Object Management in a Distributed Futures System

Edward Oakes

May 29, 2020

Object Management in a Distributed Futures System

by Edward Oakes

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

[illegible]

Professor Scott Shenker
Research Advisor

5/27/2020
(Date)

* * * * * * *

[illegible]

Professor Ion Stoica
Second Reader

5/27/2020
(Date)

Abstract

Object Management in a Distributed Futures System

by

Edward Oakes

Master of Science in Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Scott Shenker, Chair

In recent years, there has been an increasing demand for distributed data processing across a wide range of application domains. In addition to the previous generation of large scale data processing, there are also new emerging applications that are centered around novel artificial intelligence and machine learning techniques. These applications require a much more flexible programming interface than the traditional static execution graph offered by bulk synchronous parallel systems. In response, a number of domain-specific distributed systems have been built to address the needs of each new type of application. Ray has the promise to act as a unified execution engine for these applications, enabling high-performance distributed execution with a simple but flexible futures-based programming model. However, the system falls short of these promises due to two key shortcomings: application-agnostic least recently used (LRU) eviction for shared memory objects and high overhead for small objects. This work proposes a novel object management architecture for distributed futures that enables exact reference counting for shared-memory objects and reduces the overhead for tasks that depend on or produce small objects to that of nearly a single remote procedure call.

Contents

Acknowledgments

This work would not have been possible without close collaboration with Eric Liang and Stephanie Wang (particularly Chapter 4, much of which was joint work).

A special thanks to the rest of the Ray team and open source collaborators for making working on the project an educational, fun, and meaningful experience (including, but not limited to): Hao Chen, Eric Liang, Richard Liaw, Simon Mo, Philipp Moritz, Mehrdad Niknami, and Robert Nishihara.

My time in the NetSys Lab was filled with intriguing discussion, genuine camaraderie, and great lunchtime conversation. Thank you to everyone in the lab who helped make it a truly special environment: Emmanuel Amaro, Chris Branner-Augmon, Lukas Burkhalter, Michael Chang, Vivian Fang, Silvery Fu, Yotam Harchol, Anwar Hithnawi, Murphy McCauley, and Wen Zhang.

Chapter 1

Introduction

The past two decades have seen a proliferation in the prevalence and capabilities of large-scale data processing. Steered by the demand of increasingly data-driven applications, a number of systems have been developed to perform computations over massive data sets using large-scale parallelism [19, 23, 43, 45]. These systems simply application development by providing a high-level API and handling parallelism, resource scheduling, and fault tolerance under the hood.

These systems offer a bulk synchronous parallel (BSP) model [41]. While this programming model drastically simplifies data analytics, recently it has proven to be limiting for a growing class of parallel applications, including reinforcement learning [33], video processing [38], and distributed machine learning training [40]. These applications require a more complex execution pattern than is supported by the BSP model, so application developers are forced to build a new system from the ground up [38], rely on lower-level primitives for distributed execution [40], or modify existing BSP systems [12].

In addition, in recent years there has been an explosion in interest in data-intensive artificial intelligence (AI) and machine learning (ML) applications [25]. These techniques are applied in wide-ranging domains from neuroscience [32], to computer networking [30], to speech recognition [10]. While these applications often require massive data sets for training and validation, they are not well served by traditional BSP systems [18]. In order to support the complex and rapidly-evolving demands of these applications, a range of machine learning-specific systems have been developed [3, 7, 37, 40].

These developments point to a unified trend: emerging applications are increasingly reliant on large-scale distributed processing, but their complex execution patterns cause the existing BSP model to fall short. Building a specialized, domain-specific system for each new emerging class of applications limits the rate of innovation and forces researchers and application developers to focus on infrastructure when they could instead be focus their efforts on solving the problem at hand.

Ray [33] offers a potential solution to this problem, offering a simple but flexible programming interface that supports dynamic task execution from any point in an application. The two key building blocks to Ray programs are *tasks*, or stateless remote functions, and *actors*, stateful remote classes. Tasks can be invoked and actors can be created and their methods can be called *dynamically* from any part of the application. When a Ray program invokes a remote task or actor method call, the system returns a reference to the result of the object, or *future*, immediately. The application can later fetch the result of the call or pass the reference to other tasks or actor method calls to encode data dependencies. A more detailed introduction to the Ray programming model and architecture is provided in Chapter 3.

This simple but flexible interface supports a wide range of distributed applications. In addition, the system provides two key benefits that enable it to support high-performance applications with this simple programming model:

- **Shared memory**. Many applications have shared data dependencies across units of computation (in Ray, task and actor method calls). Making redundant copies of this data is expensive and increases resource consumption. Ray transparently manages a shared-memory object store that allows multiple processes on the same machine to read a single immutable copy of each object.

- **Fine-grained tasks**. Task and actor method calls begin execution on the order of single-digit milliseconds. This allows Ray to support applications that are not possible on bulk synchronous parallel systems that have an order of magnitude more overhead for each invocation [33].

While these benefits are already evidenced by a number of applications written on top of Ray [29, 31, 33], they fall short in two key ways. First, while the shared memory subsystem enables efficient distribution and sharing and of objects, it is completely decoupled from the rest of the system and as a result relies on least recently used (LRU) eviction to delete objects. For many applications, this behavior leads to adverse effects as the system may evict objects that are still in use by the application. When this happens, in the best case the system is able to transparently reconstruct the evicted object and the application succeeds but experiences unnecessary performance degradation. Even worse, the object may not be reconstructable because it was generated by an external source or its generating tasks cannot safely be replayed. In this case, the application crashes despite its working set not exceeding available resources. Note that in some cases, even if the object is reconstructable an eviction may lead to the application failing as the reconstructing an object can lead to new objects being created, which may in turn lead to new evictions, causing a loop of infinite reconstruction. This eviction behavior clearly limits the set of viable application patterns that can be supported by the system.

Second, while the shared memory object store drastically improves performance for applications with large objects, it also adds a static overhead to each object creation and access. For applications that generate small objects (e.g., control plane messages or summary values), this can drastically increase overhead compared to a system written using bare remote procedure calls. Given that Ray is meant to be a *general purpose* distributed execution framework, this overhead is unacceptable.

This work proposes an object management architecture that addresses both of these problems, drastically improving performance for small objects by inlining arguments and return values (Chapter 4) and enabling exact reference counting for shared memory objects (Chapter 5). The benefits of the architecture are explored in Chapter 6.

Chapter 2

Related Work

2.1 Distributed Dataflow Systems

Distributed data-parallel systems are optimized for processing large amounts of data in parallel following a simple computational pattern. These systems [19, 23, 43, 46] require the user to specify a complete dataflow graph, which the system then decomposes into stages that execute across a set of worker nodes. The computation is orchestrated using some form of a centralized coordinator process that is responsible for scheduling execution, transferring intermediate results across worker nodes, and transparent recovery from failure.

Similar to the work presented here, many of these systems use distributed memory to minimize data movement and copies [23, 45]. However, these systems primarily focus on coarse-grained computation, so they do not have the same requirements for minimizing single-task latency Additionally, both the user specifying the computation graph upfront and the system having only one level of parallelism makes tracking the lifetime of objects simpler.

2.2 Actor Systems

The actor model is a distributed programming paradigm for stateful computation [1, 13, 17, 22]. The model defines a set of general rules for how system components should interact in order to simplify reasoning about the concurrency and fault tolerance properties of the system.

Traditionally in the actor model, all data is passed directly in messages rather than referencing an external data source or shared memory. This is prohibitively expensive for data-processing workloads that commonly have tasks that operate on shared data dependencies. Ray extends this model by supporting actors and introducing a shared memory object store that they can use to send data dependencies by reference rather than only by value. Note that while one of the key tenets of the actor system is that there should not be shared state between actors, the Ray object store does not violate this principle because all objects in the store are immutable.

The work presented here can be viewed as an extension of the actor model. The system supports both stateless tasks and stateful actors, inlined objects as described in Chapter 4 are similar to messages passed in an actor system, and the use of shared memory extends the actor model to improve efficiency for large data dependencies. Additionally, the failure properties of reference counting for shared objects is similar in nature to the supervision model [13].

2.3 Parallel Programming Systems

MPI [21] is a parallel programming framework that exposes a low-level message passing interface. MPI does not support distributed memory, instead opting for a set of synchronization primitives like broadcast and all-reduce. Given that the lifetimes of object references are less flexible than in Ray and only exist as they are passed as message between processes, tracking them is straightforward.

Other systems support dynamic dataflow graphs, similar to Ray [3, 34, 39]. Many of these systems have a similar notion of object references, but rely on a centralized scheduler or master. The master handles all resource requirements, task placements, and object reference counting in the system. This simplifies the challenge of managing the lifetimes of objects due to the centralized logic, but offers limited scalability.

2.4 Distributed Memory

There is a large body of existing work on how to enable the appearance of a single shared address space that references memory across a cluster of machines, called distributed shared memory (DSM) [35]. The original goal of this work was to enable applications written for a single machine, where there is a single virtual address space, to seamlessly be scaled to a cluster setting where memory is distributed across many machines. The programming interface is much lower-level than the work presented here, and the attempt at maintaining this low-level abstraction has led to significant challenges in performance, consistency, and fault tolerance [16, 26, 28, 35]. In contrast, the abstraction provided by Ray is much higher level, only exposing the notion of static-sized, immutable objects, and is tightly coupled with the programming model [33], easing the requirements of the distributed memory subsystem.

The interface provided by the shared memory object store in Ray is similar in concept to that of many recently developed key-value stores [20, 27] and other distributed memory architectures for parallel computing [11, 15, 36]. These systems have a broad range of performance and semantics requirements, and while they don't explicitly support shared memory for processes on the same machine, it would likely be a straightforward extension. Many of these could likely be used in place of the Ray distributed memory subsystem, but they have more complex requirements due to supporting a wider range of applications. Additionally, these systems are not tightly coupled to the programming model and therefore would still require a similar reference counting solution to the one presented in this work.

2.5 Distributed Reference Counting and Garbage Collection

There is a large body of research on general-purpose distributed reference counting and garbage collection [8]. This work was largely an extension of existing garbage collection protocols developed for programming languages like Java [9]. The primary challenge in this body of work is supporting a very general programming paradigm: objects can be distributed throughout the system arbitrarily and any object in the system may hold a reference to any other object. This flexibility requires addressing classically difficult problems such detecting reference cycles [24] and performing liveness checks [44]. In Ray, object lifetimes are tightly coupled with the programming model, so the lifetime of an object can be determined on a single process without applying these techniques.

Chapter 3

Ray Overview

This chapter provides a brief overview of Ray [33] as required to understand Chapter 4 and Chapter 5. This chapter describes the existing programming interface and system architecture of Ray as of version 0.7.7 [5].

Programming Model

Ray supports stateless distributed computation via *tasks*. A simple example program using tasks is shown in Figure 3.1. Tasks are defined as simple functions and are executed in worker processes across the cluster. When the application submits a task, the system immediately returns a *reference* to the eventual result of the task and asynchronously submits the task for execution. The application can then use the reference to fetch the result of the task or pass it to another task as a data dependency.

In addition to stateless tasks, Ray also supports stateful computation via *actors*. A simple example program using an actor is shown in Figure 3.2. A program defines an actor as a class that can be initialized and supports a number of *remote methods*. When an actor is instantiated by the application, the actor class is initialized within a worker process on the cluster and consumes its resources for the actor's lifetime. The application can then submit remote method calls on the actor, which have the same behavior as tasks except that they are only executed in the target actor process and can read and mutate the state of the actor class.

If an object containing the result of a task is lost due to a node failure or eviction, the system attempts to transparently reconstruct the object by

```python
object_id_1 = f.remote()
object_id_2 = f.remote()
ray.get(add.remote(
    object_id_1, object_id_2))
```

Figure 3.1: Python code illustrating the API for Ray tasks. `f.remote` is called twice, asynchronously executing the tasks and immediately returning futures (called `Object IDs`) that reference their outputs. These futures are passed into another remote task, `add.remote`. The result of `add.remote` is retrieved synchronously using `ray.get`, which will return once all three tasks have finished executing.

```python
@ray.remote
class Counter:
    def __init__(self):
        self.counter = 0

    def increment(self, count):
        self.counter += count
        return self.counter

counter = Counter.remote()
intermediate = generate_count.remote()
ray.get(counter.increment(intermediate))
```

Figure 3.2: Python code illustring the API for Ray actors. The actor is defined as an ordinary class, `Counter`, and decorated with `@ray.remote`. This class is instantiated as a remote actor and then remote method calls can be made on it. These method calls behave like remote tasks, executing asynchronously and returning futures referencing their outputs.

resubmitting its lineage, or the chain of tasks that created it, using a lineage stash protocol [42]. This may not be possible in all cases, for instance if the task depends on an actor method call (which cannot be reconstructed due to internal actor state) or an object created directly from an external data source.

System Architecture

A high-level overview of the Ray system architecture is shown in Figure 3.3. A description of each system component and its role in distributed execution is provided below.

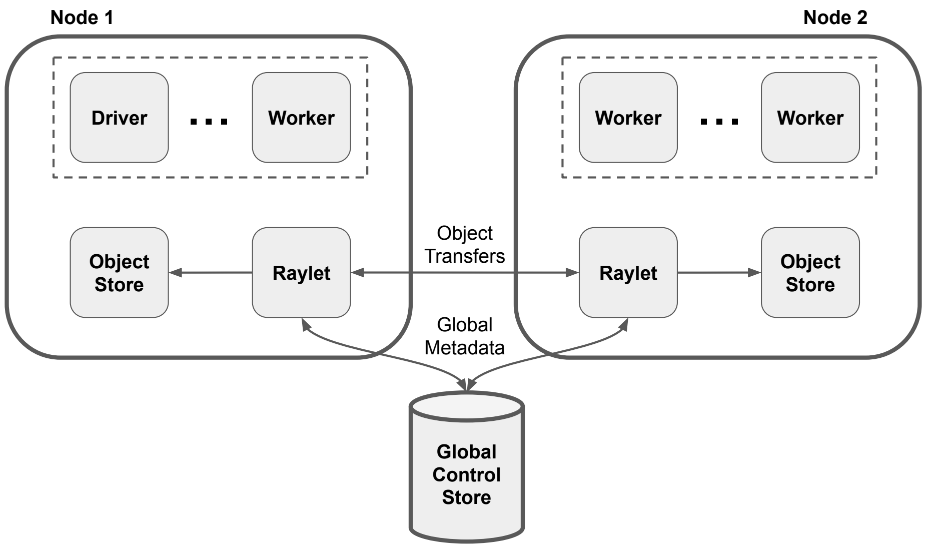


Figure 3.3: Overview of the Ray system architecture. Nodes in the system are uniform and each consists of a raylet that manages the node resources, a shared memory object store, and a set of worker processes. The global control store holds global metadata about each node and the locations of objects.

Driver Process Each application consists of a single driver process that executes the top-level user-input program. The driver process submits tasks and creates actors across the cluster by submitting them to its local raylet. The driver process can be located on any node in the cluster and is equivalent to a worker process except that it cannot execute tasks or actors.

Worker Processes A number of worker processes are started on each node in the cluster. These worker processes execute remote tasks and actors that are submitted by the driver process or other worker processes. While executing tasks or actor method calls, worker processes may create new actors, submit tasks, or create or read objects from the shared memory object store. When an actor is instantiated, it is created within a worker process and consumes the worker process for its lifetime.

Raylets Each node has a single raylet process that manages its resources. When a driver or worker process on the node wants to submit a task, it sends it to its local raylet which is responsible for making a scheduling decision. Object retrieval also goes through the raylet: when a process wants to retrieve an object, it first requests the raylet to fetch the object if it is not local. Then, once the object has been written to the local object store, the process reads it directly from shared memory. The raylet sends periodic heartbeats with information about the node to the global control store (GCS).

Shared Memory Object Store Objects passed as input to tasks and returned as results of tasks are placed in the shared memory object store, where a single copy can be read by any process on the same machine. These objects are immutable once they have been created and transfers of objects across machines are managed by the raylets. When a new object is created but there is not sufficient space for it in the object store, the object store evicts the objects that were least recently accessed to make space.

Global Control Store The Global Control Store (GCS) holds global metadata about each node, including a potentially out-of-date view of its resource availability, the actors placed on it, and the objects it contains in its shared memory object store. Raylet processes on each node exchange resource information by writing and reading periodic heartbeats to and from the GCS. When a task is submitted, the raylet uses this information to make a scheduling decision of where to execute the task (described in more detail in Chapter 4.

Chapter 4

Selective Object Inlining

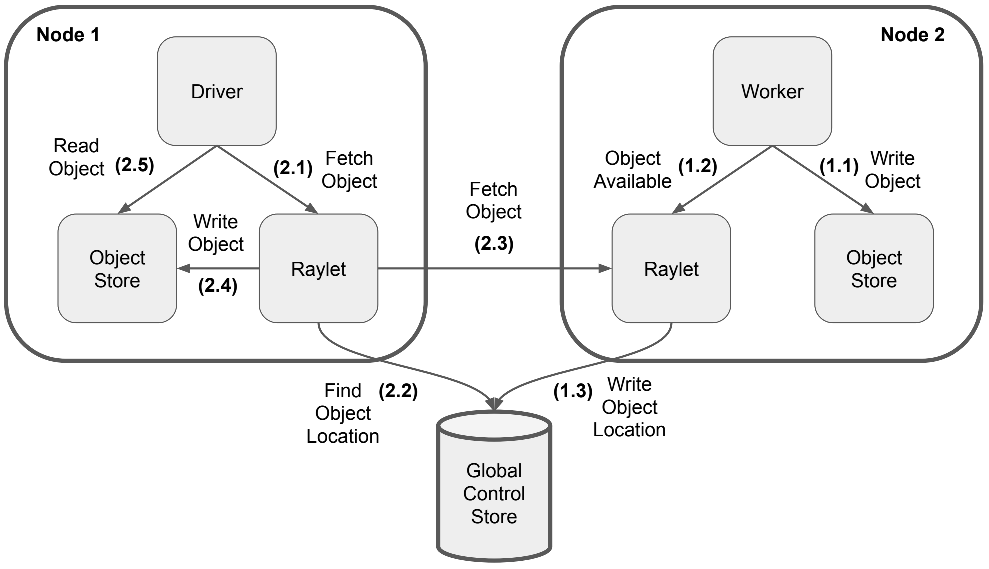


Figure 4.1: The protocol for (1) writing and (2) retrieving shared memory objects. The locations of all shared memory objects are stored in the *Global Control Store* (GCS). When a process creates a shared memory object, it writes the object to its local object store (1.1) and notifies its local raylet that the object has been created (1.2). The raylet then writes the object location to the GCS (1.3). When another process tries to retrieve this object, it requests its local raylet to fetch the object (2.1). If the object is not local, the raylet then looks up the object's locations in the GCS (2.2), fetches the object from a raylet that has the object (2.3), and writes the object to the local object store (2.4). The local process then directly reads the object from the local object store (2.5).

Shared memory provides compelling benefits for applications with tasks that have shared data dependencies, particularly when the data is large.

However, managing this transparent shared memory system also introduces substantial static overhead per object for tasks that create and read objects.

Figure 4.1 describes the protocols for a creating and retrieving shared memory objects. In order to create an object (e.g., the return value of an executed task), a worker must write the object to the object store and indicate that the object is available to its local raylet, which then writes the object location to the GCS for future lookups. On the retrieving side, a process requests its local raylet to fetch the object. The local raylet looks up the object's locations in the GCS, then retrieves the object and writes it to the local object store. Finally, the raylet returns to the retrieving process which then directly reads the object from the local object store.

This protocol is very flexible, allowing an object to be dynamically fetched from any process in the system with a reference to it. However, it also introduces substantial overhead: communicating between multiple local and remote processes and writing to and reading from remote memory.

For large objects, the cost of making copies and transferring the data is high. This cost dominates the static per-object cost of the overheads described above, and enabling multiple processes to read the same object via shared memory has huge benefits. Therefore, the trade-off is clear for large objects: while this shared memory system adds a small amount of overhead, the benefits of shared memory far outweigh this cost.

However, applications with small data dependencies must also pay this performance penalty without reaping the benefits. Consider the common case where a driver submits a task that returns a small value such as a status. In this case, the cost to copy and transfer the data is negligible and the flexibility of sharing references to the object is unnecessary. Instead of using this heavyweight protocol, this overhead could be avoided by directly send the result from the executing process to the retrieving process without making the object globally accessible.

This chapter describes a hybrid approach that transparently enables the benefits of shared memory for large objects while instead opting for a low-overhead, direct remote procedure call (RPC) path for small objects. The design goals for this hybrid approach are as follows:

- Minimal overhead for small objects. The overhead for executing and fetching the result of a task that has both small data inputs and outputs should be as close as possible to the baseline of a single RPC call.

- Transparency from the perspective of the application. The system should automatically select between the execution paths for small and large objects to maximize performance without requiring input from the application. Furthermore, a single application should be able to benefit from both execution paths across different tasks.

Section 4.1 describes the direct RPC execution path implemented to reduce overhead for small objects and Section 4.2 describes how the system dynamically selects between the direct RPC path and the existing shared memory path for each object created in the system.

4.1 Direct RPC Task Submission

Ray supports both stateful *actors* and stateless *tasks*. Actors consume resources for their entire lifetime, but individual remote method calls on them do not require any form of scheduling. However, for tasks, a scheduling decision must be made for each invocation. The existing scheduling procedure for tasks is described in Figure 4.2. Raylet processes manage the resource usage and scheduling of each node and communicate this information globally via heartbeats to the *Global Control Store* (GCS). When a process executes a task, it submits the task to its local raylet, which either schedules it on a local worker if there are resources available or forwards it to a remote raylet. This forwarding is called *spillback scheduling*.

A naïve approach to minimize overhead for small objects would be to inline them directly into the message to execute or finish executing a task. However, the current scheduling architecture requires that these messages pass through both the local and remote raylets. Adding data transfer to these messages, even if objects are small, could substantially increase the load on both the submitting and executing raylets due to additional memory copies. Instead, if the messages were sent directly between the submitting and executing workers, there would not be any additional overhead introduced. This section describes modifications made to task scheduling to support such

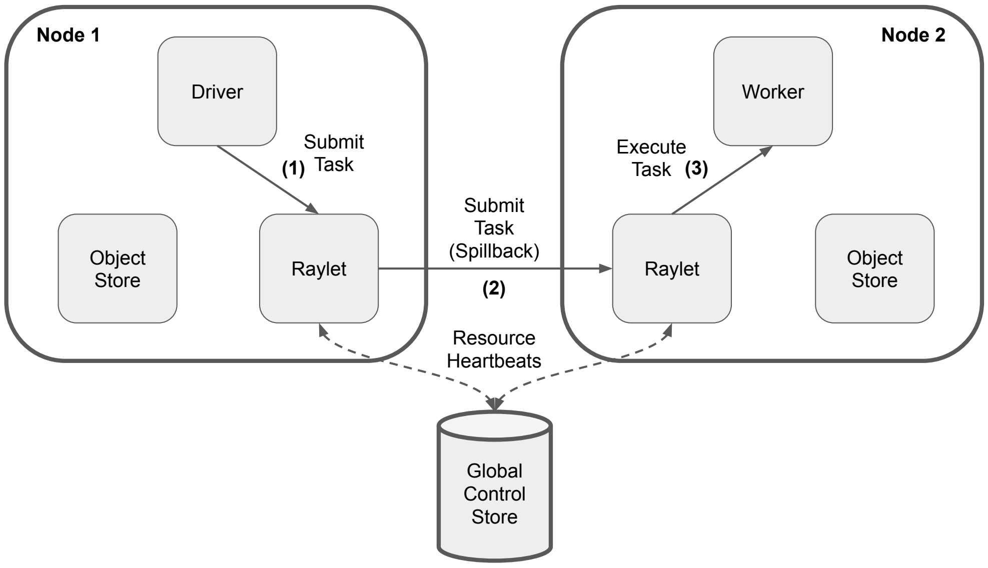


Figure 4.2: The raylet-based task submission protocol. The raylet on each node is responsible for scheduling local resources. When a driver process wants to execute a task, it first submits the task to its local raylet. If there are sufficient resources to execute the task on the local node, the raylet will execute it on a local worker process. Otherwise, the raylet will perform spillback scheduling by forwarding the submitted task to a remote raylet with sufficient resources. Then, the remote raylet will execute the task on a worker process on its local node.

a protocol where task execution and returns are made via direct RPCs between the calling and executing worker processes.

Converting actor method calls to direct RPCs is straightforward: a reference to an actor simply contains that actor's RPC address (i.e., the IP address and port that it is listening on). Then, processes submitting method calls on the actor send the task specification directly to the actor. The actor queues and executes calls locally, responding directly to the caller when each call is finished.

However, submitting tasks via direct RPC is less straightforward. The protocol must still respect resource scheduling, but clearly if each process submitting tasks independently chooses a worker process to execute tasks on, these scheduling decisions may conflict. Instead, there must be a consistent view of system resources for each node.

To achieve this, the protocol is modified to continue to use the raylet for the *control plane*, making scheduling decisions, but removing it from the *data plane*, actually submitting tasks and returning their results. The modified scheduling protocol is shown in Figure 4.3. Instead of the submitting worker sending the full task specification to the raylet, which then makes a scheduling decision and forwards the task along accordingly, the submitting worker instead sends only the metadata about a task and the raylet responds with a scheduling decision. If the raylet has sufficient resources to schedule the task locally, it returns a *worker lease* for a worker on its local node. While the submitting worker holds this lease, it will not be handed out to any other requesting workers and it can submit tasks directly to the worker, which will also respond directly when the task completes. If the raylet does not have sufficient resources to schedule the task locally, it instead returns a *spillback redirect* to a remote raylet. The submitting worker then sends the same initial request to the remote raylet returned in the redirect. This process repeats until a worker lease is granted.

Note that this protocol adds additional overhead for each scheduling decision because the worker must wait for and process a response from each raylet before actually submitting the task. This cost is amortized by caching worker leases across multiple requests with the same resource requirements (meaning that the scheduling decision made for the initial request that granted the worker lease is still valid). The submitting worker returns the worker lease to the raylet when it runs out of tasks with the given resource shape or a timeout is reached (500 ms by default), whichever comes first.

4.2 Selective Inlining

As described above, the overhead of going through the shared memory object store for small arguments to and return objects from remote tasks and actor calls is significant. However, given that with the modified scheduling protocol described in Section 4.1, tasks are submitted directly between worker processes via RPC, small objects can trivially be inlined into both the submission and return RPCs.

Instead of only holding references to the shared memory object store, each worker process also maintains an *in-process object store* that keeps small

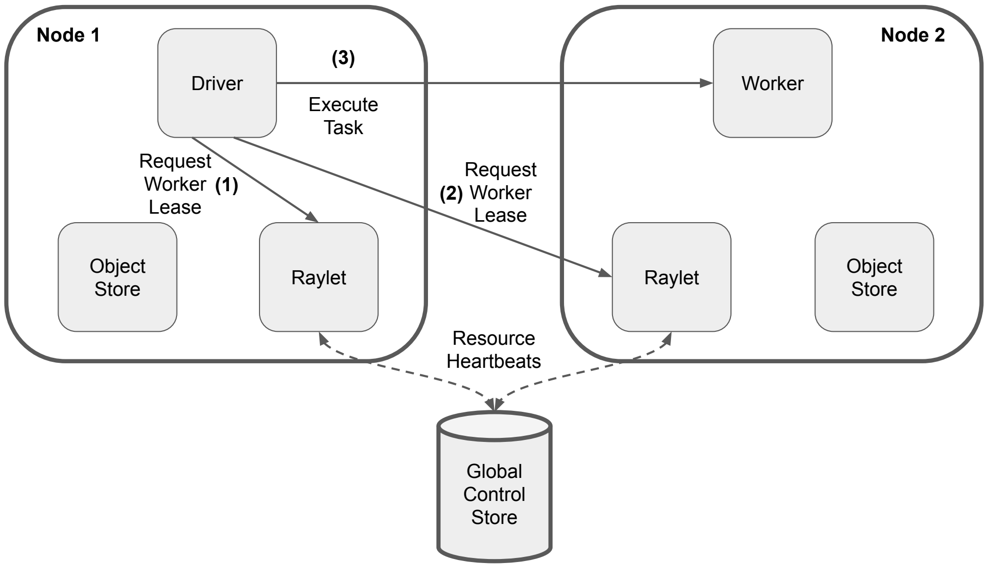
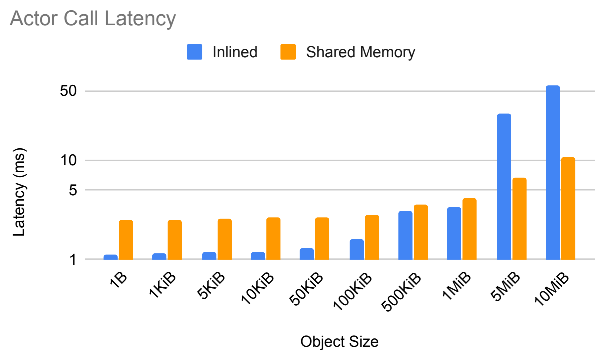


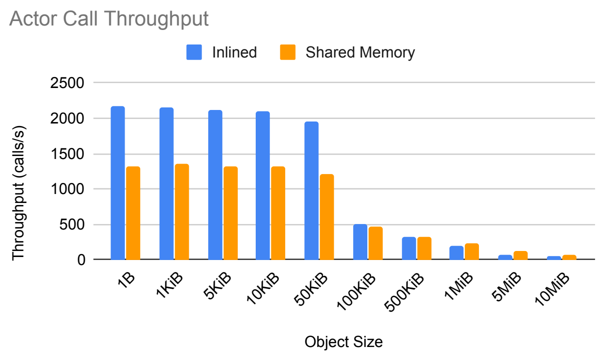
Figure 4.3: The direct RPC task ssubmission protocol. When a driver process wants to execute a task, it sends the task scheduling information of the task (e.g., resource requirements and dependencies) to its local raylet. If there are sufficient resources on the local node, the raylet will select a worker process to execute the task and return a lease to that worker to the driver. The driver will then directly send an RPC to execute the task to that worker. If the raylet does not have sufficient resources on its local node to execute the task, it instead returns a redirect to the driver to a remote raylet. The driver then performs the same leasing request to the remote raylet.

objects on the heap. When a remote task returns a small object, instead of putting it in the object store and returning a reference to it, it instead returns the object inline in the return RPC. The calling process then stores the object in the in-process object store, which is transparently accessed by the application using the same interface as the shared memory object store. If the application submits another task with an object in the in-process store as a dependency, the object will similarly be inlined in the submission RPC.

While this behavior improves performance for small objects, for large objects it is undesirable as the application can no longer benefit from shared memory and object transfer speed may be lower as the worker processes are not optimized for transferring large objects. Figure 4.4 shows two experiments used to empirically determine an object size threshold for when to inline objects and when to put them in the shared-memory object store.



(a) Latency of actor calls with a single input and output of a varying size, measured as the time it takes to call and synchronously retrieve the result of a single call.



(b) Throughput of actor calls with a single input and output of a varying size, measured by spawning a chain of 1000 calls, each of which depends on the result of the previous call.

Figure 4.4: Comparing the latency and throughput of actor calls for varying object sizes used for both input and output objects. Both throughput and latency are improved by inlining for small objects: latency is better up to ~500 KiB objects and throughput is better up to ~100 KiB objects.

Each of these experiments consists of two actors on separate AWS m4.xlarge instances. All remote method calls take as input a single object of varying size and return as output of the same size. The first experiment measures latency by synchronously calling a task and fetching its result. The second experiment measures throughput by asynchronously submitting a chain of 1000 tasks where each task depends on the previous task's output. In both experiments, the inlined objects case outperforms shared memory for small objects and vice versa for large objects. Latency is better for inlined objects of up to 500 KiB objects and throughput is better up to 100 KiB. Therefore, a default threshold of 100 KiB for inlining objects is chosen.

Chapter 5

Object Lifetime Management

The programming model of Ray is very flexible: objects can be created, tasks can be invoked, and actors can be instantiated dynamically from any part of the program. This flexibility offers great benefits in terms of expressibility to applications, enabling novel applications such as distributed reinforcement learning that could not be implemented with more limited programming models [29]. However, this flexibility also introduces challenges for the system to track the complex life cycle of resources.

One key example of this is tracking the life cycle of objects in the shared memory object store. These objects can be dynamically created by a call to `ray.put()` or invoking a remote task or actor method call. References to the objects can then be passed as dependencies into potentially many future remote tasks or actor method calls. This mechanism is very flexible: references to the objects can be held onto by the application for an indefinite period before being retrieved or passed as dependencies and a single object can be passed as a dependency for possibly many remote calls.

In Ray, these objects are simply evicted from the object store on each node independently when that object store becomes full. Objects are chosen for eviction using a Least Recently Used (LRU) policy. While this policy is acceptable for many applications, there are classes of applications for which this behavior can cause degraded performance or the application to simply fail. A trivial example of such an application is shown in 5.1. In this application, an initial object is created before then processing a number of intermediate results before finally fetching the result of the initial object. If enough intermediate objects are created to fill the local object store before retrieving the

```python
initial_object = expensive_task.remote()
intermediates = []
for i in range(NUM_INPUTS):
    intermediates.append(ray.get(compute.remote()))
ray.get(initial_object)
```

Figure 5.1: Python code for a simple Ray application that may fail under LRU eviction. If enough intermediate objects are created to fill the local object store before retrieving the initial object, the initial object would have been evicted. The object may or may not be able to be transparently reconstructed.

initial object, the initial object would have been evicted. In the ideal case, this initial object would be able to be reconstructed transparently (e.g., if the creating task is pure), so the application would perform redundant work but succeed. However, if the initial object cannot be reconstructed, the application would fail despite being feasible given the resources on the node. This behavior is *inefficient* because it causes unnecessary reconstruction, *unpredictable* because the application does not know which objects will be evicted and therefore which objects may be reconstructed, and *limiting* because it prevents some applications from successfully running even when they have enough resources to be feasible.

This chapter describes an *exact reference counting* protocol in Section 5.1 and an *object pinning* mechanism in Section 5.2 that together address this issue. These extensions ensure that any object that has a reference in the application has at least one copy *pinned*, meaning that the object cannot be evicted. Having at least one pinned copy of an in-scope object prevents spurious reconstruction or application failure due to premature eviction.

5.1 Reference Counting

There are two types of references that an application can have to an object in Ray: application references and submitted task references.

Application references are returned by API calls that directly create an object or spawn a task or actor method call that returns an object. These references are used by the application to wait for the object to be created

```
object_id = f.remote()
ray.get(g.remote(object_id))
# Object still in scope due
# to application reference.
del object_id
# Object no longer in scope.
```

(a)

```
object_id = f.remote()
g.remote(object_id)
del object_id
# Object still in scope due
# to submitted task reference.
ray.get(g.remote())
# Object no longer in scope.
```

(b)

Figure 5.2: Two applications illustrating application and submitted task references. In both cases, an object is initially created as the return value of a call to `f.remote` and then passed into a second call to `g.remote`. In the first case (a), the application waits for both tasks to finish, after which there are no submitted references but a local reference exists. The application then removes the local reference, causing the object to go out of scope. In the second case (b), the application deletes its local reference, after which there are no local references but a submitted reference exists. The application then waits for the tasks to finish, after which the submitted reference is removed and the object goes out of scope.

(e.g., the creating task to finish), pass a reference to the object to another remote call, or retrieve the value of an object.

Submitted task references are created when the application passes an object to a remote task or actor method call. The underlying object is then presented as an argument to the implementation of the remote task or actor method. Importantly, a submitted task reference can be created before an object is actually available because the API is inherently asynchronous.

In order for an object to be considered *out of scope*, it must have a reference count of zero for both application references and submitted task references. When this happens, the application can no longer use the underlying value of the object, so it is safe to evict the object. An application can have either application references but no submitted task references or submitted task references but no application references. Figure 5.2 illustrates each of these behaviors.

Note that while the primary motivation for this reference counting protocol is shared memory objects, the same protocol is also applied to evicting objects that are returned in-line (as described in Section 4.2) from the in-process object store.

Application References

Application reference counting is largely delegated to the application language frontend. The system keeps a simple counter of the number of application references currently held by the application. This counter is initialized to `1` when an object is first created. The application frontend is then required to increment the counter each time the object reference is copied and to decrement the counter when a copy of the object reference is destroyed.

Submitted Task References

As described in Section 4.1, tasks and actor method calls are made as direct RPCs between worker processes. This makes tracking submitted task references simple: a worker increments the submitted task reference count for each dependency of a task when the submission RPC is sent to the executing worker and decrements the reference count for each of the dependencies when the return RPC is received.

Limitations

In a garbage-collected language such as Java [14], it is possible that the application reference count does not decrease until long after the application has actually ceased to use a reference. This is also a problem for some reference-counted languages such as Python that rely on garbage collection to break reference cycles [2]. This does not pose a significant problem until the object store is full, in which case an application will fail to create a new object (as described in section 5.2). The problem is addressed by broadcasting a global message to all processes in the system that triggers the local garbage collector.

Additionally, the reference counting protocol as describe assumes that references are limited to the process that creates them. While an object can be shared as a dependency with a remote task or actor, the receiving task or actor does not have a reference, so the object cannot outlive the lifetime of the called task or actor method. The task or actor could instead return a message to the caller indicating that it should send the object to another task or actor, but they cannot pass a reference directly. However, this behavior could be supported by extending the protocol here using existing distributed counting techniques [8].

5.2 Object Pinning

Section 5.1 describes a protocol for determining *when* references may be used by the application in the future (or conversely when they may be safely removed). Given this information, there are a few properties that the system must support in order to guarantee application correctness:

- At least one copy of an object should be pinned in while a reference to it exists in the application.
- Once no references exist to an object, the space it occupies should be made available to new objects that may be created. This includes if the process holding references to an object dies (or the node the process is on is removed from the system).

In addition to these guarantees, the mechanism used to achieve them should add limited overhead to the system.

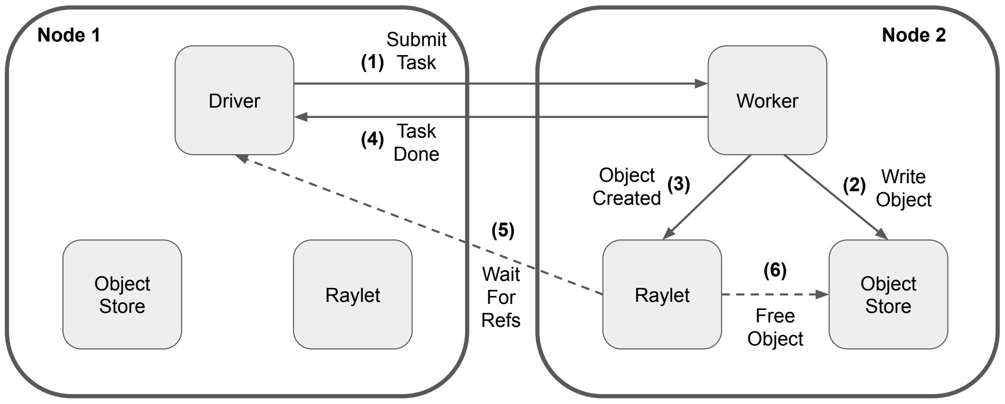


Figure 5.3: Object pinning protocol. The creating process sends a message to its local raylet with the address of the owning process. The raylet then polls the owning process until the object goes out of scope, at which point it frees the object.

Figure 5.3 shows the mechanism used to achieve these guarantees. When a process first creates an object, it sends a message to its local raylet with the address of the owning process (i.e., the process that holds references to that object), which may or may not be on the same node. Then, the raylet sends a separate RPC to the owning process, which the owning process responds to once the reference count for the object reaches zero. At that point, the raylet frees the object from the local object store. This long-polling RPC doubles as a failure detection mechanism: the raylet uses keepalive messages while the RPC is outstanding to check that the owning process is still reachable. If these keepalives fail, the raylet considers the object out of scope.

This protocol clearly meets the guarantees outlined above: while a reference to an object exists in the owning process, the raylet will ensure that it is not evicted and once those references go out of scope, the raylet is tasked with making the space available for new objects. However, there are further considerations for how the raylet should maintain that the object remains in the object store while there are references to it and subsequently how its space is made available to other objects once the references are out of scope. There are multiple possible mechanisms that could be used to meet these requirements, which are summarized in Table 5.1.

First, the raylet could *pin* objects that are in scope, preventing them from being evicted by the object store, and then *unpin* objects that go out of scope. Pinning the objects guarantees that they will not be evicted and unpinning them allows them to be lazily evicted from the object store when new objects are created. However, this could artificially inflate the resource usage of the object store because objects that are no longer in use may still be consuming memory. Additionally, applications that rely on LRU eviction (e.g., by keeping references to objects that are no longer needed) will fail because the object store will fill with pinned objects.

Instead, the raylet could leave all objects unpinned and explicitly free all copies of objects from the object store when they go out of scope. This approach addresses both the concern of inflated resource usage, as objects will be explicitly freed, and the concern of applications that rely on LRU, as this approach still falls back on LRU eviction when the working set exceeds available resources. However, the overhead to this approach may be higher because the raylet must broadcast a message to explicitly free copies

Mechanism	Inflated Memory	LRU Fallback	Overhead
Pinning + unpinning	Yes	No	Lower
Explicit free only	No	Yes	Higher
Pinning + explicit free	No	No	Higher

Table 5.1: Comparing object pinning and freeing strategies. Pinning + explicit free is chosen as the default.

of object when it goes out of scope. Additionally, this approach may cause unpredictable application behavior as applications may sometimes succeed and sometimes fail based on the resource contention and eviction pattern of the object store.

A hybrid approach could also be taken: pinning objects in scope and then explicitly freeing them when they go out of scope. This addresses the concern of inflated resource usage but not that of LRU-dependent applications, while incurring the same overhead as the second approach.

These options are made configurable, but the hybrid approach is taken by default in order to minimize resource usage and avoid unpredictable system behavior.

Chapter 6

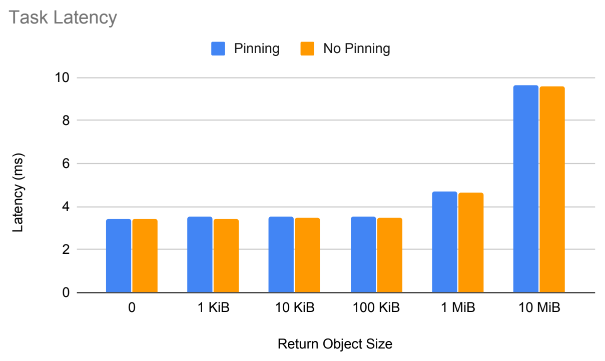
Evaluation

The changes outlined in chapters 4 and 5 are implemented in Ray as a combination of C++ and Python code. Each worker process and raylet runs a gRPC [4] server and makes remote procedure calls to other worker processes and raylets using asynchronous gRPC unary calls over TCP. Application references are tracked using Python's built-in object reference counting mechanisms. These changes were included across multiple official releases of Ray from v0.8.0 to v0.8.5. Experiments evaluating the changes presented in this work are run using Ray v0.8.5 [6] and Ray v0.7.7 is used as a baseline [5].

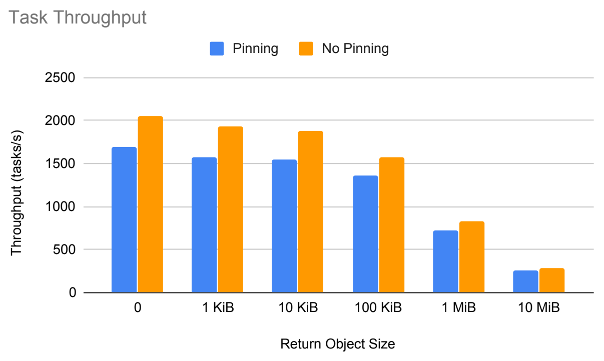
6.1 Object Pinning Overhead

This section evaluates the overhead of the object pinning mechanism described in Section 5.2. Two separate experiments are run to evaluate the impact on latency and throughput. To isolate the overhead of object pinning, each experiment is run across two conditions: with object pinning enabled and disabled. Object pinning is disabled by modifying worker processes to not send an RPC to their local raylet when they create an object. This will also prevent the raylet from sending RPCs to the owner process to wait for an object to go out of scope. Additionally, object inlining is disabled for these experiments, so even objects below the 100 KiB threshold are created in the shared memory object store. Both experiments described below consist of two AWS m4.xlarge instance nodes, each hosting a single Ray actor.

To measure latency, the first actor synchronously calls a remote method



(a) The impact of object pinning on single-task latency. One actor synchronously makes remote method calls to another, which returns an object of the given size.

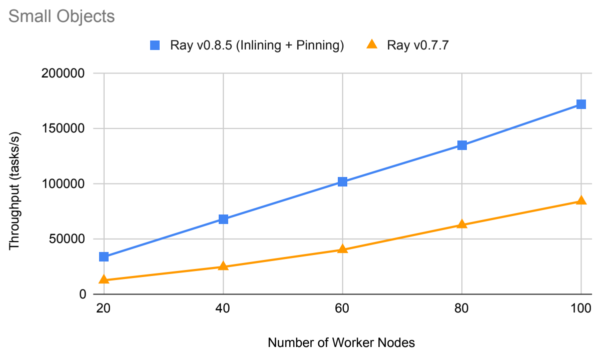


(b) The impact of object pinning on task throughput. One actor asynchronously submits batches of 1000 remote method calls on another and then fetches their results. Each call returns an object of the given size.

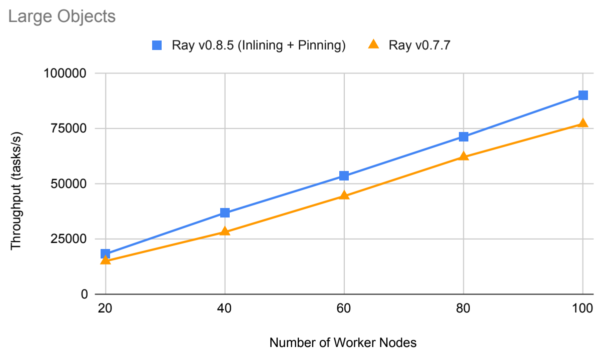
Figure 6.1: Overhead of object pinning. Each experiment is run both with object pinning enabled and disabled. Object pinning is disabled by modifying worker processes to not send an RPC to their local raylet to pin an object when they create it. Object inlining is disabled.

on the second actor and fetches the result. To isolate the overhead of the pinning protocol on the second actor writing and the first actor fetching the return object, the remote method call has no input and returns a single object whose size is varied. Given that the object pinning protocol is asynchronous from the perspective of both actors, we expect a very minimal impact on latency: the only overhead on latency should be the cost of the second actor sending an additional RPC to its local raylet to pin the object after creating it. Additionally, we expect this overhead to be static with respect to the size of the return object. The results of this experiment are shown in Figure 6.1 (a). As expected, the impact of object pinning on latency is minimal, with an average of 69 microseconds, and nearly constant across conditions.

To measure throughput, the first actor asynchronously submits a batch of 1000 independent remote method calls on the second actor and then fetches their results. In this case, we expect more overhead than observed for the latency experiment. When the raylet pins each object, it sends a message to the local object store. This increases the load on the object store process, which is the bottleneck both for creating return objects from the second actor



(a) Zero-sized objects.



(b) 1 MiB objects.

Figure 6.2: Impact of direct RPC task submission, object inlining, and object pinning on overall performance. A varying number of driver processes submit chains of tasks to a varying number of worker nodes. One driver process is allocated for every five worker nodes. Each driver process is placed on its own node in addition to the number of worker nodes listed.

and retrieving them from the first. The overhead should be most impactful for small object sizes, because for larger objects the actual data transfer will dominate the runtime. The results of this experiment are shown in Figure 6.1 (b). Pinning decreases throughput for this workload by from 17% for zero-sized objects to 10% for 1 MiB objects. Note that here we have object inlining disabled to examine the impact of object pinning, but in practice there would be no object pinning overhead for objects 100 KiB and smaller (which were shown here to be impacted the most).

6.2 Throughput and Scalability

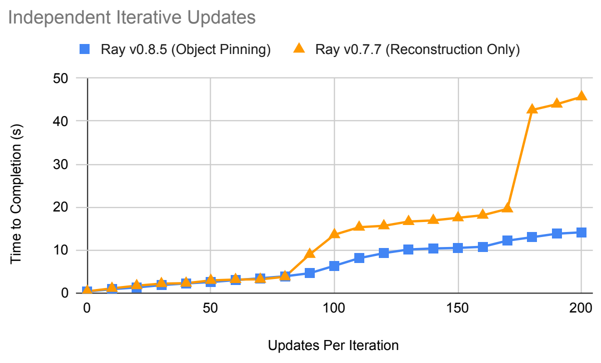
This section evaluates the overall impact that direct RPC task submission, selective inlining, and object pinning have on throughput and scalability for fine-grained tasks. This is evaluated by running a large scale-experiment where a number of driver processes submit chains of tasks to a number of worker nodes (AWS m4.xlarge instances). Each task returns a single output, either zero-sized or 1 MiB, and takes as input the output of the previous task in the chain. The number of driver processes is varied alongside the number of worker nodes, with each driver process submitting tasks to its own set of five worker nodes. To minimize resource contention, each driver process is allocated on its own worker node.

The results of this experiment are shown in Figure 6.2. For both small and large objects, Ray v0.8.5 scales linearly with the number of drivers and workers and Ray v0.7.7 scales nearly linearly. For the small objects case, object inlining drastically improves performance, as the throughput of Ray v0.8.5 outperforms Ray v0.7.7 by a margin of 2.72x for the 20-node condition and 2.05x for the 100-node condition. For large objects, despite the additional overhead of object pinning demonstrated in Section 6.1, Ray v0.8.5 still outperforms Ray v0.7.7 by a margin of 1.22x for the 20-node condition and 1.17x for the 100-node condition. This improvement is due to the direct RPC task submission removing the remote raylet as a bottleneck.

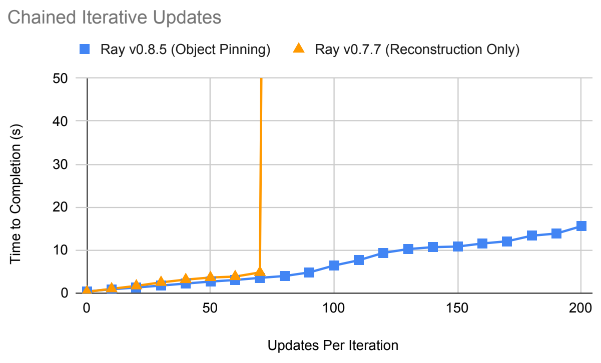
6.3 Benefits of Exact Reference Counting

This section evaluates the performance and correctness benefits of exact object reference counting. As illustrated in Chapter 5, there are some applications that are adversely impacted by the per-node Least Recently Used object eviction employed by Ray v0.7.7. Here, we quantify this impact using a simple workload representative of iterative data processing (e.g., k-means clustering or stream processing). The workload runs 50 iterations, where each iteration updates a 1 MiB cumulative object with the results of processing a varied number of 1 MiB records. This is run for two conditions: one where a new cumulative object is generated for each iteration and one where the same cumulative object is updated repeatedly across iterations. The workload is run on a single AWS m4.16xlarge (64-core) instance with 100 MiB allocated for the shared memory object store.

The results of this experiment are shown in Figure 6.3. First examining the results of the independent iterations experiment, running with object pinning the runtime of the workload scales linearly with the number of updates processed per iteration. The same is true of the condition without object pinning until the total size of the updates processed per iteration exceeds the allocated size of the object store. When this happens, LRU eviction evicts the



(a) Each iteration is independent.



(b) Each iteration depends on the output of the previous iteration. Ray v0.7.7 crashes for all cases with 70 or more updates per iteration.

Figure 6.3: Iterative data processing workload. Each iteration, one cumulative object is updated with the results of processing a number of updates. All objects are 1 MiB each, the workload is run for 50 iterations for each condition, and the number of updates per iteration is varied. The workload is run on a single AWS m4.16xlarge instance with 64 cores and 100 MiB allocated for the shared memory object store.

cumulative object as well as some of the updates whose results have not yet been processed. This causes the system to reconstruct the objects that were evicted but still in use, causing redundant work and slowing the application. This causes significant performance degradation, with the workload taking 3.2x longer to complete than with pinning for 200 updates per iteration.

For the chained update case, where each iteration depends on the output of the previous, the negative impact of this behavior is amplified. In this case, if the cumulative object is evicted for the condition without pinning, the system must not only re-process updates from the current iteration, but also potentially many previous iterations as well. Reconstructing these previous iterations may in turn force more evictions, causing the system to enter into a potentially infinite loop of reconstruction. This causes the condition without pinning to completely fail to complete for all experiments with 70 or more updates per iteration.

Chapter 7

Conclusion

This work describes a new architecture for object management in a distributed, futures-based task execution system. By dynamically selecting between inlining values object values in direct RPC messages and sending references to objects in a shared memory object store, the system simultaneously achieves similar performance to low-level mechanisms for small objects and high efficiency using shared memory for large objects. Further, the system supports exact reference counting for objects in shared memory, supporting a wide range of applications that would not be supported by more rudimentary eviction-based approaches. The reference counting approach is tightly coupled to the programming language, with reference counting metadata distributed throughout the cluster in the processes that require it rather than in a centralized master or scheduler. This decentralized approach is demonstrated to achieve high-performance and scalability, even for workloads with many submitting and executing processes distributed throughout the cluster.

Limitations and Extensions

Much of the overhead incurred by the object pinning mechanism shown in Section 6.1 is due to the fact that the data plane of the shared memory object store on each node, the object store process, is located separately from the control plane, the raylet process. Simply merging the object store and raylet processes would remove the need for expensive interprocess communication between the two while also reducing the amount of interprocess communication for driver and worker processes. Pinning overhead could be further reduced by simply merging the object store and raylet processes, which is feasible given that they provide a single logical interface to the application.

Additionally, one key assumption made in this work is that the programming model does not allow an object reference to escape the process that created it. That is, even when processes send objects by-reference through the object store, the receiving process is only exposed to the value of the object and cannot directly pass a reference to it to another process without making a copy. However, this behavior may be restrictive for some applications as it may require more complex control flow in order to coordinate efficient data transfer between processes that are not the owner of an object. The protocol could be expanded to include passing references between processes using existing distributed reference counting techniques [8].

Finally, one limitation of the object pinning protocol presented in Section 5.2 is that the initial copy of an object is always pinned on the node that it was created on. In some situations, this may be limiting. For example, consider the case where a single task generates the data dependencies for many dependent tasks. In this case, the total size of the output objects may exceed the size of the shared memory object store of the node the task is running on, causing the task to fail despite there being sufficient memory in the cluster for the objects. This can be addressed in the application by splitting the generating task into multiple tasks, each one generating some subset of the objects, but this requires the application to treat the capacity of object store on each node independently instead of having a single logical abstraction. A clear extension of this work would be to enable pinned copies of objects to be dynamically relocated using some form of handoff protocol between the raylets on each machine.

Bibliography

[1] Akka. `https://akka.io/`.

[2] Notes on Python Cyclic References. `https://hg.python.org/cpython/file/4e687d53b645/Modules/gc_weakref.txt`.

[3] PyTorch - Remote Reference Protocol. `https://pytorch.org/docs/stable/notes/rref.html`.

[4] grpc - a high-performance, open source universal rpc framework. `https://grpc.io`, 2020.

[5] Ray v0.7.7 release. `https://github.com/ray-project/ray/tree/ray-0.7.7`, 2020.

[6] Ray v0.8.5 release. `https://github.com/ray-project/ray/tree/ray-0.8.5`, 2020.

[7] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI). Savannah, Georgia, USA*, 2016.

[8] Saleh E. Abdullahi and Graem A. Ringwood. Garbage collecting the internet: A survey of distributed garbage collection. *ACM Comput. Surv.*, 30(3):330–373, September 1998.

[9] Ole Agesen, David Detlefs, and J. Eliot Moss. Garbage collection and local variable type-precision and liveness in java virtual machines. In

Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation, PLDI '98, page 269–279, New York, NY, USA, 1998. Association for Computing Machinery.

[10] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, Jie Chen, Jingdong Chen, Zhijie Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Ke Ding, Niandong Du, Erich Elsen, Jesse Engel, Weiwei Fang, Linxi Fan, Christopher Fougner, Liang Gao, Caixia Gong, Awni Hannun, Tony Han, Lappi Johannes, Bing Jiang, Cai Ju, Billy Jun, Patrick LeGresley, Libby Lin, Junjie Liu, Yang Liu, Weigao Li, Xiangang Li, Dongpeng Ma, Sharan Narang, Andrew Ng, Sherjil Ozair, Yiping Peng, Ryan Prenger, Sheng Qian, Zongfeng Quan, Jonathan Raiman, Vinay Rao, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Kavya Srinet, Anuroop Sriram, Haiyuan Tang, Liliang Tang, Chong Wang, Jidong Wang, Kaifu Wang, Yi Wang, Zhijian Wang, Zhiqian Wang, Shuang Wu, Likai Wei, Bo Xiao, Wen Xie, Yan Xie, Dani Yogatama, Bin Yuan, Jun Zhan, and Zhenyao Zhu. Deep speech 2 : End-to-end speech recognition in english and mandarin. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 173–182, New York, New York, USA, 20–22 Jun 2016. PMLR.

[11] David G Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. Fawn: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 1–14, 2009.

[12] Michael Armbrust. SPARK-20928: Continuous Processing Mode for Structured Streaming. `https://issues.apache.org/jira/browse/SPARK-20928`, 2017.

[13] Joe Armstrong. *Making reliable distributed systems in the presence of software errors.* PhD thesis, Mikroelektronik och informationsteknik, 2003.

[14] Ken Arnold, James Gosling, and David Holmes. *The Java programming language*. Addison Wesley Professional, 2005.

[15] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobbler, Michael Wei, and John D Davis. {CORFU}: A shared log design for flash clusters. In *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, pages 1–14, 2012.

[16] John K Bennett, John B Carter, and Willy Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 168–176, 1990.

[17] Phil Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. Orleans: Distributed virtual actors for programmability and scalability. Technical Report MSR-TR-2014-41, 3 2014.

[18] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015.

[19] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[20] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, pages 401–414, 2014.

[21] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, 9 2004.

[22] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, page 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[23] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[24] Richard E Jones and Rafael D Lins. Cyclic weighted reference counting without delay. In *International Conference on Parallel Architectures and Languages Europe*, pages 712–715. Springer, 1993.

[25] M. I. Jordan and T. M. Mitchell. Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245):255–260, 2015.

[26] Pete Keleher, Alan L Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. *Distributed Shared Memory: Concepts and Systems*, pages 211–227, 1994.

[27] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[28] Kai Li. Ivy: A shared virtual memory system for parallel computing. *ICPP (2)*, 88:94, 1988.

[29] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph E. Gonzalez, Michael I. Jordan, and Ion Stoica. RLlib: Abstractions for distributed reinforcement learning. In *International Conference on Machine Learning (ICML)*, 2018.

[30] Eric Liang, Hang Zhu, Xin Jin, and Ion Stoica. Neural packet classification. In *Proceedings of the ACM Special Interest Group on Data*

Communication, SIGCOMM '19, page 256–269, New York, NY, USA, 2019. Association for Computing Machinery.

[31] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E. Gonzalez, and Ion Stoica. Tune: A research platform for distributed model selection and training. *CoRR*, abs/1807.05118, 2018.

[32] Adam H. Marblestone, Greg Wayne, and Konrad P. Kording. Towards an integration of deep learning and neuroscience. *bioRxiv*, 2016.

[33] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications.

[34] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. CIEL: A universal execution engine for distributed data-flow computing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 113–126, Berkeley, CA, USA, 2011. USENIX Association.

[35] B. Nitzberg and V. Lo. Distributed shared memory: a survey of issues and algorithms. *Computer*, 24(8):52–60, 1991.

[36] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, et al. The RAMCloud storage system. *ACM Transactions on Computer Systems (TOCS)*, 33(3):7, 2015.

[37] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. 2017.

[38] Alex Poms, Will Crichton, Pat Hanrahan, and Kayvon Fatahalian. Scanner: Efficient video analysis at scale. *ACM Trans. Graph.*, 37(4):138:1–138:13, July 2018.

[39] Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In Kathryn Huff and James Bergstra, editors, *Proceedings of the 14th Python in Science Conference*, pages 130 – 136, 2015.

[40] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[41] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[42] Stephanie Wang, John Liagouris, Robert Nishihara, Philipp Moritz, Ujval Misra, Alexey Tumanov, and Ion Stoica. Lineage stash: Fault tolerance off the critical path. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 338–352, New York, NY, USA, 2019. Association for Computing Machinery.

[43] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2012.

[44] Paul R Wilson. Uniprocessor garbage collection techniques. In *International Workshop on Memory Management*, pages 1–42. Springer, 1992.

[45] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[46] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache Spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65, October 2016.