Ph.D. DISSERTATION

A Flexible Architecture for Optimizing Distributed Data Processing

분산 데이터 처리 최적화를 위한 유연한 아키텍처

FEBRUARY 2021

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Youngseok Yang

Ph.D. DISSERTATION

A Flexible Architecture for Optimizing Distributed Data Processing

분산 데이터 처리 최적화를 위한 유연한 아키텍처

FEBRUARY 2021

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Youngseok Yang

A Flexible Architecture for Optimizing Distributed Data Processing

분산 데이터 처리 최적화를 위한 유연한 아키텍처

지도교수 전 병 곤

이 논문을 공학박사 학위논문으로 제출함

2020 년 12 월

서울대학교 대학원

컴퓨터 공학부

양 영 석

양영석의 공학박사 학위논문을 인준함

2020 년 12 월

위 원 장	엄 현 상	Eom Hyeonsang
부위원장	전 병 곤	
위 원	이 영 기	
위 원	이 재 욱	
위 원	전 명 재	

Abstract

Optimizing scheduling and communication of distributed data processing for resource and data characteristics is crucial for achieving high performance. Existing approaches to such optimizations largely fall into two categories. First, distributed runtimes provide low-level policy interfaces to apply the optimizations, but do not ensure the maintenance of correct application semantics and thus often require significant effort to use. Second, policy interfaces that extend a high-level application programming model ensure correctness, but do not provide sufficient fine control.

In this paper we propose a flexible architecture for optimizing distributed data processing. Our architecture aims to enable composable and reusable optimization policies tailored for various deployment scenarios including harnessing transient resources, performing geo-distributed data analytics, mitigating data skew, and handling large on-disk shuffle. To realize this architecture, we propose a new approach to build distributed dataflow optimization policies, and a new approach to harness transient resources in datacenters. Our evaluation results show that our flexible architecture brings performance improvements on par with existing specialized runtimes tailored for a specific deployment scenario.

Keywords: Distributed Data Processing Systems, Performance Optimization, Datacenter Resources

Student Number: 2014-22685

Contents

List of Figures

List of Tables

Chapter 1

Introduction

It is becoming increasingly important to optimize scheduling and communication for different characteristics of resources and data in distributed data processing. Examples of such characteristics widely discussed in recent literature are geographically-distributed resources [33, 53, 71, 72], cheap transient resources [57, 59, 66, 75, 77], disk-based large data shuffle [47, 56, 80], and skewed data [37, 40, 41, 55]. Researchers have shown that the existing scheduling and communication methods, unaware of these characteristics, often suffer from substantial performance degradation.

In this paper we propose a flexible architecture for optimizing distributed data processing. We aim to provide users with capabilities to adapt distributed data processing to different resource and data characteristics.

Target users: We target the following types of users. First, we target data processing application developers who are knowledgeable of their deployment environments, and wish to fine tune the performance of their applications. Second, we target cloud Data-as-a-Service (DaaS) [8, 9] developers who manage

	Number	Size (CPU, ..)	Priority	Autoscaling
Resource	O	O	O	X

	Datacenter	Rack	Node	Container	Core/cache
Placement	O	O	O	O	X

	Intra-stage	Inter-stage	Inter-job
Pipelining	O	O	X

	Key distribution	Task time	Resource usage
Statistics	O	O	X

	Parallelism	Cloning	Inter-key	Intra-key
Partitioning	O	O	O	X

	Device selection	Data selection	Eviction policy
Caching	O	O	X

Table 1.1: Capabilities of the current design of our flexible architecture for optimizing distributed data processing.

the execution of various data processing applications submitted by clients. Third, we target datacenter operators and site reliability engineers [19] who manage datacenter resources and applications that run on the resources.

Capabilities: Table 1.1 lists the capabilities of the current design of our flexible architecture. When acquiring datacenter resources, our architecture allows specifying the number, the size (e.g., CPU, memory), and also the priority [70] (e.g., transient, reserved) of resource containers. At the moment we use a fixed set of resource containers, and do not support autoscaling [8] that dynamically changes the resource configurations. For operation placement, our architecture supports placements at the levels of datacenters [33, 71, 72], racks and nodes [25, 35], and resource containers [32, 69]. However, the current design

does not support core or cache-level placement [34]. In case of operator pipelining, we support intra-stage [23] (e.g., one-to-one dependency) and inter-stage [35, 77] (e.g., shuffle dependency) pipelining, but does not support inter-job pipelining [1]. We use runtime statistics such as data key size distribution [35, 37] and task time distribution [7, 35], but does not make use of actual resource usage [21]. For partitioning, we enable configuring parallelism [7, 25, 35], cloning [7, 35], and inter-key data partitioning [35, 37]. However, our current design does not allow partitioning intra-key data [8, 21]. For caching, we allow selecting which devices to use and which data to cache, but does not allow configuring different eviction policies [7]. Our architecture can be extended to support the capabilities that are unsupported at the moment.

To realize this architecture, we propose a new approach to build distributed dataflow optimization policies, and a new approach to harness transient resources in datacenters. First, we show how to enable fine control and at the same time ensure correctness in building new dataflow optimization policies. Second, we show how to leverage the relationship between computations to reliably run the computations that are most likely to cause high recomputation costs if evicted on transient resources. In the rest of the paper, we describe the background, our two new approaches, related work, and conclusion and future directions.

Chapter 2

Background

2.1 Optimization Policy Interfaces

We first discuss in detail the existing runtime policy interfaces and application-level policy interfaces using concrete code examples. Specifically we describe the interfaces of Dryad [35] and Optimus [37].

The Dryad policy interface allows for arbitrary modifications to its directed-acyclic graph (DAG) representation of applications. In a Dryad DAG, a vertex represents a unit of work performed on a machine and an edge represents a data transfer from a vertex to another. For example, a map-reduce application can be represented in Dryad as a number of map vertices fully connected with a number of reduce vertices. The Dryad runtime coordinates the scheduling and communication of the vertices on a cluster of machines.

Figure 2.1 shows the pseudocode of two example Dryad policies [10]. Here, `ConnectionManager` is a callback-based abstraction that listens to events from the configured upstream vertices. First, `TreeAggregate` builds an aggregation

```
class TreeAggregate implements ConnectionManager {
 void onUpstreamVertexEvent(event) {
  mapVertexGroups = analyzeLocationsAndSizes(event)
  aggregateVertices = newVertices(mapVertexGroups)
  connect(mapVertexGroups, aggregateVertices)
 }
}

class Repartition implements ConnectionManager {
 void onUpstreamVertexEvent(event) {
  desiredPartitions = analyzeDataStatistics(event)
  modifyPartitionVertices(desiredPartitions)
  modifyReduceVertices(desiredPartitions)
 }
}
```

Figure 2.1: Pseudocode of Dryad policies. The Dryad policy interface provides fine control over distributed scheduling and communication, but does not ensure correctness.

tree with a goal to use network bandwidth resources more efficiently. Suppose `TreeAggregate` listens to the map vertices in a map-reduce application, to obtain the information on the locations and sizes of map vertex outputs. Using the information, `TreeAggregate` groups map vertices, creates intermediate aggregation vertices, and then connects each map vertex group to an aggregation vertex. Second, `Repartition` dynamically distributes data with a goal to handle data skew. Suppose the map-reduce application additionally has bucketizer vertices that consume sample output data from the map vertices, and partition

vertices that partition the original map vertex outputs prior to transferring the data to the reduce vertices. Then, `Repartition` can be used to monitor the bucketizer vertices, and modify the partition and reduce vertices with the goal to evenly distribute the map outputs. As shown by these examples, runtime policies can configure various scheduling and communication methods.

However, the flexibility of runtime interfaces comes at a cost: the policy developer must exercise care to ensure application correctness when developing, reusing, and composing different policies [7, 35, 37, 62]. First, the interface allows for a bug in `TreeAggregate` to miss connecting one of the map vertices to an intermediate aggregation vertex, making the optimized DAG produce partial results. Second, `Repartition` can break application semantics when applied on a random vertex in a different DAG that does not use bucketizer and partition vertices. Third, applying both `TreeAggregate` and `Repartition` on the same DAG can lead to conflicting executions that produce incorrect results. Manually building a combined policy can require a significant effort for complex policies, such as the `DrDynamicAggregateManager` in Dryad that consists of 1.3K lines of C++ code [10]. As a consequence, runtime policies have been mostly hard coded in runtimes and data processing application compilers such as the DryadLINQ compiler [37, 78], and the Hive compiler [68]. The authors of Optimus also report that their system-level optimization policies are hard-coded in the DryadLINQ compiler, maintaining the DAG property and operator semantics for the pre-defined operators in DryadLINQ [37].

In contrast to runtime interfaces, Optimus provides an application-level policy interface that ensures correctness, by restricting the interface to substituting DryadLINQ subqueries. Figure 2.2 shows the pseudocode for optimizing a matrix multiplication application described in the original Optimus paper [37]. The code defines two alternative subqueries for multiplying two matrices, and a

```
// Application code
mulA = defineMatMulSubqueryA(matrixX, matrixY)
mulB = defineMatMulSubqueryB(matrixX, matrixY)

// Policy code
stats = collectDataStatistics(matrixX, matrixY)
rewriter.registerAlternatives(stats, mulA, mulB)
```

Figure 2.2: Pseudocode of an Optimus policy. The application-level Optimus policy interface ensures correctness, but provides coarse-grained control of substituting subqueries.

policy for selecting a subquery to use for the execution. Note that as long as the two subqueries produce the same results, changing the policy code does not alter the semantics of the application. However, as this example shows, such application-level policy interfaces lack fine-grained control over scheduling and communication like selecting the types of resources to run specific computations on. The main reason is that application programming models are designed to hide distributed execution from application developers.

2.2 Transient Resources

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.

We target datacenters in which resource managers [32, 69, 70] 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 over-provisioned 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 [58, 70].

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 [32, 70]. 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* [32, 45, 70]. 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 [32].

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 [59]. 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

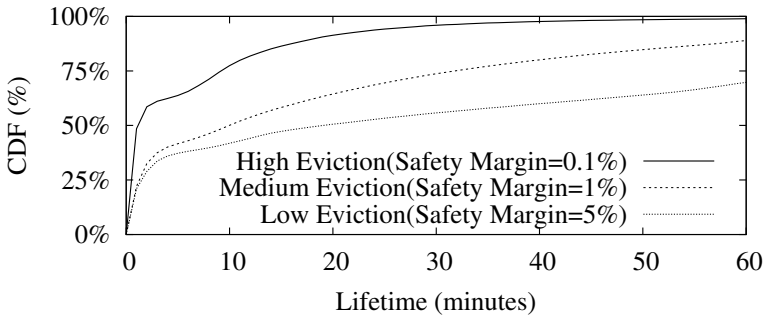

Figure 2.3: CDFs of transient container lifetimes over different safety margins.

1-minute intervals, which is commonly used for curve-fitting of experimental data [24]. Considering LC jobs as those tagged as the most latency-sensitive and the highest-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* [70]. 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 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 2.1: 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 2.2: Collected idle memory from total memory allocated to LC jobs over different safety margins. Baseline indicates collection of all idle memory.

safety margins, as depicted in Figure 2.3 and Table 2.1. 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 2.2. 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 [66, 75]. 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.

Chapter 3

Building Distributed Dataflow Optimization Policies

3.1 Overview

It is becoming increasingly important to optimize scheduling and communication for different characteristics of resources and data in distributed data processing. Examples of such characteristics widely discussed in recent literature are geographically-distributed resources [33, 53, 71, 72], cheap transient resources [57, 59, 66, 75, 77], disk-based large data shuffle [47, 56, 80], and skewed data [37, 40, 41, 55]. Researchers have shown that the existing scheduling and communication methods, unaware of these characteristics, often suffer from substantial performance degradation.

Distributed runtimes such as Dryad [35], Tez [62], and the Spark runtime [7] provide low-level interfaces to plug in computation scheduler and data channel policies to optimize for such diverse deployment scenarios. These policy interfaces have direct access to control messages and data elements, and can apply

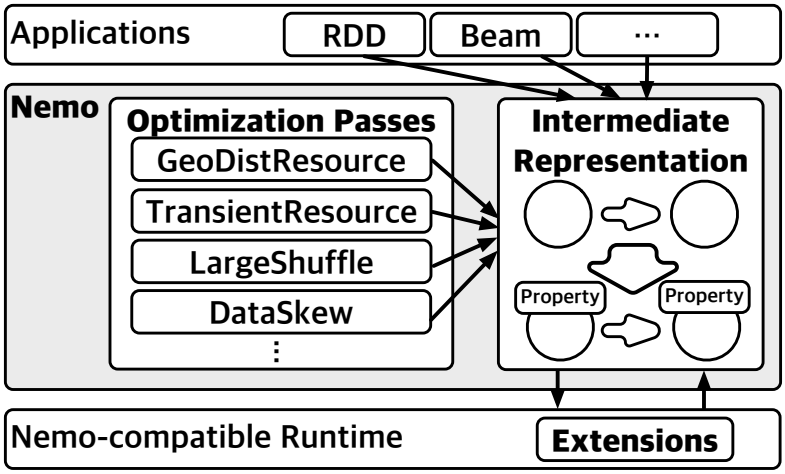

Figure 3.1: Nemo optimizes scheduling and communication of distributed data processing.

optimizations such as placing computations on specific types of resources and performing in-memory data shuffle. Unfortunately, runtime policy developers must exercise care to ensure that the policies they build and apply maintain correct application semantics. The main reason is that runtime interfaces are designed to be general, and allow for arbitrary modifications to scheduling and communication methods.

On the other hand, policy interfaces integrated with a high-level application programming model offer indirect control over runtime execution. For example, Optimus [37] integrates with the DryadLINQ programming model to enable specifying alternative DryadLINQ subqueries. This ensures correct application semantics as long as the specified subqueries compute the same results, and thus reduces the effort required to build different optimization policies. However, such application-level interfaces do not provide sufficient fine control over distributed scheduling and communication, because application programming models are designed to hide distributed execution from application developers.

To overcome the limitations of existing interfaces, we believe it is critical

to introduce a new policy interface that provides both fine control for high performance, and also ensures correct application semantics for ease of use. In this work we take a middle ground between the existing runtime and application-level interfaces. We design a policy interface that transforms an intermediate representation (IR) of applications to express indirect but fine-grained control over distributed scheduling and communication.

There are three main challenges to designing an optimization framework that embodies this middle ground approach. First, the framework should define the IR transformation methods that provide fine control and also ensure correctness. Second, the framework should enable the development of reusable and composable user-defined optimization policies that transform the IR. Third, the framework should apply the transformations of the IR in the distributed execution of the application.

Figure 3.1 depicts our Nemo optimization framework that addresses the challenges. Specifically, its IR directed-acyclic graph (DAG), optimization passes, and runtime extensions address the three challenges, respectively. Nemo integrates with high-level application programming model libraries, and compatible distributed runtimes.

First, the Nemo IR DAG represents a data processing application with vertices representing logical operations and edges representing data dependencies. To ensure that the transformed IR DAG produces the same outputs as the original IR DAG, we provide two types of transformation methods: reshaping and annotation. Reshaping methods can insert a set of utility vertices whose semantics are known to Nemo, such as a vertex that samples data. Annotation methods set execution properties of each vertex and edge to configure fine-grained scheduling and communication, such as speculative cloning and data persistence strategies. Nemo ensures correctness using the information about

the communication patterns (e.g., shuffle) of edges, and the information about the configured utility vertices and execution properties.

Second, the Nemo optimization pass abstraction enables expressing optimizations as a function that takes as input an IR DAG and calls its transformation methods. Because a pass is a simple function, different combinations of passes can be applied across different applications. We show that optimization techniques previously employed in specialized runtimes, such as Iridium [53] and Pado [77], can be expressed as optimization passes with concise lines of code.

Third, the Nemo runtime extensions integrate with the underlying runtime to apply the IR DAG transformations. Runtimes typically provide a runtime DAG abstraction to run computations on a cluster of machines [7, 35, 62]. Our scheduler extension applies various scheduling policies when scheduling the IR vertices of an IR DAG through a runtime DAG. It also rewrites the runtime DAG during job execution to apply run-time optimizations. Our data channel extension applies the optimized data communication within the runtime DAG.

We have implemented Nemo, and also a distributed runtime that is compatible with Nemo. At present, Nemo provides full support for Beam [2] applications and a subset of Spark RDD [79] applications. Our runtime integrates with REEF [73] to run on Hadoop YARN [4] and Mesos [32] clusters. We have evaluated Nemo in a cluster of Amazon EC2 instances using different optimization passes, datasets, and resource environments. Evaluation results show that each optimization pass brings performance improvements on par with existing specialized runtimes, and combinations of passes further improve performance for scenarios with a combination of different resource and data characteristics. Nemo is currently an Apache Incubator project [5].

3.2 System Design

The goal of the Nemo optimization framework is to support fine control over distributed execution of data processing applications, and at the same time maintain correct application semantics. Concretely, given a DAG representation of a data processing application with deterministic operations and a user-defined policy P where $DAG' = P(DAG)$, Nemo aims to provide the following properties.

- **Correctness**: Given the same inputs the optimized DAG' should produce the same outputs as the DAG, even when P is applied while the DAG is being executed. This ensures that the optimizations maintain correct application semantics.
- **Reusability**: The same P should be applicable to different DAGs. This enables reusing the same policy across different data processing applications, although the effects may differ between applications.
- **Composability**: If P and P' do not override optimizations specified by the other policy then enable composing different policies like $P'' = (P \circ P')$. If the policies do have a conflict, then automatically detect it for analysis. This enables distinct policies that each optimizes for a different resource or data characteristic to be incorporated into a single policy.

We show how Nemo combines an intermediate representation (IR) DAG, optimization passes, and runtime extensions to ensure these properties. First, the IR DAG provides reshaping and annotation methods for specifying optimizations (Section 3.2.1). Second, optimization passes define functions that operate on the IR DAG methods (Section 3.2.2). Third, runtime extensions apply the optimizations in the underlying runtime (Section 3.2.3).

IR DAG Reshaping: irdag.insert()		
$Relay(f: x \rightarrow x), e$	:	$V \cup \{v\}, E \setminus \{e\} \cup \{e.comm(e.src \rightarrow v), oneToOne(v \rightarrow e.dst)\}$
$Reshuffle(f: x \rightarrow x), e$	:	$V \cup \{v\}, E \setminus \{e\} \cup \{e.comm(e.src \rightarrow v), shuffle(v \rightarrow e.dst)\}$
$Sampling(f: x \rightarrow sv.f(x)), sv, rate$	:	$V \cup \{v\}, E \cup \{e.comm(e.src \rightarrow v) \vert e \in E \wedge e.dst = sv\}$
$Trigger(f: x \rightarrow udf(x)), udf, e$	:	$V \cup \{v\}, E \cup \{oneToOne(e.src \rightarrow v)\}$
(V/E = original vertex/edge set, v = inserted vertex, f = function of v, $e.comm$ = oneToOne/shuffle/broadcast)		
IR Vertex Annotation: v.set()		
$Parallelism/Integer$	:	sets the number of tasks for executing v
$SpeculativeCloning/Thresholds$	:	sets the thresholds for determining and cloning straggler tasks
$ResourceSite/Map(Index, Site)$	:	sets the geographical sites of the resources to place tasks on
$ResourcePriority/Enum(Transient)$	:	sets the priority of the resources to place tasks on
IR Edge Annotation: e.set()		
$DataFlow/Enum(Pull, Push)$	:	$e.dst$ is scheduled after $e.src$ finishes, or scheduled concurrently
$DataStore/Enum(Memory, Disk)$	:	$e.src$ tasks store output data for e in memory, or disk
$NumPartitions/Integer$	:	sets the number of partitions that $e.src$ tasks create for e
$PartitionSets/List(Set(Index))$	:	sets the partitions that each $e.dst$ task fetches for e
$Persistence/Enum(Keep, Discard)$	:	sets whether to keep or discard data after $e.dst$ processes e

Table 3.1: Example IR DAG transformation methods for optimizing scheduling and communication. Reshaping methods take as input a utility vertex and additional arguments. Annotation methods take as input a key/value execution property.

3.2.1 Intermediate Representation

The Nemo IR DAG aims to provide the desired *DAG* representation of an application. The main challenge in designing the IR DAG is defining the methods for transforming it. For Nemo to ensure the desired properties, we make explicit both the intention and the effect of the optimization for each method invocation. For example, instead of providing a single method to insert arbitrary computations, we provide multiple higher-level methods such as those specifically for increasing parallelism, speculative cloning, and sampling. We describe the IR DAG reshaping and annotation methods that embody this approach, and in particular how those methods enable ensuring correctness. We then discuss the types of applications and runtimes supported by our IR DAG design.

Transforming an IR DAG

The Nemo IR DAG represents a data processing application with vertices representing logical operations and edges representing data dependencies. When executed, an IR vertex is translated into parallel tasks that run on multiple nodes. An IR edge can be translated into key-partitioned data blocks that are produced by tasks. The initial IR DAG translated from an application, such as an RDD [79] and Beam [2] application, typically consists of vertices containing functions defined by the application, and edges with the information on communication patterns (one-to-one, shuffle, broadcast).

Table 3.1 shows example reshaping and annotation methods Nemo provides to transform the IR DAG. The reshaping methods specify a new utility vertex to insert into the IR DAG, and Nemo inserts new edges to connect the specified vertex with the existing vertices in the IR DAG. Table 3.1 specifies four utility vertices. `Relay` and `Reshuffle` simply apply an identity function to forward data from an upstream vertex to a downstream vertex, connecting with the downstream vertex with the one-to-one and the shuffle dependency, respectively. `Sampling` vertex applies the same function as an existing vertex, and consumes the same data that the existing vertex consumes. During the execution, Nemo schedules only a subset of `Sampling` tasks according to the given sampling rate. `Trigger` vertex applies a user-defined function on intermediate data. When a `Trigger` vertex executes and completes, Nemo collects the results of the user-defined function to generate a message. Nemo then halts the execution of the job, and uses the message to trigger a corresponding run-time optimization pass, which we describe in Section 3.2.2. The IR DAG also supports deleting the inserted utility vertices.

The annotation methods configure scheduling and communication of vertices

and edges by annotating specified execution properties. Table 3.1 specifies nine execution properties. For scheduling, we have execution properties for deciding how, where, and when to schedule tasks. `Parallelism` and `SpeculativeCloning` configure how many tasks to schedule. `ResourceSite` and `ResourcePriority` specify where to schedule the tasks. `DataFlow` determines whether or not to schedule source and destination tasks concurrently. For communication, we enable configuring the medium to store intermediate data with `DataStore`, the persistence method with `Persistence`, and data partitioning strategies with `NumPartitions` and `PartitionSets`. Combinations of different execution properties can express optimizations that can require significant efforts to implement with runtime policy interfaces. For example, we can configure upfront task cloning with a persistent in-memory data shuffle that pushes data eagerly from transient resources to reserved resources, through simply annotating appropriate `SpeculativeCloning`, `ResourcePriority`, `Persistence`, `DataStore`, and `DataFlow` properties on two vertices and a shuffle edge that connects them. The IR DAG also supports looking up the execution properties annotated on vertices and edges.

Ensuring Correctness

The reshaping methods ensure correctness, because Nemo connects the newly inserted utility vertex with existing vertices correctly. As shown in Table 3.1, only the outputs of the `Relay` and `Reshuffle` vertices are consumed by existing vertices, and these outputs are equivalent to the data that the existing vertices originally consumed. The other utility vertices, on the other hand, do not reach data sinks and thus do not affect the final results that the IR DAG produces. When a utility vertex is specified to be deleted, Nemo reverts appropriate changes.

The annotation methods ensure correctness through enabling Nemo to examine the configured execution properties. For each vertex in the IR DAG, Nemo checks its execution properties and the execution properties of its neighboring edges and vertices, while also examining the communication patterns of the edges. This ensure correctness because execution properties do not use and modify computation semantics [29, 36, 81] inside each vertex, and also do not have direct access to control messages and data elements in the runtime. For example, Nemo checks that the sets in the `PartitionSets` are disjoint and together contain all offsets for the `NumPartitions`, to read each partition exactly once. Nemo also checks that `PartitionSets` and `NumPartitions` are set on shuffle edges, and that vertices connected with an one-to-one edge have the same `Parallelism`. `Persistence`, for example, is not checked, because discarded intermediate data can always be recomputed from the source data when needed.

Our transformation methods ensure correctness even when invoked during the execution of the IR DAG. Because the IR DAG is decoupled from the underlying runtime, Nemo ensures correctness by controlling when to apply the transformations of the IR DAG in the runtime. Specifically, we define that a vertex is being executed when its tasks are being executed, and an edge is being executed when its source or destination vertex is being executed. First, if the transformed vertices and edges have not yet been executed, then we apply the changes immediately, such that the changes are used when they are executed. Second, if they are being executed, then we delay applying the changes until they finish execution to ensure correctness. Third, if they have already finished execution, then we apply the changes immediately, such that the changes are used when they are re-executed due to reasons such as faults.

Supported Applications and Runtimes

The current design of the IR DAG supports data processing applications that can be represented as a DAG of data-parallel and deterministic operators that process bounded data. Many real-world applications, such as Beam and RDD batch applications and also higher-level domain-specific applications like machine learning and SQL applications, meet this assumption. The current IR DAG would need to be extended to support other types of applications, such as those that have cyclic dependencies and process unbounded data [50].

The IR DAG assumes an underlying distributed runtime that supports configuring and applying utility vertices and execution properties. Existing runtimes can be enhanced to provide full support for the IR DAG optimizations through introducing additional features. For example, new data channels in addition to the existing ones (FIFO, File, TCP Pipe) can be introduced in Dryad [35] to provide support for various combinations of the `DataStore`, `DataFlow`, and `Persistence` execution properties. Similarly, a feature to dynamically add computations to a running application can be introduced in Tez [62] and the Spark runtime [7] to apply utility vertices inserted at run time.

3.2.2 Optimization Passes

Nemo optimization passes aim to provide the desired user-defined policy abstraction P. A pass is a function that receives an input IR DAG and produces a transformed IR DAG. We first describe how to develop and compose passes. We then describe how Nemo applies the given passes on the IR DAG.

Developing and Composing Passes

We describe the rationale and the algorithm for several example passes to demonstrate how to develop and compose new passes. We can write two types of passes: compile-time and run-time. Compile-time passes take as input only an IR DAG, and are run prior to job execution. Run-time passes additionally receive a message produced by a `Trigger` vertex during job execution.

Geo-distributed data analytics: We aim to cope with the low and variable capacity of WAN links when processing data that are geographically distributed [33,53,71,72]. To reduce network bottlenecks, we formulate the problem of placing computations to geographically distributed sites as a linear program (LP), similar to specialized scheduler extensions like Iridium [53]. Here, we use bandwidth information and data size estimations. We also use an off-the-shelf linear solver library, since Nemo allows using external libraries when writing a pass. The pseudocode of this algorithm is as follows.

```
CompileTimePass GeoDistPass(irdag):
 solution = solveLP(bwInfo(), sizeEstimates(irdag))
 for v in irdag.vertices:
  v.set(newResourceSite(solution.get(v)))
```

Harnessing transient resources: We aim to reduce recomputation costs when using transient resources that are cheap but frequently evicted [57,59,66, 75,77]. Based on the communication patterns, we identify operations that incur large recomputation costs and place them on reserved resources. We place the other operations on transient resources. We also quickly move intermediate data produced on transient to reserved resources. This applies key scheduling and communication optimizations employed in specialized runtimes like Pado [77]. The pseudocode of this algorithm is as follows.

```
CompileTimePass TransientResourcePass(irdag):
 for v in irdag.vertices.topologicallySorted():
  if (allOneToOneFromReserved(v.inEdges)
   || existsNonOneToOne(v.inEdges)):
   v.set(ResourcePriority.Reserved)
  else:
   v.set(ResourcePriority.Transient)
  for e in v.inEdges:
   if fromTransientToReserved(e.src, v):
    e.set(DataFlow.Push)
```

Large-scale data shuffle: We aim to reduce random disk read overheads that can grow quadratically with data size when shuffling data, similar to specialized shuffle systems like Sailfish [56] and Riffle [80]. We insert a `Relay` vertex to specify shuffling data in memory as soon as produced and writing the data as-is to a local disk. We also ensure that the in-memory data are discarded once transferred, to avoid running into out of memory errors. Following computations sequentially read the data from the local disk, after the shuffle completes. The pseudocode of this algorithm is as follows.

```
CompileTimePass LargeShufflePass(irdag):
 for e in irdag.edges.filter(isShuffleEdge()):
  rv = newRelayVertex()
  irdag.insert(rv, e)
  rv.inEdge.set(DataFlow.Push, DataStore.Memory)
  rv.inEdge.set(Persistence.Discard)
  rv.outEdge.set(DataFlow.Pull, DataStore.Disk)
```

Mitigating data skew: We aim to assign the same amount of data across parallel computations to prevent stragglers. We first set the number of partitions for the data to be shuffled. We then insert a `Trigger` vertex with a function for

obtaining the set of data partition sizes. We also ensure that the shuffle receiver is executed after the the shuffle sender and the `Trigger` vertex complete, at which point we will have obtained the statistics and optimized the execution of the shuffle receiver. The pseudocode of this algorithm is as follows.

```
CompileTimePass SkewCTPass(irdag):
 for e in irdag.edges.filter(isShuffleEdge()):
  e.set(newNumPartitions(e), DataFlow.Pull)
  irdag.insert(newOptVertex(), sizeFunction(), e)
```

At run time, when the `Trigger` vertex completes and makes available the set of size numbers, we partition the set into subsets such that the sum of the numbers in the subsets are as equal as possible. We then assign each subset to a distinct shuffle receiver task. The pseudocode of this algorithm is as follows.

```
RunTimePass SkewRTPass(irdag, message):
 subsets = partition(message)
 message.edge.set(newPartitionSets(subsets))
```

Finally, we can compose multiple passes to build an optimization policy like the following example. Registering a run-time pass requires specifying a compile-time pass that inserts `Trigger` vertices, which produce the same type of message the run-time pass uses.

```
policyBuilder.register(LargeShufflePass)
policyBuilder.register(SkewRTPass, SkewCTPass)
policy = policyBuilder.build()
```

Applying Passes

Given an IR DAG and a policy composed of passes, Nemo first applies the compile-time passes on the IR DAG in the same order as they were registered.

The optimized IR DAG output by the last compile-time pass is executed. As the execution progresses, each `Trigger` vertex completes execution and produces a message. For each message, Nemo runs the corresponding run-time pass to transform the IR DAG. Nemo runs the passes for different messages serially.

After applying each pass, Nemo checks whether the IR DAG produced by the pass is correct as described in Section 3.2.1, and also whether the pass has encountered a conflict with a previous pass. A conflict occurs when a pass overwrites the value of an execution property set by a previous pass to a different value, or deletes a utility vertex inserted by a previous pass. Nemo throws an error and refuses to execute in case of a check failure after running a compile-time pass. Upon a check failure of a run-time pass, Nemo just ignores the IR DAG output by the pass and logs the failure, as stopping an already running application can be costly.

Figure 3.2 shows how Nemo runs two example policies. Both policies first apply the `LargeShufflePass`, which inserts a `Relay` vertex between `V1` and `V3`, and annotates `E5` and `E4`. The first policy then applies the `TransientResourcePass`, which performs annotations without any conflict with the previous pass. The second policy applies the `SkewCTPass`, which inserts a `Trigger` vertex, and tries to annotate `E5` with the pull `DataFlow`. However, the `SkewCTPass` encounters a conflict as the push `DataFlow` has already been set for `E5` by the previous `LargeShufflePass`.

Fundamentally, the conflict in the second policy occurs because the `LargeShuffPass` tries to shuffle data eagerly in memory, whereas the `SkewCTPass` tries to use the statistics of the data before the downstream computations start to consume the data. If undetected, this conflict results in a pull-based in-memory data shuffle, where the outputs of all `V1` tasks are stored in memory before the `Relay` tasks start fetching the data. Although this configuration avoids disk

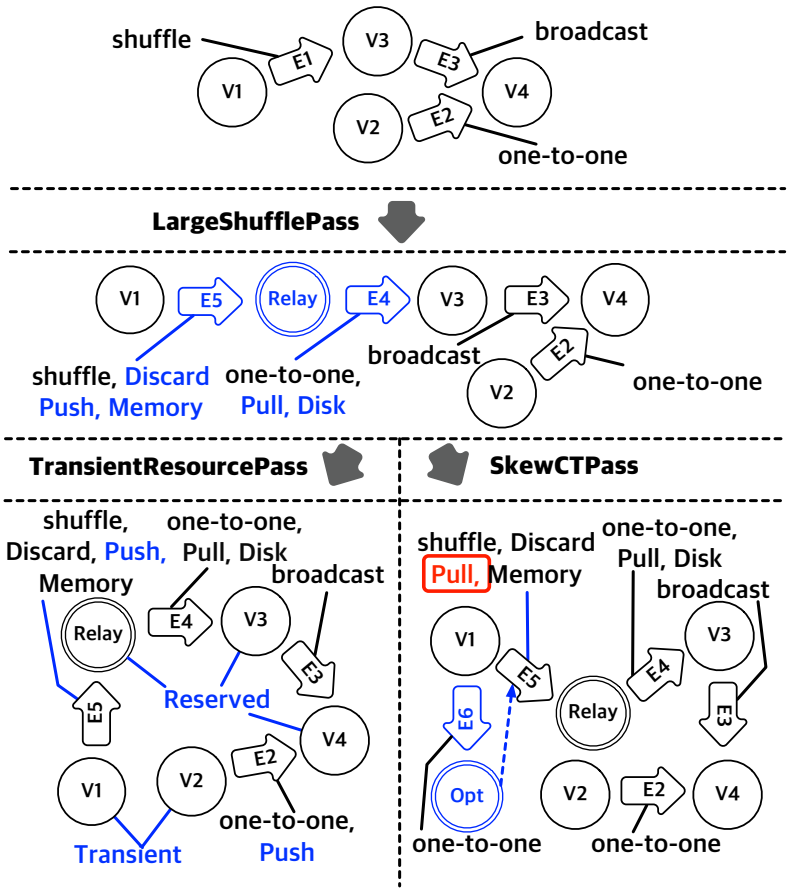

Figure 3.2: A policy composed of the `LargeShufflePass` and the `TransientResourcePass`, and another policy composed of the `LargeShufflePass` and the `SkewComplieTimePass` are applied on an input IR DAG.

seek overheads and also handles data skew at the same time, it can cause out of memory errors for large input data.

Because Nemo detects such conflicts explicitly, we can quickly address the issue. In this case, we design a new `SkewSamplingPass` that avoids the conflict with the `LargeShufflePass`. This new compile-time pass clones the IR DAG using `Sampling` vertices, and first runs the clone to obtain the statistics of sampled

data. Our third policy with the `LargeShufflePass` and the `SkewSamplingPass` can be applied together on the IR DAG to optimize for both large data shuffle and data skew. However, compared to the `SkewCTPass`, the `SkewSamplingPass` incurs the cost of executing additional vertices and using the statistics of sampled data rather than the entire data.

Next, we describe how these various transformations of the IR DAG are reflected in the distributed execution.

3.2.3 Runtime Extensions

We use a Nemo-compatible runtime depicted in Figure 3.3 to describe how the Nemo runtime extensions apply the IR DAG transformations in the distributed runtime. Upon job launch, the runtime starts a master process and executor processes on user-specified resources. In the master, the `NemoScheduler` extension operates on the task DAG abstraction that the runtime provides for scheduling tasks to executors. Executors spawn a thread to run each scheduled task, and uses the `NemoChannel` extension to communicate data between the tasks. In the rest of the section we describe how these extensions apply optimizations.

First, we set up the initial task DAG using the IR DAG optimized by compile-time passes (1). Here, we merge neighboring IR vertices into the same tasks as much as possible to minimize data communication overheads, while considering communication patterns of the IR edges and related execution properties such as the `Resource` properties and the `Parallelism` property. In case of a `Trigger` vertex, we also register a callback handler to collect the results produced by the corresponding tasks from executors as a message. Upon job start, we select candidate tasks for scheduling, which are the source tasks and their children tasks connected with the push `DataFlow` (2). For each candidate task, we select candidate executors by comparing the corresponding `Resource` properties of

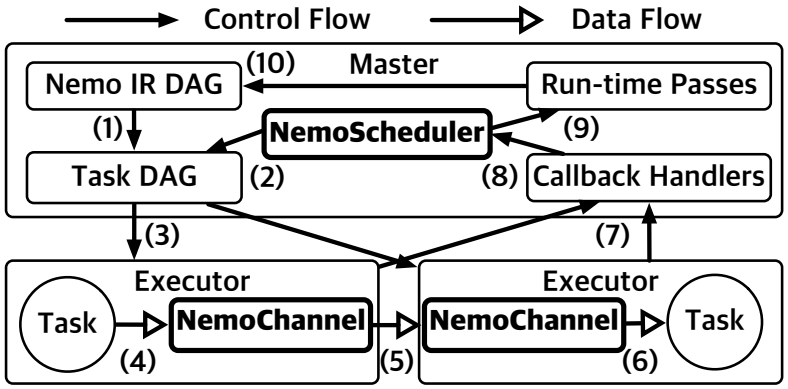

Figure 3.3: Nemo runtime extensions (bold) apply optimizations in a distributed runtime.

the task with the information on the executors. We then schedule the task to a candidate executor with the least number of running tasks (3).

When a task emits a data element, we write it to the corresponding `DataStore` implementation, creating a data block when all data elements for the channel are written (4). If the corresponding edge is shuffle, then the block is partitioned into `NumPartitions`. When a task reads input data elements, we look for the locations of the input data blocks, blocking the call when looking for blocks that are not yet available. We fetch the input data elements from the local and remote `DataStore`s, while applying `PartitionSets` for shuffle edges (5-6). Once all of the downstream tasks successfully read a block, we decide to either keep or discard the block based on the `Persistence` property.

Upon learning about task progress and executor status, we schedule new tasks, restart tasks to recover from failures and evictions, and clone tasks based on the `SpeculativeCloning` property (7-8). When a message is produced for a `Trigger` vertex, we postpone scheduling new tasks, invoke the corresponding run-time pass (9), rewrite the task DAG based on the new IR DAG output by

the run-time pass at the correct timing described in Section 3.2.1 (10), and resume scheduling.

3.2.4 New Optimizations

Nemo enables new and sophisticated optimizations with the following two techniques. First, Nemo enables new composite optimizations that combine multiple existing optimizations. Second, Nemo provides a principled approach to system extension: new utility vertices and execution properties.

To illustrate how Nemo enables new composite optimizations, we compare the lines of code written to implement some of the optimization techniques in Dryad [35] and Nemo. First, Nemo requires much fewer lines of code to implement each optimization technique compared to Dryad. Second, Nemo allows for composing various techniques while ensuring correctness, reusability, and composability, as the optimizations are expressed with pre-defined IR DAG modification methods. In contrast, developers must exercise care when composing different optimization techniques in Dryad, as the techniques in Dryad are expressed with even-driven interfaces that allow for arbitrary runtime DAG modification.

- Dryad (C++)
  - DrDynamicAggregateManager.cpp: 1215 LOC
  - DrDynamicBroadcast.cpp: 161 LOC
  - DrDynamicDistributor.cpp: 266 LOC
  - DrDynamicRangeDistributor.cpp: 98 LOC
  - DrPipelineSplitManager.cpp: 218 LOC
- Nemo (Java)

- SkewAnnotatingPass.java: 34 LOC
- SkewReshapingPass.java: 47 LOC
- TransientResourcePriorityPass.java: 42 LOC
- TransientResourceDataTransferPass.java: 43 LOC
- LargeShuffleAnnotatingPass.java: 32 LOC
- LargeShuffleReshapingPass.java: 23 LOC
- UpfrontCloningPass.java: 26 LOC

Nemo enables introducing new optimizations through adding new utility vertices and execution properties. First, new utility vertices can be added with the following guideline: outputs of the newly inserted utility vertex should not alter the outputs of existing vertices. Second, new execution properties can be added along with a property-specific correctness checker, and checkers for the dependencies between the new execution property and existing properties (i.e, modified vertex and edge, and neighboring vertices and edges). Third, corresponding runtime extensions can be added in the runtime.

3.3 Implementation

We have implemented Nemo and a distributed runtime that is compatible with Nemo in around 32K lines of Java code. Our Nemo implementation consists of the following three components similar to Musketeer [27] and LLVM [42]: frontend, optimizer, and backend.

The frontend translates applications such as Beam and RDD applications into an IR DAG (Section 3.2.1). At present, our frontend provides translation support for all Beam [2] operators, and a subset of `RDD` [79] operators such as `map`, `reduce`, `collect`, `broadcast`, and `cache`. The main reason for not fully

supporting RDDs is that the current iterator implementation used in Nemo is not readily compatible with the various `RDD` implementations. In the future we plan to modify our iterator implementation to address this limitation. The optimizer applies optimization passes on the IR DAG (Section 3.2.2). The backend configures the underlying runtime with the optimizer and the runtime extensions (Section 3.2.3).

Existing Beam applications can run on Nemo by modifying the line importing the Beam `PipelineRunner` implementation to our implementation of the runner. The frontend converts each Beam `PTransform` to an IR vertex, and `PCollection` to an IR edge. The frontend also obtains the information on communication patterns during the translation. For example, it specifies shuffle edges for the incoming `PCollection`s of the `GroupByKey` `PTransform`s.

Similar to Beam, existing RDD applications can run on Nemo with simple modifications to the lines importing the implementations of `SparkSession` and `SparkContext` to our implementations of the classes. Each RDD becomes an IR edge, and each user-defined function that generates an RDD becomes an IR vertex. Our frontend also aims to respect all of the user-specified parameters on RDDs such as parallelism and data caching, by setting the execution properties on the translated IR DAG accordingly.

Our runtime implementation is built on top of REEF [73], and consists of master and executor processes similar to the Nemo-compatible runtime described in Section 3.2.3. A REEF job consists of the driver that obtains containers from a resource manager, and evaluators that provide runtime environments on containers. To take advantage of the abstractions provided by REEF, the runtime master runs as the REEF driver and the runtime executors run as the REEF evaluators. Through the integration with REEF [73], our runtime runs on resource managers such as Hadoop YARN [4] and Mesos [32].

3.4 Experimental Evaluation

We evaluate Nemo on the following three dimensions. First, we evaluate how Nemo applies fine control under different resource and data characteristics. Second, we evaluate how different combinations of optimization passes optimize the same application. Third, we evaluate how the same Nemo policy optimizes different applications.

We run data processing applications with different combinations of following resource and data characteristics: geographically distributed resources, transient resources, large-shuffle data, and skewed data. We run each application five times, and we report the mean values with error bars showing standard deviations.

We use h1.4xlarge Amazon EC2 instances, each of which provides 16 vCPUs, 64 GiB memory, two 2 TB HDDs, and 10 Gbps network. We use different numbers of instances for different experiments. On each instance, one of the two disks is used by a Hadoop Distributed File System [4] cluster that we set up on the instances, and the other is used as a scratch disk for maintaining intermediate data. Input datasets are stored in HDFS, and fetched by the systems at the beginning of each job.

3.4.1 Fine Control

In this experiment, we evaluate how Nemo applies fine control under different resource and data characteristics. For comparison we run Spark 2.3.0 [7], because it is an open-source, state-of-the-art system. We also run a specialized runtime for each deployment scenario. Specifically, we run Iridium [53] for geo-distributed resources, Pado [77] for transient resources, and Hurricane [21] for data skew. We examine the results of Beam applications on Nemo and Pado, Spark RDD applications on Spark and Iridium, and a Hurricane application on Hurricane.

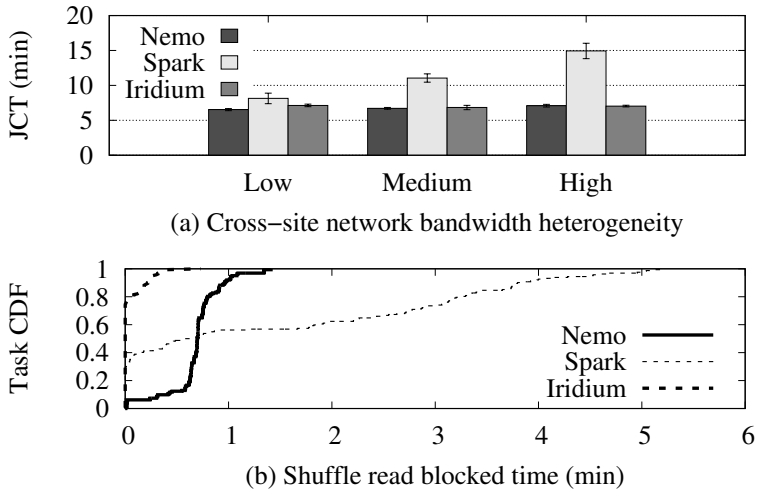

Figure 3.4: JCT for different cross-site network bandwidths, and CDF of shuffle read blocked time of tasks under the high cross-site network bandwidth heterogeneity.

We confirm that the baseline performance is comparable for Beam and basic RDD applications on Nemo. We also confirm that the baseline performance is comparable for Spark and Nemo with the `DefaultPass`, which configures pull-based on-disk data shuffle with locality-aware computation placement similar to Spark. We observe that the overhead of running the compile-time passes on Nemo is roughly 200ms. We also measure and report run-time overheads of the `Relay` vertex, `Trigger` vertex, and `SkewRTPass` in this section.

Geo-Distributed Resources: To set up geo-distributed resources and heterogeneous cross-site network bandwidths, we use Linux Traffic Control [12] to control the network speed between instances, as described in Iridium [53]. Each site is configured with 2Gbps uplink network speed, and a specific downlink network speed between 25Mbps and 2Gbps. We experiment with Low, Medium, and High bandwidth heterogeneity with the fastest downlink outperforming the

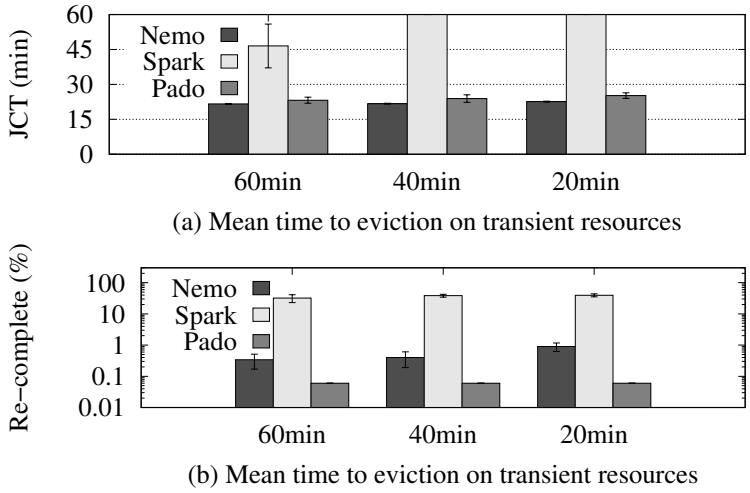

Figure 3.5: JCT and ratio of re-completed tasks to original tasks for different mean times to eviction on transient resources.

slowest downlink by 10×, 41×, and 82×. With this, we use 20 EC2 instances as resources scattered across 20 sites. To evaluate data shuffle under heterogeneous network bandwidths, we use a workload that joins two partitions of 373GB Caida [15] network trace dataset and computes network packet flow statistics.

The job completion time (JCT) of Iridium, Spark, and Nemo optimized with the `GeoDistPass`, are shown on Figure 3.4 (a). Spark degrades significantly with larger bandwidth heterogeneity, since tasks that fetch data through slow network links become stragglers. In contrast, Iridium and Nemo are stable across different network speeds. Figure 3.4 (b) shows that the cumulative distributive function (CDF) of shuffle read time has a long tail for Spark compared to Iridium and Nemo. Iridium and Nemo show comparable performance with similar largest shuffle read blocked times, although Iridium shows overall better shuffle read blocked times using a more sophisticated linear programming model.

Transient Resources: Based on existing works [66, 75, 77], we classify

resources that are safe from eviction as reserved resources and those prone to eviction as transient resources. We set up 10 EC2 instances for providing transient resources and 2 instances for reserved resources. When an executor running on transient resources is evicted, we allow the system to immediately re-launch a new executor using the transient resources to replace the evicted executor as described in Pado [77]. To evaluate handling long and complex DAGs with transient resources, we run an Alternating Least Squares [39] (ALS) workload, an iterative machine learning recommendation algorithm, on 10GB Yahoo! Music user ratings data [17] with over 717M ratings of 136K songs given by 1.8M users. We use 50 ranks and 15 iterations for the parameters. By varying the mean time to eviction for transient resources, we show how systems deal with the different eviction frequencies. The distribution of the time to eviction is approximated as an exponential distribution, similar to TR-Spark [75].

Figure 3.5 (a) shows the JCT of Pado, Spark and Nemo optimized with the `TransientResourcePass` for different mean times to eviction. With the 40-minute and 20-minute mean time to eviction, Spark is unable to complete the job even after running for an hour, at which point we stop the job. The main reason is heavy recomputation of intermediate data across multiple iterations of the ALS algorithm, which is repeatedly lost in recurring evictions. On the other hand, Nemo and Pado successfully finish the job in around 20 minutes, as both systems are optimized to retain a set of selected intermediate data on reserved resources. Figure 3.5 (b) shows the ratio of re-completed tasks to original tasks for different mean times to eviction. It shows that Nemo and Pado re-complete significantly fewer tasks compared to Spark, leading to a much shorter JCT. Nemo and Pado show comparable performance although Nemo re-completes more tasks, because the tasks that both systems re-complete are executed quickly and do not cause cascading recomputations of parent tasks.

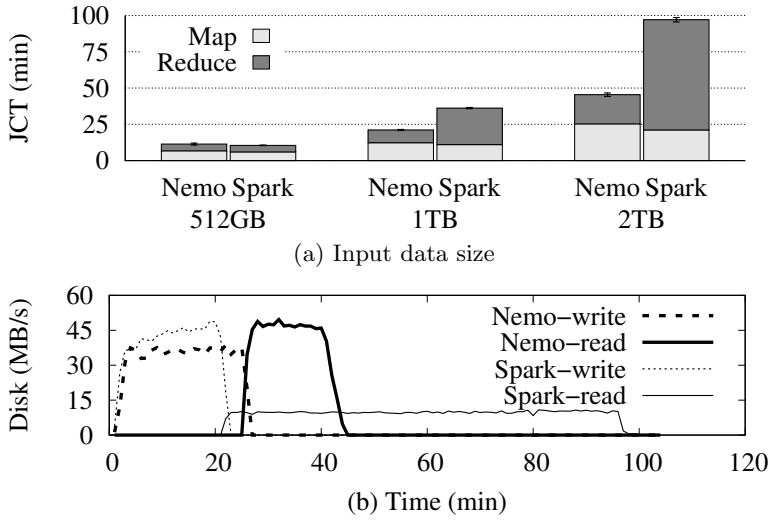

Figure 3.6: JCT for different input data sizes, and mean throughput of scratch disks for maintaining intermediate data when processing the 2TB input data.

Large-Shuffle Data: We evaluate how Nemo and Spark handle large shuffle operations using 512GB, 1TB, and 2TB data of the Wikimedia pageview statistics [13] from 2014 to 2016, as the datasets provide sufficiently large amount of real-world data. We use a Map-Reduce application that computes the sum of pageviews for each Wikimedia project. We choose the ratio of map to reduce tasks to 5:1, similar to the ratios used in Riffle [80] and Sailfish [56], and use 20 EC2 instances to run the workload.

The JCT of Spark and Nemo optimized with the `LargeShufflePass` are shown on Figure 3.6 (a). Both show comparable performance for the 512GB dataset, but Nemo outperforms Spark with larger datasets. To understand the difference, we measured the mean throughput of the disks used for intermediate data. Figure 3.6 (b) illustrates the mean disk throughput of scratch disks used for intermediate data when running the 2TB workload. Here, a spike in the

write throughput is followed by a spike in the read throughput, which illustrates disk writes during the map stage followed by disk reads during the reduce stage while performing the shuffle operation. For Spark, the disk read throughput during the reduce stage is as low as about 10 MB/s, indicating severe disk seek overheads. In contrast, the throughput is as high as 45 MB/s for Nemo, as the `LargeShufflePass` enables sequential read of intermediate data by the following reduce tasks, which minimizes the disk seek overhead.

To measure the overhead of the `Relay` vertex inserted by the `LargeShuffPass` before the reduce operation, we have also run the 2TB workload on Nemo without the `LargeShufflePass`. The reduce operation begins 56 seconds earlier without the `LargeShufflePass` and the `Relay` vertex, where 56 seconds represent 2.05% of the JCT of Nemo with the `LargeShufflePass`.

Skewed Data: To experiment with different degrees of data skewness, we generate synthetic 200GB key-value datasets with two different key distributions: Zipf and Top10. For the Zipf distribution, we use parameters 0.8 and 1.0 with 1 million keys [21]. Datasets with Top10 distribution have heaviest 10 keys that represent 20% and 30% of the total data size. We run a Map-Reduce application that computes the median of the values per key on 10 EC2 instances. Because this application is non commutative-associative, for evaluating Hurricane we use an approximation algorithm similar to Remedian [60] to fully leverage its task cloning optimizations [21]. The Hurricane application also uses 4MB data chunks and uses its own storage to handle input and output data, similar to the available example application code.

Figure 3.7 (a) shows the JCT of Hurricane, Spark, and Nemo optimized with the `SkewCTPass` and the `SkewRTPass`. Performance of Spark degrades significantly with increasing skewness. Especially, Spark fails to complete the job with the 1.0 Zipf parameter, due to the load imbalance in reduce tasks

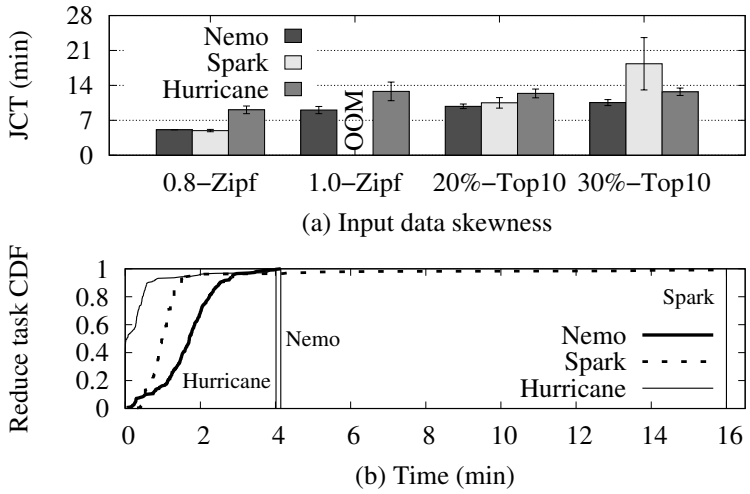

Figure 3.7: JCT for different input data skewness, and CDF of reduce task completion time when processing the 30%-Top10 skewed data. Each vertical line in the CDF graph denotes the completion time of the slowest reduce task.

with skewed keys which leads to out-of-memory errors. In contrast, both Nemo and Hurricane handle data skew gracefully. In particular, Nemo achieves high performance, and at the same computes medians correctly without using an approximation algorithm.

Figure 3.7 (b) shows the CDF of reduce task completion time when processing the 30%-Top10 dataset. The CDF for Spark shows that reduce tasks with popular keys take a significant amount of time to finish compared to other tasks. In contrast, the slowest task completes much quicker for Hurricane and Nemo. We have observed short-lived tasks alongside with longer tasks in Hurricane with its task cloning optimization, and longer tasks with balanced completion times for Nemo with its data repartitioning optimization.

To measure the overhead of the `Trigger` vertex inserted by the `SkewCTPass`, we also run the 30%-Top10 workload on Nemo without the `SkewCTPass` and the

`SkewRTPass`. The reduce operation begins 35 seconds earlier without the `Trigger` vertex, where 35 seconds represent 5.52% of the JCT of Nemo configured with the `SkewCTPass` and the `SkewRTPass`.

These results for each deployment scenario show that each optimization pass on Nemo brings performance improvements on par with specialized runtimes tailored for the specific scenario.

3.4.2 Composability

We now evaluate combinations of different optimization passes. Table 3.2 summarizes the results.

Skewed Data on Geo-distributed Resources: In this experiment, we use the same 1.0-Zipf workload for the skew handling experiment in Section 3.4.1, because the workload showed the largest load imbalance. We use 10 EC2 instances representing geo-distributed sites with heterogeneous network speed in between 25Mbps to 2Gbps. Here, DP and GDP run into out-of-memory errors due to the reduce tasks with skewed keys that are requested to process excessively large portions of data. SKP and GDP+SKP both successfully complete the job with the skew handling technique in SKP, but GDP+SKP outperforms SKP by also benefiting from the scheduling optimizations in GDP.

Large Shuffle on Transient Resources: For this experiment, we use the same 1TB workload for the large shuffle experiment in Section 3.4.1, to use sufficiently large data that incurs disk seek overheads. In this case, we use 10 reserved instances and 10 transient instances with the 20-minute mean time to eviction setting.

Most notably, DP and LSP fail to complete even after 100 minutes, at which point we stop the job, and TP runs into out-of-memory errors. We have observed that heavy recomputation caused by frequent resource eviction significantly slows

Skewed data on Geo-distributed	**Large Shuffle on Transient**	**Large Shuffle with Skewed**
DP: OOM	DP: 100m	DP: OOM
GDP: OOM	TP: OOM	LSP: OOM
SKP: 27.2m	LSP: 100m	SSP: OOM
GDP + SKP: 14.9m	TP + LSP: 48.2m	LSP + SSP: 31.4m

Table 3.2: JCT when using different combinations of `DefaultPass` (DP), `GeoDistPass` (GDP), `SkewCTPass` (SKP), `TransientResourcePass` (TP), `LargeShufflePass` (LSP), and `SkewSamplingPass` (SSP).

down the DP and LSP cases. We have also found out that the LSP optimization makes the application much more vulnerable to resource evictions compared to DP. The main reason is that with LSP, eviction of a single receiving task in the shuffle boundary leads to the entire recomputation of the sending tasks of the shuffle operation, to completely re-shuffle the intermediate data in memory. In contrast, DP does not need to recompute shuffle sending tasks whose output data are not evicted and stored in local disks. TP by itself also is not sufficient, as it leads to out-of-memory errors while pushing large shuffle data in memory from transient resources to reserved resources.

TP+LSP is the only case that successfully completes the job by leveraging both optimizations in TP and LSP. With TP+LSP, the job pushes the shuffle data from transient to reserved resources, and also streams them to local disks on reserved resources that are safe from evictions. This allows TP+LSP to handle frequent evictions on transient resources, and also to utilize disks for storing large shuffle data with minimum disk seek overheads. However, TP+LSP incurs the overhead of using only half of the resources (transient or reserved) for each end of the data shuffle. As a result, the JCT for TP+LSP with transient

resources is around twice the JCT for LSP without using transient resources, which is displayed in Section 3.4.1. Nevertheless, we believe that this overhead is worthwhile, taking into account that transient resources are much cheaper than reserved resources from the perspective of datacenter utilization [59, 77].

Large Shuffle with Skewed Data: For this experiment, we generate a synthetic key-value dataset with a skewed key distribution that is around 1TB in size, as the datasets used in Section 3.4.1 for skew handling are not sufficiently large to incur disk seek overheads. This dataset has the distribution where heaviest 20 keys represent 30% of the total data size. Using this dataset, we run the same application that we have used for the skewed data experiment in Section 3.4.1 on 20 EC2 instances.

In this experiment, only SSP+LSP successfully completes the job, whereas all other cases run into out-of-memory errors. DP and LSP fails to complete the job, due to particular tasks assigned with excessively large portions of data, incurring out-of-memory errors. SSP by itself also runs into out-of-memory errors although it repartitions data across the receiving tasks of the shuffle boundary. We have observed that with large data size, the absolute size of the heaviest keys is significantly larger compared to smaller scale experiments with skewed data shown in Section 3.4.1. Without the LSP optimization, this problem is combined with random disk read overheads that degrade the running time of the shuffle receiving tasks, leading to out-of-memory errors. In contrast, SSP+LSP successfully completes the job by leveraging both of the optimizations from SSP and LSP.

These various results confirm that Nemo can apply combinations of distinct optimization passes to further improve performance for deployment scenarios with a combination of different resource and data characteristics.

3.4.3 Reusability

Finally, we evaluate how the same Nemo policy optimizes different applications. In addition to different applications used in prior experiments, we apply the policies on several ad-hoc BeamSQL [2] TPC-H [16] queries (Q) with different scale factors (SF), as they are widely used for benchmarking distributed data processing systems. Here, 1 SF is approximately 1GB of input data. We specifically use workloads that handle smaller input and intermediate data compared to the previous experiments, and thus are much less affected by the issues that occur in the specific scenarios like disk-seek overheads and resource evictions.

First, using 20 nodes with the `LargeShufflePass`, we observe 20.8 minute JCT for SF1000 Q3 that is 25% smaller than the JCT without the optimization, but no significant performance improvements for SF1000 Q14. We also observe 41.1 minute JCT for SF3000 Q12 that brings 22% performance improvements. Second, we do not observe meaningful performance improvements for SF100 Q4 and Q13 with the `SkewCTPass` on 10 nodes, as the dataset is not skewed. Finally, using 8 transient nodes with the 10-minute expected eviction rate and 2 reserved resources, we apply the combination of the `TransientResourcePass` and the `LargeShufflePass` on SF100 Q4 and Q14. For the respective queries, we observe JCTs of 8.2 minutes and 3.4 minutes, which are smaller than when not applying the optimizations by 9% and 15%.

These results as well as the results of different workloads in previous experiments confirm that the same optimization passes on Nemo can speed up different workloads instantly, with varying degrees of effectiveness.

3.5 Discussion

Nemo provides a programming interface for building correct, reusable, and composable optimization policies. We discuss several directions to extend the interface and further facilitate the development of new policies.

Ensuring resource constraints: Although Nemo provides execution properties to specify where to place computations and data, Nemo relies on the runtime to determine the actual resources to acquire. To ensure that the resource constraints are met in the execution, we can incorporate the information into the IR DAG on the resource availability and acquisition.

Declaring optimizations ahead of time: To enable compile-time analysis of run-time pass conflicts and optimizations, we can provide the option to declare intended optimizations ahead of time. For example, we can receive more explicit information on the predicates (e.g., is a shuffle edge) and actions (e.g., store in memory) that a run-time pass intends to use.

Leveraging historical information: We can enable passes to use information on previous executions of the same application, and employ more sophisticated techniques such as machine learning to determine how to transform the IR DAG. To facilitate this, we can maintain a database that stores the information of the executed IR DAGs and their performance metrics, and provide an interface for passes to access the information in the database.

3.6 Summary

This chapter presented Nemo, an optimization framework that provides fine control over distributed scheduling and communication of data processing applications, and at the same time ensures correct application semantics. We hope Nemo serves as a platform for dataflow optimization research and development.

Nemo is available at https://nemo.apache.org.

Chapter 4

Harnessing Transient Resources in Datacenters

4.1 Overview

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 [45, 70].

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 [45, 70, 76]. 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 [59] shows that evictions can occur only a few minutes after the batch jobs are newly allocated with transient resources.

Many distributed data processing engines [25, 35, 79] 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 [66] and TR-Spark [75] 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 alleviate 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 [2], a programming model initially developed by the Google Dataflow [8] team, and runs on various datacenter resource managers like Mesos [32] and Hadoop YARN [4] by using the REEF library [73].

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 [59]. The results show that under a high rate of evictions, Pado outperforms Spark 2.0.0 [7] by up to 5.1× and checkpoint-enabled Spark, which encompasses ideas proposed by Flint [66], by up to 3.8×. Using Pado, datacenters can greatly increase utilization through effectively running batch jobs using wasted idle resources aggressively collected from datacenters.

4.2 System 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.

4.2.1 Design Overview

A plethora of distributed data processing engines [25,35,79] 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 4.1 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 4.1(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

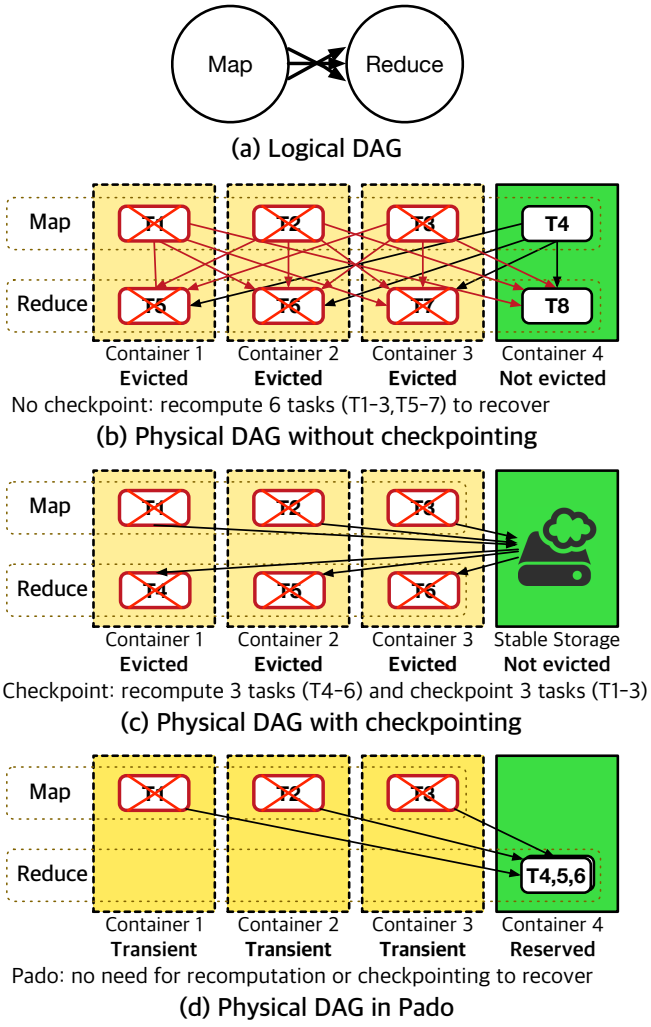

Figure 4.1: A Map-Reduce job's logical(a) and physical DAG representation in existing data processing engines, without(b) and with(c) checkpointing, as well as in Pado (d). We consider a case where transient containers 1 to 3 are evicted while running the `Reduce` operator. The arrows indicate dependencies of tasks, and red arrows indicate those of the tasks that must be relaunched upon evictions.

processing engines without 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 4.1(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 [37, 38].

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 [66] and TR-Spark [75] 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, that observes and processes the DAG prior to the execution, along with a runtime 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.

4.2.2 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 subpado/graphs/ 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.

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

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:        end if
13:    else if op.inEdges=∅ then                               ▷ Source Operator
14:        if op.isRead then
15:            op.mark(transient)
16:        else if op.isCreated then
17:            op.mark(reserved)
18:        end if
19:    end if
20: end for
```

of multiple parent tasks, similar to Reduce tasks in the Map-Reduce example in Section 4.2.1. 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.

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

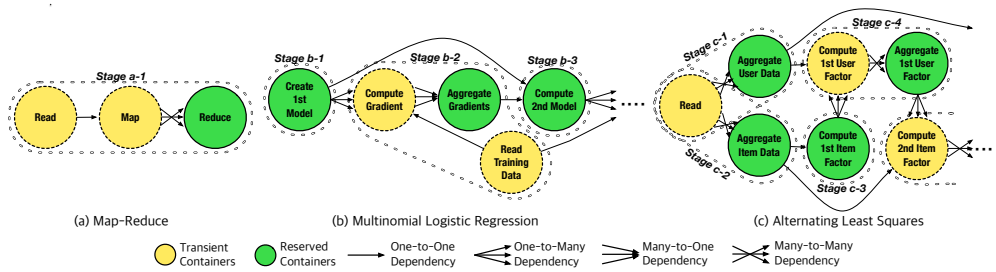

Figure 4.2: Compilation results of different workloads. Operators with complex dependencies with parent operators are placed on reserved containers, and all stages finish on operators placed on reserved containers.

Partitioning

To facilitate the execution and to easily keep track of the job progress, the compiler partitions the marked logical DAG into subpado/graphs/ 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

Algorithm 2 Logical DAG Partitioning Algorithm

```
1: Input: Logical DAG op-placed-dag
2: Output: DAG of Pado Stages stages
3: for op ∈ topologicalSort(op-placed-dag) do
4:     if op.isMarked(reserved) or op.outEdges=∅ then
5:         currStage := stages.newStage()
6:         recursiveAdd(currStage, op)
7:     end if
8: end for
9:
10: function recursiveAdd(currStage, op)
11:     currStage.add(op)
12:     for all parentOp ∈ op.inEdges do
13:         if parentOp.isMarked(transient) then
14:             recursiveAdd(currStage, parentOp)
15:         else if parentOp.isMarked(reserved) then
16:             parentOp.stage.addChild(currStage)
17:         end if
18:     end for
19: end function
```

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.

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 4.2 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 4.2(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 [31]. 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 4.2(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 [39]. It alternates its computation and aggregation between user and item factors, and its compilation result is illustrated in Figure 4.2(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.

4.2.3 Runtime

The Pado Runtime receives and efficiently executes the DAG of Pado Stages with several techniques. As illustrated in Figure 4.3, the runtime consists of the *Pado master* that orchestrates the distributed workload, and multiple *Pado executors* 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. 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

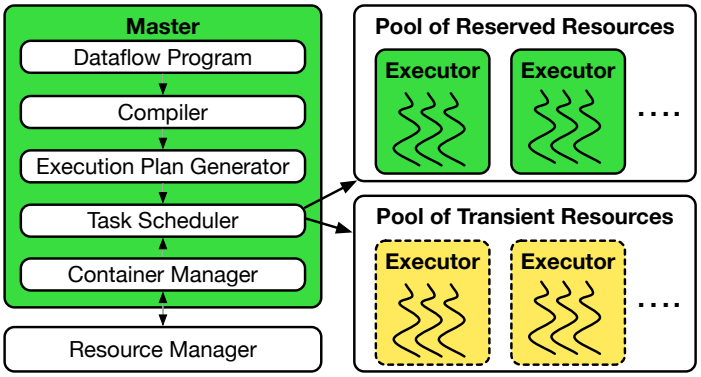

Figure 4.3: Pado Runtime

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.

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.

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.

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 4.2.3. 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.

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.

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.

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.

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 4.2(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 [25]. 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 Gra-`

dients operator in Figure 4.2(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 vector 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.

4.3 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 [2], an open source programming model also supported by other data processing engines like Google Cloud Dataflow [8], Flink [3], and Spark [7]. Beam programs are represented as logical DAGs of `Transform`s, which are operators for transforming one or more distributed data sets. Example `Transform`s 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 `Transform`s and applies the compilation algorithms as described in Section 4.2.2. 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 [32] and Hadoop YARN [4] using REEF [6, 73], 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 4.2.3.

4.4 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.

4.4.1 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.

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 2.3 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.

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 [66], Spark-checkpoint selec-

tively 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 [11] cluster as stable storages for checkpointing. We also observed similar trends of experiment results using a non-replicated HDFS [4] cluster as stable storages.

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

Workloads

On the engines presented above, we run three data analytics applications: Alternating Least Squares (ALS), Multinomial Logistic Regression (MLR), and Map-Reduce (MR). For Spark and Spark-checkpoint, we use MLlib [14] 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 4.2. 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! [17], 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 [74]. 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 [13]. 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.

4.4.2 Eviction Rate

As discussed in Section 2.2, 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 2.3 and Table 2.1. 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 4.4.3 and Section 4.4.4.

Alternating Least Squares

The results of running ALS according to different eviction rates are as shown in Figure 4.4. 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× faster than Spark-checkpoint and 4.1× 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 4.2.1, 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

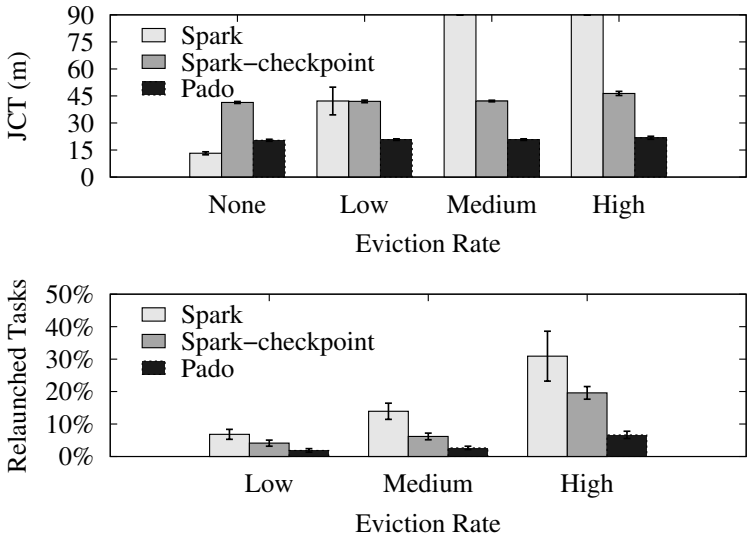

Figure 4.4: Job completion times, and ratio of relaunched tasks to original tasks in ALS under different eviction rates

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.

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.

Multinomial Logistic Regression

The results of MLR are shown in Figure 4.5. Under the high eviction rate, Pado is 2.7× faster than Spark-checkpoint and more than 3.5× 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

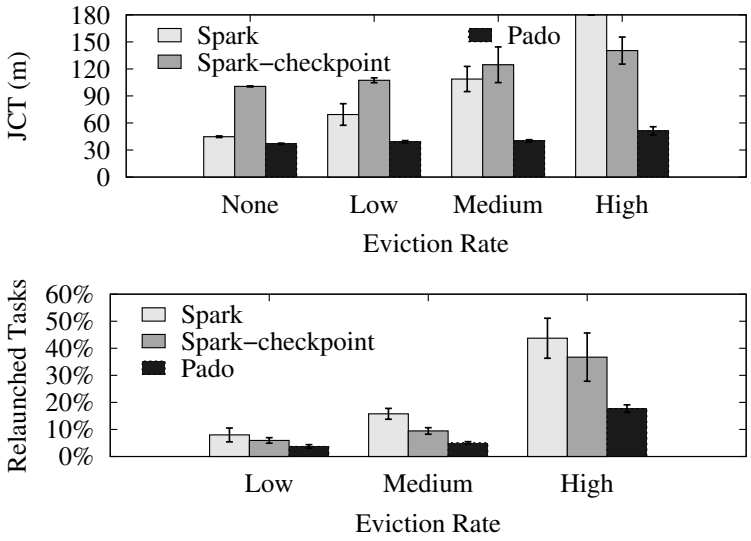

Figure 4.5: Job completion times, and ratio of relaunched tasks to original tasks in MLR under different eviction rates

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.

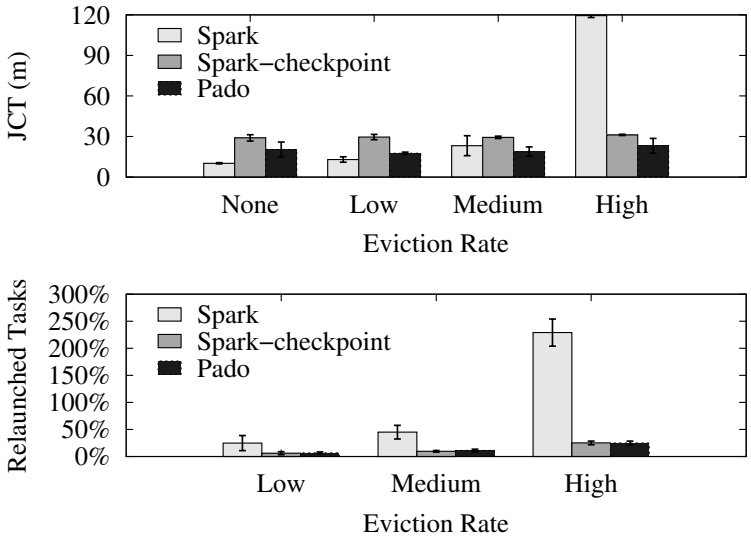

Figure 4.6: Job completion times, and ratio of relaunched tasks to original tasks in MR under different eviction rates

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 Spark-checkpoint has to checkpoint large amounts of data

repeatedly.

Map-Reduce

The results of MR are shown in Figure 4.6. 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× faster than Spark-checkpoint and 5.1× 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.

4.4.3 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 4.7, 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

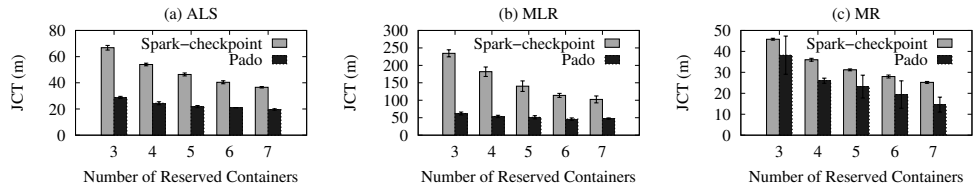

Figure 4.7: The job completion times of applications with different numbers of reserved containers, in addition to 40 transient containers under the high eviction rate

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× 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×).

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.

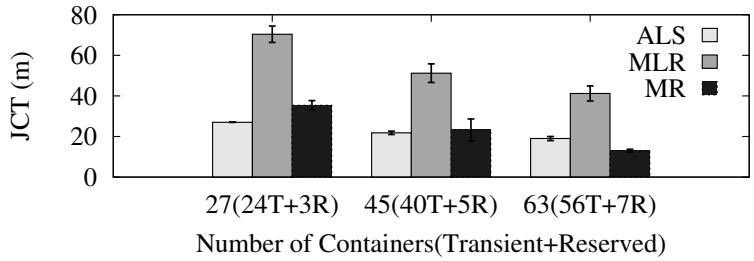

Figure 4.8: The job completion times of applications on Pado with different numbers of a fixed 8 : 1 ratio of transient and reserved containers under the high eviction rate

4.4.4 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 4.8, 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.

4.5 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 [82], 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 [82] 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 [82] 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 [82] 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 4.7, 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 pado/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.

4.6 Summary

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×, and checkpoint-enabled Spark by up to 3.8×. We believe Pado can

significantly increase datacenter utilization by efficiently using the wasted idle resources in current datacenter environments.

Chapter 5

Related Work

5.1 Dataflow Optimization Approaches

Nemo builds on many years of research in dataflow processing, relational database, and compiler optimizations. Nevertheless, we believe the set of trade-offs we have chosen to design the IR DAG, optimization passes, and runtime extensions for optimizing distributed dataflow processing makes Nemo a unique system.

Dataflow processing: Nemo differentiates itself from the existing application-level [37] and runtime-level [7, 35, 37, 62] approaches to dataflow scheduling and communication optimizations by taking a middle ground approach. Nemo provides a policy interface that transforms an intermediate representation (IR) of applications to express indirect but fine control over distributed scheduling and communication.

Our decoupled system design and our DAG-based IR are similar to Musketeer [27]. However, our work is complementary to Musketeer, as we focus on

providing fine control over physical scheduling and communication in our IR, whereas Musketeer focuses on dynamically mapping its IR to a range of different execution runtimes.

The SparkSQL Catalyst optimizer [18] takes as input a SparkSQL application and outputs a Spark RDD application, which Nemo can take as input. Compared to Nemo, Catalyst has more information about application semantics (e.g., 'Add' '1' and '2'), but has less fine control over scheduling and communication (e.g., speculative task cloning).

Recently proposed dynamic query optimizers [44, 48] for distributed dataflow processing runtimes operate on high-level logical plans for SQL queries. Leveraging the semantics of SQL queries and the runtime information, these optimizers focus on choosing an optimal logical plan, for example by finding an optimal join order. Nemo operates on a lower-level IR DAG that supports general dataflow processing applications, and provides the methods to configure scheduling and communication methods of each data-parallel operation in the applications.

Weld [52] takes as input code that composes imperative libraries such as Pandas [49] and Numpy [64], creates a combined Weld IR program, and outputs optimized assembly code using LLVM. Weld can reduce data movement overheads across such imperative libraries, but it is not designed to optimize distributed scheduling and communication like Nemo.

Relational databases: Many of the optimizations in Nemo, such as parallelization and distributed scheduling optimizations, can be traced to research in parallel databases [26, 28]. Nemo enables expressing and composing various types of such optimizations for distributed dataflow processing applications, by introducing a policy interface that provides fine control and at the same time ensures correctness.

Our idea of annotating operators with execution properties is similar to using

query hints in relational databases to influence the optimizer [22]. Nevertheless, these works focus on restricting the search space of SQL query execution plans, whereas Nemo focuses on tuning the scheduling and communication of dataflow processing applications.

Compilers: Our approach of expressing optimizations as passes that transform an IR is similar to LLVM [42]. However, in contrast to the LLVM IR that represents assembly code, the Nemo IR explicitly captures the dependencies and the communication patterns of coarse-grained, data-parallel operations. This enables passes on Nemo to express various distributed scheduling and communication optimizations.

Verified compilers, such as CompCert [43], aim to ensure the correctness of optimized assembly code using formal verification methods. Nemo aims to ensure the correctness of optimized distributed execution of dataflow processing applications, by introducing utility vertices and execution properties that make it simple to ensure correctness.

5.2 Optimizing for Transient Resources

Pado is designed to run dataflow programs represented as a logical DAG of operators, like other general-purpose data processing engines [25, 35, 79]. 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 [66] checkpoints the frontier of the RDD [79] lineage graph in every dynamically updated intervals. TR-Spark [75] 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 [54, 63]. 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.

Chapter 6

Conclusion and Future Directions

6.1 Conclusion

We presented a flexible architecture for optimizing distributed data processing. We first showed how to enable fine control and at the same time ensure correctness in building new dataflow optimization policies. We then showed how to leverage the relationship between computations to reliably run the computations that are most likely to cause high recomputation costs if evicted on transient resources. We hope our flexible architecture serves as a platform for dataflow optimization research and development.

6.2 Future Directions

6.2.1 Shared Resources

Nemo operates with node-level and task-level granularities for data locality and job statistics. First, Nemo uses node-level, rack-level, and datacenter-level

localities. Second, Nemo uses execution properties that leverage task time (`SpeculativeCloning`), resource availability (`ResourceLocality`), and cross-datacenter bandwidth (`ResourceSite`).

However, recent works on core and memory sharing within each machine have enabled new configuration and optimization knobs for data processing jobs. First, Shenango [51] enables fine granularity core reallocation at every 5 microseconds. Second, Elfen scheduling [76] introduces the nanonap syscall to stop the batch thread execution without yielding its SMT lane to the OS scheduler. Third, SAM [67] colocates tasks that share data and distributes tasks with high cache capacity and memory bandwidth behaviors.

We can extend Nemo to better leverage these recent works on core and memory sharing. First, we can introduce a new vertex execution property (`CoreRelocation`) that enables cross-core and cross-socket relocation. For example, we can set thresholds for triggering relocation based on statistics such as thread and packet queues, SMT utilization, intra and inter socket coherence, memory bandwidth utilization, and remote access (NUMA). Second, we can extend the current `Trigger` vertex to be triggered during vertex execution, rather than triggered after a vertex execution. Third, we can enable dynamic optimizations based on hardware statistics using the new execution property and the extended `Trigger` vertex. For example, we can create new policies that combine core-level locality optimizations with existing optimizations that leverage node-level, rack-level and datacenter-level locality.

6.2.2 New Hardware and Architectures

Nemo currently supports common hardware used in datacenters. For compute, Nemo supports CPUs. For storage, Nemo supports, through the `DataStore` execution property, memory, disk (HDD), and distributed filesystem. For network,

Nemo supports Ethernet.

However, a number of recent works have focused on enabling using new hardware and architectures in datacenters. First, INSIDER [61] enables effective in-storage computing (ISC) with virtual file abstraction which abstracts ISC as file operations. Second, Octopus [46] proposes a RDMA-enabled distributed memory filesystem that closely couples non-volatile memory (NVM) and remote direct memory (RDMA). Third, GPU-accelerated incremental HDFS proposed by Shredder [20] provides content-based chunking instead of fixed-size chunking which enables reusing previous map task results. Fourth, LegoOS [65] proposes a disaggregated operating system for disaggregated hardware using loosely-copuled monitors. Fifth, the work on energy-efficient ultra-low latency SSD [30] shows that the idle power consumption is larger for older generation storages, and writes are more power-hungry than reads for newer generation storages.

We can extend Nemo to leverage these new opportunities to accelerate computation, increase cost efficiency, and reduce power consumption. First, for compute, we can introduce new utility vertices to accelerate computation. We can introduce the `Chunking` vertex that supports GPU-accelerated chunking. We can also introduce the `ISC` vertex to support offloading computations to ISC, for example through interpreting Beam and Spark user-defined functions. Second, for storage, we can introduce new `DataStore` execution property options. For example, we can introduce options for ISC-supported devices, RDMA-enabled memory filesystems, optane SSDs and SSDs, and CPU cache for disaggregated operating systes. Third, we can introduce new optimization passes that aim for energy-efficiency. For example, we minimize idle time through pipelining when using HDDs, and apply in-memory shuffle and tree-aggregate optimizations before writing to minimize data writes when using optane SSDs. Fourth, we can combine the use of new hardware and architectures with existing optimization

in Nemo. For example, we can opt to not consider data source locality and fully utilize cores, when using RDMA-enabled memory filesystems.

Bibliography

[1] Apache Airflow. https://airflow.apache.org.

[2] Apache Beam. https://beam.apache.org.

[3] Apache Flink. https://flink.apache.org/.

[4] Apache Hadoop. https://hadoop.apache.org.

[5] Apache Nemo. https://nemo.apache.org.

[6] Apache REEF. http://reef.apache.org.

[7] Apache Spark. https://spark.apache.org.

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

[9] Databricks. https://databricks.com/.

[10] Dryad Research Prototype. https://github.com/MicrosoftResearch/Dryad.

[11] GlusterFS. https://www.gluster.org.

[12] Linux Traffic Control. https://lartc.org/manpages/tc.txt.

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

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

[15] The CAIDA Anonymized Internet Traces 2016 Dataset. https://www.caida.org/data/passive/passive_2016_dataset.xml.

[16] TPC-H. http://www.tpc.org/tpch.

[17] 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.

[18] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark sql: Relational data processing in spark. In *ACM SIGMOD*, 2015.

[19] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. *Site Reliability Engineering: How Google Runs Production Systems.* " O'Reilly Media, Inc.", 2016.

[20] Pramod Bhatotia, Rodrigo Rodrigues, and Akshat Verma. Shredder: Gpu-accelerated incremental storage and computation. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, 2012.

[21] Laurent Bindschaedler, Jasmina Malicevic, Nicolas Schiper, Ashvin Goel, and Willy Zwaenepoel. Rock you like a hurricane: Taming skew in large scale analytics. 2018.

[22] Nicolas Bruno, Surajit Chaudhuri, and Ravishankar Ramamurthy. Power hints for query optimization. In *ICDE*, 2009.

[23] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. Flumejava: Easy, efficient data-parallel pipelines. In *PLDI*, 2010.

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

[25] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[26] D. J. Dewitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. I. Hsiao, and R. Rasmussen. The gamma database machine project. *IEEE Trans. on Knowl. and Data Eng.*, 1990.

[27] Ionel Gog, Malte Schwarzkopf, Natacha Crooks, Matthew P. Grosvenor, Allen Clement, and Steven Hand. Musketeer: All for one, one for all in data processing systems. In *EuroSys*, 2015.

[28] Goetz Graefe. Encapsulation of parallelism in the volcano query processing system. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, 1990.

[29] Zhenyu Guo, Xuepeng Fan, Rishan Chen, Jiaxing Zhang, Hucheng Zhou, Sean McDirmid, Chang Liu, Wei Lin, Jingren Zhou, and Lidong Zhou. Spotting code optimizations in data-parallel pipelines through periscope. In *OSDI*, 2012.

[30] Bryan Harris and Nihat Altiparmak. Ultra-low latency ssds' impact on overall energy efficiency. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*, 2020.

[31] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer Publishing Company, New York, NY, 2009.

[32] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, 2011.

[33] Kevin Hsieh, Aaron Harlap, Nandita Vijaykumar, Dimitris Konomis, Gregory R Ganger, Phillip B Gibbons, and Onur Mutlu. Gaia: Geo-distributed machine learning approaching lan speeds. In *NSDI*, 2017.

[34] Eunji Hwang, Hyungoo Kim, Beomseok Nam, and Young-ri Choi. Cava: exploring memory locality for big data analytics in virtualized clusters. In *CCGRID*. IEEE, 2018.

[35] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

[36] Eaman Jahani, Michael J. Cafarella, and Christopher Ré. Automatic optimization for mapreduce programs. *Proc. VLDB Endow.*, 2011.

[37] Qifa Ke, Michael Isard, and Yuan Yu. Optimus: A dynamic rewriting framework for data-parallel execution plans. In *EuroSys*, 2013.

[38] Steven Y. Ko, Imranul Hoque, Brian Cho, and Indranil Gupta. On availability of intermediate data in cloud computations. In *HotOS*, 2009.

[39] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 2009.

[40] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. Skew-resistant parallel processing of feature-extracting scientific user-defined functions. In *SOCC*, 2010.

[41] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. Skewtune: Mitigating skew in mapreduce applications. In *ACM SIGMOD*, 2012.

[42] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, 2004.

[43] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 2009.

[44] Youfu Li, Mingda Li, Ling Ding, and Matteo Interlandi. Rios: Runtime integrated optimizer for spark. In *SOCC*, 2018.

[45] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *ISCA*, 2015.

[46] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: an rdma-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017.

[47] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. Themis: Fair and efficient GPU cluster scheduling. In *17th USENIX Sym-*

posium on Networked Systems Design and Implementation (NSDI 20), 2020.

[48] Kshiteej Mahajan, Mosharaf Chowdhury, Aditya Akella, and Shuchi Chawla. Dynamic query re-planning using qoop. In *OSDI*, 2018.

[49] Wes McKinney et al. Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference*, 2010.

[50] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *SOSP*, 2013.

[51] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019.

[52] Shoumik Palkar, James J Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, Matei Zaharia, and Stanford InfoLab. Weld: A common runtime for high performance data analytics. In *CIDR*, 2017.

[53] Qifan Pu, Ganesh Ananthanarayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Paramvir Bahl, and Ion Stoica. Low latency geo-distributed data analytics. In *ACM SIGCOMM*, 2015.

[54] Mayank Pundir, Luke M. Leslie, Indranil Gupta, and Roy H. Campbell. Zorro: Zero-cost reactive failure recovery in distributed graph processing. In *SOCC*, 2015.

[55] Smriti R. Ramakrishnan, Garret Swart, and Aleksey Urmanov. Balancing reducer skew in mapreduce workloads using progressive sampling. In *SOCC*, 2012.

[56] Sriram Rao, Raghu Ramakrishnan, Adam Silberstein, Mike Ovsiannikov, and Damian Reeves. Sailfish: A framework for large scale data processing. In *SOCC*, 2012.

[57] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *SOCC*, 2012.

[58] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *SOCC*, 2012.

[59] Charles Reiss, John Wilkes, and Joseph L. Hellerstein. Google cluster-usage traces: format + schema. Technical report. `https://github.com/google/cluster-data`.

[60] Peter J Rousseeuw and Gilbert W Bassett Jr. The remedian: A robust averaging method for large data sets. *Journal of the American Statistical Association*, 1990.

[61] Zhenyuan Ruan, Tong He, and Jason Cong. INSIDER: Designing in-storage computing system for emerging high-performance drive. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019.

[62] Bikas Saha, Hitesh Shah, Siddharth Seth, Gopal Vijayaraghavan, Arun Murthy, and Carlo Curino. Apache tez: A unifying framework for modeling and building data processing applications. In *ACM SIGMOD*, 2015.

[63] Sebastian Schelter, Stephan Ewen, Kostas Tzoumas, and Volker Markl. All roads lead to rome: optimistic recovery for distributed iterative data processing. 2013.

[64] SciPy.org. NumPy. `https://www.numpy.org`.

[65] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. Legoos: A disseminated, distributed OS for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018.

[66] Prateek Sharma, Tian Guo, Xin He, David Irwin, and Prashant Shenoy. Flint: Batch-interactive data-intensive processing on transient servers. In *EuroSys*, 2016.

[67] Sharanyan Srikanthan, Sandhya Dwarkadas, and Kai Shen. Data sharing or resource contention: Toward performance transparency on multicore systems. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, 2015.

[68] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu, and Raghotham Murthy. Hive-a petabyte scale data warehouse using hadoop. In *ICDE*, 2010.

[69] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *SOCC*, 2013.

[70] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *EuroSys*, 2015.

[71] Raajay Viswanathan, Ganesh Ananthanarayanan, and Aditya Akella. Clarinet: Wan-aware optimization for analytics queries. In *OSDI*, 2016.

[72] Ashish Vulimiri, Carlo Curino, P. Brighten Godfrey, Thomas Jungblut, Jitu Padhye, and George Varghese. Global analytics in the face of bandwidth and regulatory constraints. In *NSDI*, 2015.

[73] Markus Weimer, Yingda Chen, Byung-Gon Chun, Tyson Condie, Carlo Curino, Chris Douglas, Yunseong Lee, Tony Majestro, Dahlia Malkhi, Sergiy Matusevych, Brandon Myers, Shravan Narayanamurthy, Raghu Ramakrishnan, Sriram Rao, Russel Sears, Beysim Sezgin, and Julia Wang. Reef: Retainable evaluator execution framework. In *ACM SIGMOD*, 2015.

[74] Eric P. Xing, Qirong Ho, Wei Dai, Jin Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. Petuum: A new platform for distributed machine learning on big data. In *ACM SIGKDD*, 2015.

[75] Ying Yan, Yanjie Gao, Yang Chen, Zhongxin Guo, Bole Chen, and Thomas Moscibroda. Tr-spark: Transient computing for big data analytics. In *SOCC*, 2016.

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

[77] Youngseok Yang, Geon-Woo Kim, Won Wook Song, Yunseong Lee, Andrew Chung, Zhengping Qian, Brian Cho, and Byung-Gon Chun. Pado: A data processing engine for harnessing transient resources in datacenters. In *Proceedings of the Twelfth European Conference on Computer Systems*, 2017.

[78] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, 2008.

[79] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.

[80] Haoyu Zhang, Brian Cho, Ergin Seyfe, Avery Ching, and Michael J Freedman. Riffle: optimized shuffle service for large-scale data analytics. In *EuroSys*, 2018.

[81] Jiaxing Zhang, Hucheng Zhou, Rishan Chen, Xuepeng Fan, Zhenyu Guo, Haoxiang Lin, Jack Y. Li, Wei Lin, Jingren Zhou, and Lidong Zhou. Optimizing data shuffling in data-parallel computation by understanding user-defined functions. In *NSDI*, 2012.

[82] Yunqi Zhang, George Prekas, Giovanni Matteo Fumarola, Marcus Fontoura, Íñigo Goiri, and Ricardo Bianchini. History-based harvesting of spare cycles and storage in large-scale datacenters. In *OSDI*, 2016.

요약

분산 데이터 프로세싱의 스케쥴링과 커뮤니케이션을 리소스와 데이터 특성에 맞추어 최적화 하는 것은 높은 성능을 달성하는데 매우 중요하다. 기존 최적화 방식은 크게 두가지 카테고리로 나누어진다. 첫째, 분산 런타임들은 최적화를 적용하기 위한 로우 레벨 정책 인터페이스를 제공하지만, 올바른 애플리케이션 시멘틱의 보장을 하지 않기때문에, 사용하는데 큰 노력을 필요하게 한다. 둘째, 하이 레벨 애플리케이션 프로그래밍 모델을 확장하는 정책 인터페이스들은 올바른 시멘틱 보장을 하지만, 세밀한 컨트롤을 충분하게 제공하지 못한다.

본 논문에서 분산 데이터 처리 최적화를 위한 유연한 아키텍처를 제안한다. 우리의 유연한 아키텍처는 조합 가능하고 재사용 가능한, 다양한 실행환경에 맞춘 최적화 정책 개발을 가능하게 하는 것을 목표로 한다. 예를 들어서 일시적 자원 활용, 지리적 분산 데이터 분석, 데이터 스큐 처리, 디스크를 활용한 큰 데이터 셔플 등 실행환경이 있다. 유연한 아키텍처를 실현하기 위하여 우리는 분산 최적화 정책을 개발하는 새로운 방식 및 일시적 자원을 활용하는 새로운 방식을 제안한다. 우리의 유연한 아키텍처가 특정 실행 환경에 최적화된 기존 특수 런타임들에 가까운 성능 개선을 제공함을 실험 결과를 통해 보였다.

주요어: 분산 데이터 처리 시스템, 성능 최적화, 데이터센터 자원
학번: 2014-22685