.

5.2.3 Map-Reduce

The results of MR are shown in Figure 7. Unlike other workloads, Spark performs better than other engines up to medium eviction rate, as the short and simple dependencies make evictions less costly for Spark. However, under the high eviction rate, where we reclaim idle resources aggressively, Spark degrades significantly even with a simple MR job. Under the high eviction rate, Pado is $1.3\times$ faster than Spark-checkpoint and $5.1\times$ faster than Spark. Although Pado still outperforms Spark-checkpoint, the difference is not as great as in other workloads. The main reason is that the load on reserved containers is much heavier with MR.

In summary, Pado allows datacenters to aggressive collect transient resources from unused idle resources of over-provisioned latency-critical jobs to increase datacenter efficiency. As discussed, although the eviction rate rises with the aggressiveness of resource collection, Pado can still run various data analytic jobs under such harsh conditions.

5.3 Ratio of Transient to Reserved Containers

In this experiment, we investigate how using different ratios of transient to reserved containers affect job performance. We fix the eviction rate of 40 transient containers to the high eviction rate, and vary the number of reserved containers from 3 to 7. As Spark degrades severely with the high eviction rate with all workloads, we only compare Spark-checkpoint and Pado.

As shown in Figure 8, less reserved containers degrades the performance of both Spark-checkpoint and Pado. Spark-checkpoint degrades mainly due to the reduced throughput of stable storages, whereas Pado degrades due to the reduced throughput of reserved executors. However, the trend of the slopes vary with different workloads. For ALS(a) and MLR(b), the slope of degradation for Spark-checkpoint is much greater than that of Pado, as Pado can run in-memory processing for intermediate results, whereas Spark-checkpoint suffers from the checkpointing cost on the small number of stable storages. However, for MR(c), the slope of degradation for Pado is slightly greater than that of Spark-checkpoint, as the workload for the comparatively large Reduce operation is divided among the small number of reserved containers, whereas Spark-checkpoint distributes the work among all of its transient containers. Therefore, reducing the number of reserved containers from 7 to 3 causes Pado to slow down by around $2.6\times$ for the MR workload. Nevertheless, Pado still outperforms Spark-checkpoint under every number of reserved containers, as in the case of the MLR workload (by $3.8\times$).

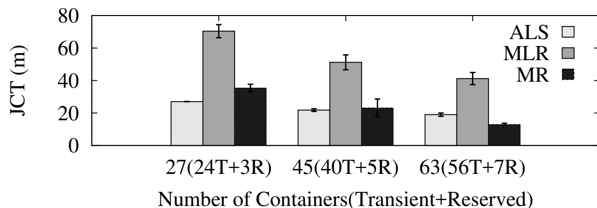

Figure 9. The job completion times of applications on Pado with different numbers of a fixed $8:1$ ratio of transient and reserved containers under the high eviction rate

To summarize, Pado can execute various data analytics workloads efficiently even when the ratio of transient to reserved containers is as high as 40:3. Thus, by using Pado, we can save reserved containers, and instead use them for other purposes, such as for running latency-critical jobs.

5.4 Scalability

In this experiment, we vary the numbers of a fixed $8:1$ ratio of transient and reserved containers to evaluate the scalability of Pado. We experiment under the high eviction rate of transient containers. As shown in Figure 9, all workloads scale on Pado with larger numbers of containers. Nonetheless, ALS scales relatively worse than the other workloads, as it is a more communication-intensive workload. Overall, this shows that Pado scales well with additional reserved and transient containers even under very frequent evictions.

6. Discussion

Pado focuses on observing the DAG and the relationships between operators to run data analytic jobs reliably under harsh conditions where evictions occur very frequently. While our

work performs well in such environments, we suggest directions in improving our system further to achieve better performances and datacenter utilization.

Datacenter Resource Scheduling: Harvest [32], a work concurrent to ours, focuses on the resource manager to solve a common goal with our system, which is to maximize datacenter utilization by using idle over-provisioned resources to run data analytic jobs. Our approach tries to overcome the frequent evictions that occur with transient resources, whereas Harvest [32] tries to minimize the number of evictions by using historic information to predict transient resource lifetimes to place them with workloads of adequate lengths. For example, it preferably schedules long jobs on transient resources that are less likely to be evicted, while scheduling short jobs on resources with short, unpredictable lifetimes. Harvest [32] and Pado tackle the problem with different aspects, and we believe that the techniques introduced in two systems are complementary. Moreover, as Pado enables workloads to run on resources with even shorter and more unpredictable lifetimes, workloads are less strictly affected by resource lifetimes. This enables resource managers to become more flexible when assigning workloads to resources of different lifetimes and enable resource managers to collect transient resources more aggressively. An interesting future research direction is to allow jobs to request resources with preferred resource lifetimes to further enhance resource managers to effectively collect and allocate idle resources to different workloads with an optimal combination of resources.

Operator Placement Optimization: With the suggested approach above, estimation of transient resource lifetimes [32] can be used to categorize resources into different lengths. Using this information, we can extend Pado to further optimize the placement algorithm to place operators on resources of different lifetimes in a more fine-grained manner. For instance, we may place the operators that are expected to have higher recomputation costs with reserved resources or those that have longer lifetimes, while placing less costly operators on resources with shorter lifetimes. This approach can further be optimized by dynamically placing and partitioning the DAG and its operators based on runtime metrics and operator statistics. Through this approach, we may place operators more optimally and better balance the load across resources with different lifetimes. For example, in the MR example illustrated in Figure 8, we can dynamically migrate work from reserved resources to transient resources with relatively long lifetimes to reduce the computational delay caused by the small number of reserved resources. By alleviating the load on reserved resources, we can also overcome workloads with deeper graphs where a larger portion of operators are placed on reserved resources due to the increased recomputation costs. Implementation and evaluation of our proposed techniques running other workloads of various depths and complexities are left as future work.

7. Related Work

Pado is designed to run dataflow programs represented as a logical DAG of operators, like other general-purpose data processing engines [12, 15, 31]. Here, each operator is scheduled as tasks and executed in parallel on multiple distributed containers. It also shares some fault-tolerance mechanisms to recover by recomputing from a certain point in the logical DAG. However, as Pado primarily focuses on harnessing transient resources in datacenters, the core runtime mechanisms, such as task scheduling and data transfer, are very different from other data processing engines.

To prevent loss of data during computations, recent works have come up with intelligent methods of checkpointing to efficiently handle data loss and interruptions. Flint [23] checkpoints the frontier of the RDD [31] lineage graph in every dynamically updated intervals. TR-Spark [29] prioritizes tasks that output the least amount of data, and performs task-level checkpointing according to resource instability. The common assumption of such works are that container evictions occur on an hourly, or on a more moderate basis, as they target spot instances. However, our goal is to use transient containers made up of the leftover idle resources reserved by LC tasks, which get evicted on a minutewise basis. Under such harsh conditions of transient resources, checkpointing has to be done very frequently, which leads to poor performances. To step away from the idea of checkpointing, Pado instead observes logical DAGs, and places a set of carefully chosen computations and the corresponding intermediate results reliably on reserved containers.

Realizing the considerable extra cost in checkpointing, there is also research on specialized processing systems that exploit domain-specific properties, like the convergence property, of particular workloads to infer the lost data [20, 22]. However, they also have limitations as they have not been designed as generic DAG processing systems, and usually give up the completeness of the result to avoid checkpointing costs. As these systems also do not target environments with frequent evictions, the completeness of their results can drop significantly, providing incorrect results and requiring more iterations to converge. On the other hand, Pado accurately executes general dataflow programs efficiently using transient containers without such restrictions and limitations.

8. Conclusion

A major problem in modern datacenters is that large amounts of resources are left idle and wasted. Running batch jobs on such transient resources increases datacenter utilization, but evictions occur very frequently on transient containers. Due to this characteristic, general data processing engines have difficulties in running jobs under such harsh conditions. They perform poorly with the cost of cascading recomputations, and incur substantial checkpointing costs, significantly slowing down the job. Pado steps away from current

approaches and focuses on the job structure to run a set of carefully selected computations, based on the relationship between dependent operators, and retain intermediate results reliably on stable reserved containers. Using the Pado Compiler with the placement and the partitioning algorithm, as well as the Runtime with several optimizations, data processing workloads can run efficiently using transient containers. Evaluation results show that Pado outperforms Spark 2.0.0 by up to $5.1\times$, and checkpoint-enabled Spark by up to $3.8\times$. We believe Pado can significantly increase datacenter utilization by efficiently using the wasted idle resources in current datacenter environments.

Acknowledgments

We thank our shepherd Evangelia Kalyvianaki and the anonymous reviewers for their insightful comments. This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No.R0126-15-1093, (SW Star Lab) Development of a Unified High-Performance Stack for Diverse Big Data Analytics).

References

[1] Apache Beam. `http://beam.incubator.apache.org`.

[2] Cloud Dataflow. `https://cloud.google.com/dataflow`.

[3] Apache Flink. `http://flink.apache.org`.

[4] GlusterFS. `https://www.gluster.org`.

[5] Apache Hadoop. `http://hadoop.apache.org`.

[6] Spark MLlib. `http://spark.apache.org/mllib`.

[7] Apache REEF. `http://reef.apache.org`.

[8] Apache Spark. `http://spark.apache.org`.

[9] Page view statistics for Wikimedia projects. `https://dumps.wikimedia.org/other/pagecounts-raw`.

[10] Yahoo! Music User Ratings of Songs with Artist, Album, and Genre Meta Information, v. 1.0. `https://webscope.sandbox.yahoo.com/catalog.php?datatype=r`.

[11] C. de Boor. *A Practical Guide to Splines*. Springer New York, 2001.

[12] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[13] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Publishing Company, New York, NY, 2009.

[14] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, 2011.

[15] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

[16] Q. Ke, M. Isard, and Y. Yu. Optimus: A dynamic rewriting framework for data-parallel execution plans. In *EuroSys*, 2013.

[17] S. Y. Ko, I. Hoque, B. Cho, and I. Gupta. On availability of intermediate data in cloud computations. In *HotOS*, 2009.

[18] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 2009.

[19] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: Improving resource efficiency at scale. In *ISCA*, 2015.

[20] M. Pundir, L. M. Leslie, I. Gupta, and R. H. Campbell. Zorro: Zero-cost reactive failure recovery in distributed graph processing. In *SOCC*, 2015.

[21] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *SOCC*, 2012.

[22] S. Schelter, S. Ewen, K. Tzoumas, and V. Markl. All roads lead to rome: optimistic recovery for distributed iterative data processing. In *ACM CIKM*, 2013.

[23] P. Sharma, T. Guo, X. He, D. Irwin, and P. Shenoy. Flint: Batch-interactive data-intensive processing on transient servers. In *EuroSys*, 2016.

[24] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *SOCC*, 2013.

[25] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *EuroSys*, 2015.

[26] M. Weimer, Y. Chen, B.-G. Chun, T. Condie, C. Curino, C. Douglas, Y. Lee, T. Majestro, D. Malkhi, S. Matusevych, B. Myers, S. Narayanamurthy, R. Ramakrishnan, S. Rao, R. Sears, B. Sezgin, and J. Wang. Reef: Retainable evaluator execution framework. In *ACM SIGMOD*, 2015.

[27] J. Wilkes and C. Reiss. ClusterData2011_2 traces. `https://github.com/google/cluster-data/blob/master/ClusterData2011_2.md`.

[28] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu. Petuum: A new platform for distributed machine learning on big data. In *ACM SIGKDD*, 2015.

[29] Y. Yan, Y. Gao, Y. Chen, Z. Guo, B. Chen, and T. Moscibroda. Tr-spark: Transient computing for big data analytics. In *SOCC*, 2016.

[30] X. Yang, S. M. Blackburn, and K. S. McKinley. Elfen scheduling: Fine-grain principled borrowing from latency-critical workloads using simultaneous multithreading. In *USENIX ATC*, 2016.

[31] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[32] Y. Zhang, G. Prekas, G. M. Fumarola, M. Fontoura, I. n. Goiri, and R. Bianchini. History-based harvesting of spare cycles and storage in large-scale datacenters. In *OSDI*, 2016.