THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Apache Nemo: A Framework for Building Distributed Dataflow Optimization Policies

Youngseok Yang and Jeongyoon Eo, *Seoul National University;* Geon-Woo Kim, *Viva Republica;* Joo Yeon Kim, *Samsung Electronics;* Sanha Lee, *Naver Corp.;* Jangho Seo, Won Wook Song, and Byung-Gon Chun, *Seoul National University*

https://www.usenix.org/conference/atc19/presentation/yang-youngseok

This paper is included in the Proceedings of the 2019 USENIX Annual Technical Conference.

July 10–12, 2019 • Renton, WA, USA

ISBN 978-1-939133-03-8

Apache Nemo: A Framework for Building Distributed Dataflow Optimization Policies

Youngseok Yang[1] Jeongyoon Eo[1] Geon-Woo Kim[2] Joo Yeon Kim[3]
Sanha Lee[4] Jangho Seo[1] Won Wook Song[1] Byung-Gon Chun[1*]
[1]*Seoul National University* [2]*Viva Republica* [3]*Samsung Electronics* [4]*Naver Corp.*

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.

We describe Apache Nemo, an optimization framework for distributed dataflow processing that provides fine control for high performance, and also ensures correctness for ease of use. We combine several techniques to achieve this, including an intermediate representation, optimization passes, and runtime extensions. Our evaluation results show that Nemo enables composable and reusable optimizations that bring performance improvements on par with existing specialized runtimes tailored for a specific deployment scenario.

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 [19, 33, 44, 45], cheap transient resources [37, 38, 42, 47, 48], disk-based large data shuffle [35, 36, 51], and skewed data [22, 24, 25, 34]. 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 [20], Tez [40], and the Spark runtime [4] 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 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.

*Corresponding author.

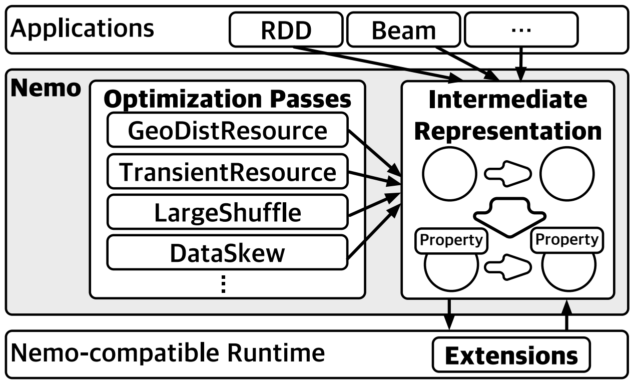


Figure 1: Nemo optimizes scheduling and communication of distributed data processing.

On the other hand, policy interfaces integrated with a high-level application programming model offer indirect control over runtime execution. For example, Optimus [22] 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 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 [33] and Pado [48], 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 [4, 20, 40]. 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 [1] applications and a subset of Spark RDD [50] applications. Our runtime integrates with REEF [46] to run on Hadoop YARN [2] and Mesos [18] 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 [3].

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

2 Background

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 [20] and Optimus [22].

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 shows the pseudocode of two example Dryad policies [5]. Here, `ConnectionManager` is a callback-based abstraction that listens to events from the configured upstream vertices. First, `TreeAggregate` builds an aggregation 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.

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

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

Figure 3: Pseudocode of an Optimus policy. The application-level Optimus policy interface ensures correctness, but provides coarse-grained control of substituting subqueries.

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 [4, 20, 22, 40]. 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 [5]. As a consequence, runtime policies have been mostly hard coded in runtimes and data processing application compilers such as the DryadLINQ compiler [22,49], and the Hive compiler [43]. 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 [22].

In contrast to runtime interfaces, Optimus provides an application-level policy interface that ensures correctness, by restricting the interface to substituting DryadLINQ subqueries. Figure 3 shows the pseudocode for optimizing a matrix multiplication application described in the original Optimus paper [22]. The code defines two alternative subqueries for multiplying two matrices, and a 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.

3 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 *DAG*s. 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 3.1). Second, optimization passes define functions that operate on the IR DAG methods (Section 3.2). Third, runtime extensions apply the optimizations in the underlying runtime (Section 3.3).

3.1 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.

IR DAG Reshaping: irdag.insert()	$Relay(f: x \to x), e$	:	$V \cup \{v\}, E \setminus \{e\} \cup \{e.comm(e.src \to v), oneToOne(v \to e.dst)\}$
	$Reshuffle(f: x \to x), e$	:	$V \cup \{v\}, E \setminus \{e\} \cup \{e.comm(e.src \to v), shuffle(v \to e.dst)\}$
	$Sampling(f: x \to sv.f(x)), sv, rate$	:	$V \cup \{v\}, E \cup \{e.comm(e.src \to v) \vert e \in E \wedge e.dst = sv\}$
	$Trigger(f: x \to udf(x)), udf, e$	:	$V \cup \{v\}, E \cup \{oneToOne(e.src \to v)\}$
	(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 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.

3.1.1 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 [50] and Beam [1] 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 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 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. 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 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.

3.1.2 Ensuring Correctness

The reshaping methods ensure correctness, because Nemo connects the newly inserted utility vertex with existing vertices correctly. As shown in Table 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 [17, 21, 52] 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.

3.1.3 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 [31].

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 [20] 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 [40] and the Spark runtime [4] to apply utility vertices inserted at run time.

3.2 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.

3.2.1 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 [19, 33, 44, 45]. 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 [33]. 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 [37, 38, 42, 47, 48]. 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 [48]. 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 [35] and Riffle [51]. 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()
```

3.2.2 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.

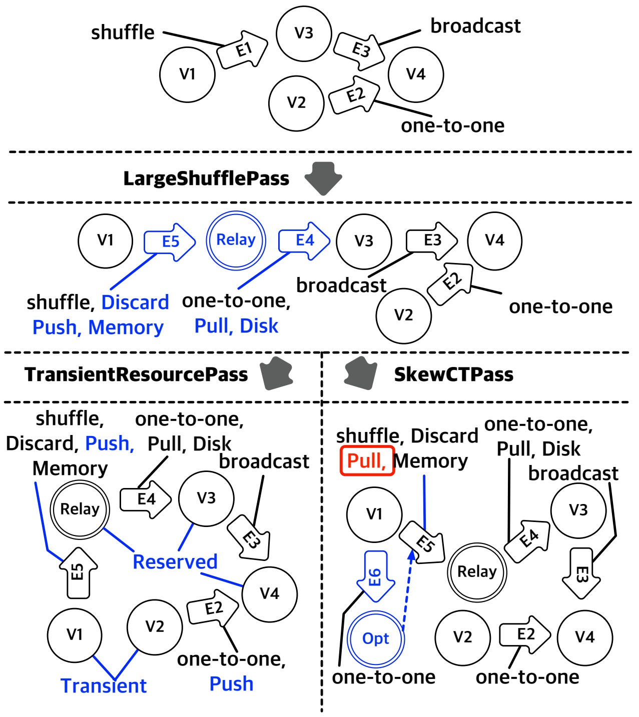


Figure 4: A policy composed of the LargeShufflePass and the TransientResourcePass, and another policy composed of the LargeShufflePass and the SkewCompileTimePass are applied on an input IR DAG.

After applying each pass, Nemo checks whether the IR DAG produced by the pass is correct as described in Section 3.1.2, 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 4 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

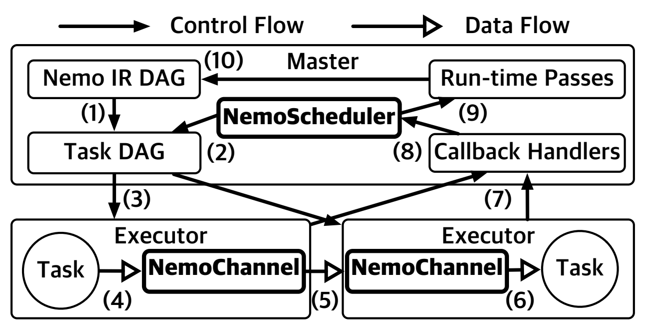


Figure 5: Nemo runtime extensions (bold) apply optimizations in a distributed runtime.

because the `LargeShufflePass` 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 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.

3.3 Runtime Extensions

We use a Nemo-compatible runtime depicted in Figure 5 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 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 `DataStores`, 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 3.1.2 (10), and resume scheduling.

4 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 [15] and LLVM [26]: frontend, optimizer, and backend.

The frontend translates applications such as Beam and RDD applications into an IR DAG (Section 3.1). At present, our frontend provides translation support for all Beam [1] operators, and a subset of `RDD` [50] 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). The backend configures the underlying runtime with the optimizer and the runtime extensions (Section 3.3).

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 `PCollections` of the `GroupByKey` `PTransforms`.

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 [46], and consists of master and executor processes similar to the Nemo-compatible runtime described in Section 3.3. 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 [46], our runtime runs on resource managers such as Hadoop YARN [2] and Mesos [18].

5 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 [2] 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.

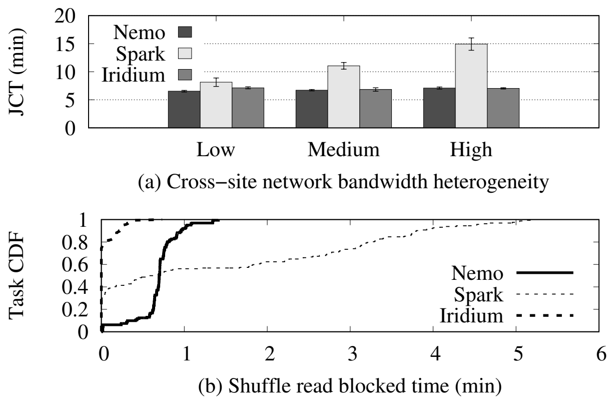


Figure 6: JCT for different cross-site network bandwidths, and CDF of shuffle read blocked time of tasks under the high cross-site network bandwidth heterogeneity.

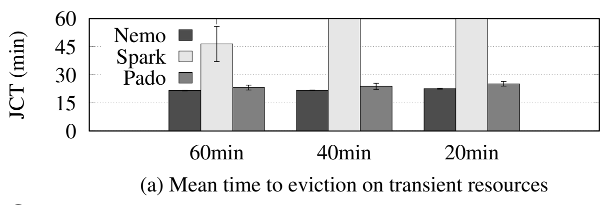
5.1 Fine Control

In this experiment, we evaluate how Nemo applies fine control under different resource and data characteristics. For comparison we run Spark 2.3.0 [4], 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 [33] for geo-distributed resources, Pado [48] for transient resources, and Hurricane [12] 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.

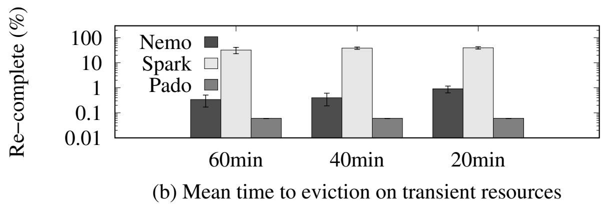
We confirm that the baseline performance is comparable for Beam and basic RDD applications on Nemo. We also confirm that the baseline performance is comparable for Spark and Nemo with the `DefaultPass`, which configures pull-based on-disk data shuffle with locality-aware computation placement similar 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 [6] to control the network speed between instances, as described in Iridium [33]. 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 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 [8] network trace dataset and computes network packet flow statistics.

The job completion time (JCT) of Iridium, Spark, and



(a) Mean time to eviction on transient resources



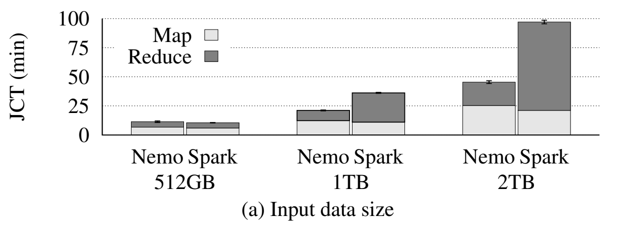
(b) Mean time to eviction on transient resources

Figure 7: JCT and ratio of re-completed tasks to original tasks for different mean times to eviction on transient resources.

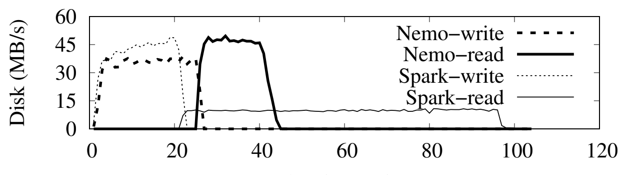
Nemo optimized with the `GeoDistPass`, are shown on Figure 6 (a). Spark degrades significantly with larger bandwidth heterogeneity, since tasks that fetch data through slow network links become stragglers. In contrast, Iridium and Nemo are stable across different network speeds. Figure 6 (b) shows that 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 [42,47,48], 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 [48]. To evaluate handling long and complex DAGs with transient resources, we run an Alternating Least Squares [23] (ALS) workload, an iterative machine learning recommendation algorithm, on 10GB Yahoo! Music user ratings data [10] 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 [47].

Figure 7 (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 7 (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.



(a) Input data size



(b) Time (min)

Figure 8: 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 [7] 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 [51] and Sailfish [35], and use 20 EC2 instances to run the workload.

The JCT of Spark and Nemo optimized with the `LargeShufflePass` are shown on Figure 8 (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 8 (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

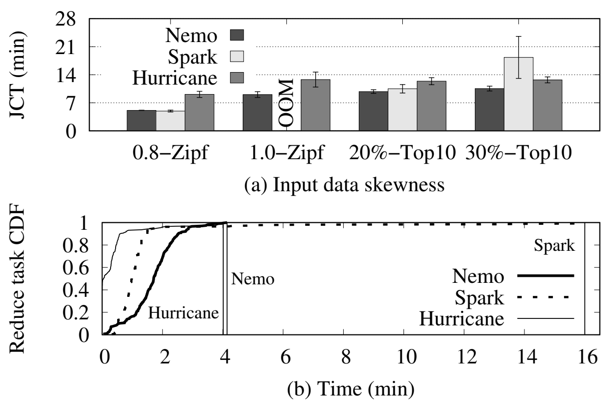


Figure 9: JCT for different input data skewness, and CDF of reduce task completion time when processing the 30%-Top10 skewed data. Each vertical line in the CDF graph denotes the completion time of the slowest reduce task.

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

tasks, which minimizes the disk seek overhead.

To measure the overhead of the `Relay` vertex inserted by the `LargeShufflePass` 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 [12]. 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 [39] to fully leverage its task cloning optimizations [12]. 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 9 (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 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 9 (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.

5.2 Composability

We now evaluate combinations of different optimization passes. Table 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 5.1, 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 5.1, 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 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 5.1. 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 [38, 48].

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 5.1 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 5.1 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 5.1. 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.

5.3 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 [1] TPC-H [9] 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.

6 Related Work

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 [22] and runtime-level [4, 20, 22, 40] 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 [15]. 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 [11] 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 [28, 29] 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 [32] takes as input code that composes imperative libraries such as Pandas [30] and Numpy [41], 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 [14, 16]. 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 [13]. 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 [26]. 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 [27], 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.

7 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.

8 Conclusion

We 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.

Acknowledgments

We thank our shepherd Ashvin Goel and the anonymous reviewers for their feedback. We thank Sunghwan Ihm, Brian Cho, Rodrigo Fonseca, and members of the Software Platform Lab at Seoul National University for their comments on the draft. We also thank Min Hyeok Kweun, Jaehyeon Park, Seonghyun Park, Woo-Yeon Lee, Taegeon Um, Gyewon Lee, Yunseong Lee, Eunji Jeong, and Soojeong Kim for helping to implement and evaluate Nemo. Finally, we are grateful to the Apache Nemo community for their contributions. This work was supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIT) (No.2015-0-00221, Development of a Unified High-Performance Stack for Diverse Big Data Analytics).

References

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

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

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

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

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

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

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

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

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

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

[11] 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.

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

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

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

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

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

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

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

[26] 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.

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

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

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

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

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

[32] 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.

[33] 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.

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

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

[36] Alexander Rasmussen, Vinh The Lam, Michael Conley, George Porter, Rishi Kapoor, and Amin Vahdat. Themis: An i/o-efficient mapreduce. In *SOCC*, 2012.

[37] 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.

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

[39] 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.

[40] 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.

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

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

[43] 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.

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

[45] 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.

[46] 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.

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

[48] 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 *EuroSys*, 2017.

[49] 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.

[50] 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.

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

[52] 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.