Lineage Stash: Fault Tolerance Off the Critical Path

Stephanie Wang
UC Berkeley

John Liagouris
ETH Zurich

Robert Nishihara
UC Berkeley

Philipp Moritz
UC Berkeley

Ujval Misra
UC Berkeley

Alexey Tumanov
Georgia Institute of Technology

Ion Stoica
UC Berkeley

Abstract

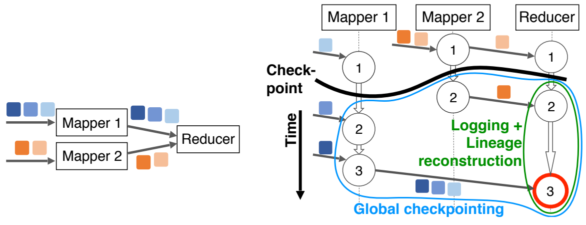
As cluster computing frameworks such as Spark, Dryad, Flink, and Ray are being deployed in mission critical applications and on larger and larger clusters, their ability to tolerate failures is growing in importance. These frameworks employ two broad approaches for fault tolerance: checkpointing and lineage. Checkpointing exhibits low overhead during normal operation but high overhead during recovery, while lineage-based solutions make the opposite tradeoff.

We propose the *lineage stash*, a decentralized causal logging technique that significantly reduces the runtime overhead of lineage-based approaches without impacting recovery efficiency. With the lineage stash, instead of recording the task's information *before* the task is executed, we record it asynchronously and forward the lineage along with the task. This makes it possible to support large-scale, low-latency (millisecond-level) data processing applications with low runtime and recovery overheads. Experimental results for applications in distributed training and stream processing show that the lineage stash provides task execution latencies similar to checkpointing alone, while incurring a recovery overhead as low as traditional lineage-based approaches.

ACM Reference Format:
Stephanie Wang, John Liagouris, Robert Nishihara, Philipp Moritz, Ujval Misra, Alexey Tumanov, and Ion Stoica. 2019. Lineage Stash: Fault Tolerance Off the Critical Path. In *ACM SIGOPS 27th Symposium on Operating Systems Principles (SOSP '19), October 27–30, 2019, Huntsville, ON, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3341301.3359653

1 Introduction

Recent data processing applications in domains ranging from stream processing [9, 28] to reinforcement learning [27] have become increasingly *online* and user-facing, making the need for low latency as critical as the need for high throughput. The *dynamic dataflow graph* [7, 27, 29] is a flexible computation model that is ideal for developing large-scale online data processing applications because it can support both batch processing [29] and fine-grained stateful computation [27]. In this model, a program expresses task parallelism through asynchronous function invocations, called *tasks.* Tasks may be stateless, i.e. free of side effects, or stateful, i.e. bound to a specific process. Figure 1 shows an example stream processing application as a dynamic dataflow in which operators can dynamically push records to downstream operators.



(a) Logical.

(b) Physical.

Figure 1. A streaming mapreduce. **(a)** Logical representation. Mappers compute a stateless function over each record (rounded box) in the input and output the results to a Reducer. **(b)** Physical representation, as a dynamic dataflow. Solid arrows show data dependencies (record batches). White arrows show stateful dependencies [27], determined by the execution order on a given process. Mappers do not have application state, but they are stateful because they can buffer records and dynamically push them to Reducer by submitting tasks, which get executed in a nondeterministic order. Reducer fails during task 3 (red), and outlined tasks must be re-executed to preserve exactly-once semantics. Lineage reconstruction (green) exactly reconstructs Reducer by replaying its inputs since the last checkpoint in the same order. Global checkpointing (blue) re-executes *all* processes' tasks since the last checkpoint, possibly in a different order (e.g., Reducer may execute task 3 before 2).

Guaranteeing fault tolerance without sacrificing low latency during normal operation is an open challenge for dynamic dataflows when tasks are fine-grained, i.e. milliseconds long. This is a challenge because many applications require exactly-once semantics, i.e. all data inputs are reflected in the final output exactly once, for global consistency.

There are two general techniques for guaranteeing global consistency after a failure: logging and global checkpointing (Fig. 1b). With logging, the system durably logs the job's

SOSP '19, October 27–30, 2019, Huntsville, ON, Canada

ACM ISBN 978-1-4503-6873-5/19/10…$15.00
https://doi.org/10.1145/3341301.3359653

computation and intermediate application data, and in the event of failure, *exactly* replays the computation to recover lost state. With global checkpointing, the system takes periodic application checkpoints, and in the event of failure, reruns the job from the latest checkpoint to a consistent but possibly different state (due to nondeterministic execution).

These differences have fundamental implications for the runtime and recovery overheads for data processing applications. In general, logging-based techniques incur a higher overhead during normal operation because they must record information during execution, but lower overhead during recovery, as they can use this information to reduce the amount of computation that must be replayed (Fig. 1b). Many systems for data processing [15, 35, 36] log the *lineage*, or the computation graph, but not the intermediate data to lower the runtime overhead of logging, at the cost of having to reconstruct lost intermediate data in case of failure. Still, the runtime overhead of lineage-based reconstruction has so far restricted its applicability to *coarse-grained* tasks. Global consistency requires that the lineage of each task be durably logged *before* execution, which requires replication to at least one remote node to tolerate non-transient failures. This would add significant overhead to dynamic dataflow graphs, since tasks are often both small and dynamically generated.

Many systems for fine-grained data processing, including Naiad [28] and Flink [9], rely on global checkpointing because it is easy to understand and adds low runtime overhead. Other than the overhead of the checkpoint itself [33], which can be reduced through asynchronous checkpointing [10, 12, 24], this approach adds minimal runtime overhead because there is no need to record execution. On the other hand, recovery requires a coordinated global rollback of the entire system to the latest checkpoint [18], which is expensive at large scale [34]. This is because previous work that is unaffected by the failure must be rolled back for consistency (Fig. 1b), and new work cannot be accepted until recovery is complete. Also, because global checkpointing alone does not promise exact re-execution, guaranteeing exactly-once semantics for interactions with the outside world adds significant runtime overhead, since every such interaction requires a checkpoint to ensure it is never rolled back [18].

In this paper, we introduce the *lineage stash*, a decentralized logging technique for dynamic dataflows that simultaneously achieves low recovery overhead and low runtime overhead. Like previous lineage-based systems, we rely on lineage reconstruction for fast recovery and low downtime. However, unlike these systems, the lineage stash doesn't require a task's lineage to be stored *before* the execution of the task. This removes the lineage overhead from the critical path during normal operation.

The main idea behind the lineage stash is that instead of storing the lineage in a reliable store on the critical path of execution, one can *forward the full lineage along with every task invocation.* Then, if the system needs to execute a task with a missing input (e.g., because of a failure), the worker running the task has full information about which upstream tasks need to be re-executed to reconstruct the missing input. Of course, this straw man solution is not practical as the lineage can grow very large, and the overhead of forwarding it can be prohibitive. To make this solution practical, we *asynchronously* store the lineage and forward only the most recent part which has not been durably stored yet. In particular, each worker keeps a lineage *stash* in local memory containing all tasks that it has seen recently. Each worker then runs a local protocol to flush its stash to a remote reliable store. Since flushing is asynchronous, it has negligible impact on application latency during normal operation.

The lineage stash is an example of causal logging [5, 17], a class of recovery techniques for message-passing systems in which processes asynchronously log nondeterministic events. The key challenge is to identify the minimum set of events that need to be logged such that we can guarantee *global consistency* after recovery while also guaranteeing *predictably low task latency* during normal operation. A naive logging approach could add prohibitive runtime overhead. For instance, one could log all messages, but in data processing, these messages can be arbitrarily large. The lineage stash minimizes the amount logged by exploiting the fact that the computation in data processing is usually deterministic, while the nondeterministic events can usually be encapsulated by the order of execution. For example, in Fig. 1b, the application's map and reduce functions are deterministic, but the order of task submission and execution is not.

In the lineage stash work, we extend ideas from both lineage reconstruction and causal logging to make them practical for large-scale, low-latency data processing. In particular, we identify the nondeterministic events that must be logged for application correctness and design an efficient protocol to store this information off the critical path of execution. We implement the lineage stash on Ray [27], a distributed framework for dynamic dataflows, and demonstrate the benefit on two representative applications in stream processing and distributed training. Whereas previous systems for these applications can achieve either low latency or low recovery time, we show that the lineage stash can achieve *both.* Thus, we present the following contributions:

1. An analysis of the nondeterministic events that must be logged in data processing applications.
2. A log storage architecture that enables simple, scalable protocols for flushing the stash and recovery.
3. The lineage stash: a causal logging technique that achieves low runtime *and* recovery overheads for fine-grained data processing applications.

2 Background

We present a case study of a stream processing application, which represents an important class of large-scale online data processing applications. We show how such applications can be expressed and executed as a dynamic dataflow, and present the open challenges in the proposed approach.

2.1 Case Study: Stream Processing

Stream processing provides the abstraction of *continuous operators* that compute a long-running query over an infinite stream of data items, or records. Each operator consumes one or more input streams and produces an output stream. This imposes a set of requirements that is representative of large-scale fine-grained data processing applications.

First, stream processing applications have stringent performance requirements during normal operation, requiring both high-throughput data processing, because of the often large data ingest, and low latency, as the query result will change over time and is generally desired as soon as possible.

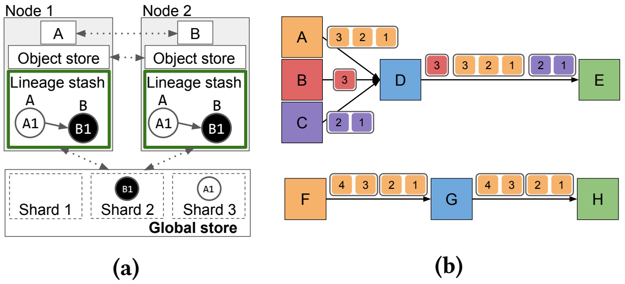
Second, because stream processing applications often run online and query results are needed as soon as possible, applications are sensitive to recovery time. Especially at large scale, when the chance of a failure is greater, it is critical that applications experience little downtime after partial failures.

Finally, the types of computation performed vary widely even in a single application, which has implications on recovery correctness [23]. While much of the data processing computation may be deterministic (i.e. a function of the input stream), typically a stream processing application will also include local state, such as a sink operator that maintains query results, as well as interactions with the external world, such as a sink operator that triggers an alert after a specified query result. A deterministic computation can be safely re-executed many times, but computations with side effects on the outside world often require exactly-once semantics, since the outside world in general cannot be rolled back.

2.2 System Model and Challenges

Existing systems for stream processing fall under two categories. Systems like Flink [9] and Naiad [28] use global checkpointing for fault tolerance and instantiate physical instances of continuous operators, each of which consumes and produces buffers, or *batches*, of records. This allows for low-latency, record-at-a-time processing. In contrast, systems like Spark Streaming [37], execute *synchronous stages* over fixed-size partitions of the input stream, and record the *lineage* of each stage for fault tolerance.

Stream processing applications can be represented as a dynamic dataflow (Fig. 1), with both *continuous operators*, as in Flink [9] or Naiad [28], and lineage-based recovery, as in Spark Streaming [37]. Each continuous operator is instantiated as a *process* with local state that can execute *tasks*, also known as methods or message handlers, submitted by upstream operators. Each task's argument is a record batch. Processes execute tasks as input batches become available (Reducer in Fig. 1), and can flush batches to downstream processes by dynamically submitting tasks (Mappers in Fig. 1), e.g., based on the maximum output buffer size.



(a)

(b)

Figure 2. (a) Lineage stash architecture, on top of a decentralized dataflow scheduler. *A* and *B* are processes that can submit tasks to each other (e.g., *A*1 submits *B*1). Dotted arrows show the protocols used to communicate between nodes. **(b)** Stream processing. *D* is a *nondeterministic* operator that reads dynamically sized batches (buffers) from multiple input sources (*A*, *B*, *C*) in any order and outputs results to downstream operator *E*. *G* is a *deterministic* operator that reads statically sized batches from a single source, *F*.

The lineage in this model is recorded at the granularity of a batch. This is in contrast to Spark Streaming [37], which records lineage at the granularity of partitions, each of which may span many batches. In Fig. 1, we show how the lineage of each batch is tracked, through data dependencies (solid arrows), and stateful dependencies (white arrows). Data dependencies are specified by the application through task arguments, while stateful dependencies are created between tasks that execute consecutively on the same process. The use of lineage can reduce downtime during recovery, as intact operators can continue processing records while lost operators can be replayed exactly from the lineage.

To execute this dataflow graph, we adopt the system model introduced by Ray [27], in which a distributed scheduler dispatches tasks to local worker processes based on their data and stateful dependencies (Fig. 2a). Each Ray node can host multiple worker processes, which may be stateful (known as *actors* [27]). Worker processes on the same node also share an in-memory *object store*, which can be used to cache immutable copies of large task outputs. System metadata, such as task descriptions and object locations, is stored in a logically centralized *global store*, which can be sharded for scalability and replicated for durability.

There are a number of challenges in applying lineage reconstruction to this setting. First, the granularity at which lineage is recorded is much finer than in previous lineage-based systems, at the level of batches that can take milliseconds to process, compared to synchronous stages that can take seconds. Since the lineage is both significantly larger and updated more frequently than in existing lineage-based systems, the common approach of logging lineage to a centralized location [15, 35, 36] on the critical path of task execution would affect both task latency and throughput.

Second, the lineage is not only larger, it must also be updated at *runtime* to guarantee exactly-once record processing. This is because asynchronous record processing introduces *nondeterministic events* when an operator processes data from multiple sources. For example, in Fig. 2b, operator *D* processes data from operators *A*, *B*, and *C* as they become available, then outputs results to *E*. If *D* fails but *E* remains active, then we must guarantee that when reconstructing *D*'s outputs, we do so in an order consistent with what *E* has seen so far. This necessitates reliably recording the order in which *D* processed its inputs during execution, which adds latency if this must be done before *E* can process the results. Note that this is not an issue in lineage-based systems that execute in synchronous stages [37]; in such systems, *D* would block results to *E* until it has processed a predetermined number of records from *A*, *B*, and *C*. Nor is it an issue when reading from a single input, as *G* does in Fig. 2b.

These problems motivate an *asynchronous* logging approach, in which task specifications are logged to a centralized reliable storage system, but off the critical path of task execution. In particular, each node logs lineage directly to a local, in-memory *lineage stash* (Fig. 2a), which is asynchronously flushed to the global store. However, this solution presents a third challenge: maintaining the decentralized state. The decentralized logging approach complicates both normal operation, as it creates local state that must be flushed, and recovery, as a failed operator's lineage is no longer guaranteed to be in a centralized location. The final challenge is thus in designing simple protocols for flushing local state and recovering after a failure.

In summary, the challenges are: (1) removing the cost of recording lineage from the critical path of task execution, (2) efficiently recording nondeterministic events, and (3) designing simple, scalable protocols for flushing and recovering lineage. In the remainder of this paper, we describe the lineage stash design and how it meets these challenges.

3 Lineage Stash Overview

The data processing applications that the lineage stash supports can be viewed logically as message-passing systems, a low-level abstraction in which a set of processes with local state communicate with each other by sending and receiving messages. For example, the continuous operators in a stream processing application can be viewed as a set of processes where each message contains a single record. In this section, we describe the relationship between a message and a *task*.

The lineage stash is a form of *rollback recovery* [18], in which information is recorded during execution to minimize the amount of work that must be redone after a failure. Informally, the lineage stash guarantees that if a process fails, then any messages that it received since its last checkpoint will be replayed in the same order. This implies that the system will recover to a *globally consistent state*–that is, for every message that has been delivered to a process, the corresponding event is reflected at the sender. This in turn implies exactly-once semantics for the application (e.g., every record is processed once in stream processing). The lineage stash can also support *end-to-end* exactly-once semantics for when the application outputs a result to the external world. As is standard in rollback recovery [18], the lineage stash targets the fail-stop model [31] and assumes that the application can identify and record any *nondeterministic* events, as well as inputs and outputs to the external world.

At a high level, we use a *causal logging* approach for recording and replaying computation. Causal logging [5, 17] is a technique that aims to lower both recovery and runtime overhead by logging asynchronously to a *stable storage* system, i.e. the Ray system metadata store [27] or the Spark scheduler [36]. Each process buffers a log of all nondeterministic events (e.g., "received message *m*") that caused its current state and piggybacks any volatile records onto its messages to other processes. If a process fails, then it can retrieve logs from the remaining processes to guide its recovery. Since all nondeterministic events from the initial execution can be replayed from the logs, this guarantees global consistency.

However, practical use of causal logging in general message-passing applications remains challenging due to the sheer variety of nondeterministic events that could occur (e.g., writing to external memory, executing on a timer, etc.). Correctness requires that *all* such events are logged during execution, which can be cumbersome, expensive, or both. On the other hand, data processing applications by nature consist of mostly *pure* computation, i.e. side effect-free and the outputs are a deterministic function of the inputs. This makes causal logging a promising approach to providing rollback recovery for decentralized data processing applications. The key system challenge is then to identify and efficiently capture the sources of nondeterminism that do occur in data processing applications. There are three questions to answer: *what* information do we log, *how* do we log that information, and *how* do we recover the initial execution from these logs?

What information should we log? A general logging approach is to reliably record every message that every process receives, including the content and the execution order. Then, assuming deterministic message handlers, recovery is simply a matter of retrieving and replaying the logs.

However, this straw man approach is clearly expensive for data processing applications. The total message content in data processing applications can be much larger than the description of the computation. This is true in communication primitives like allreduce for large arrays, as the array is often much larger than the description of the reduce function. Other data processing applications consist instead of many small messages that all undergo the same computation on the receiver. An example of this is stream processing:

logically, operators execute record-at-a-time, but physically, many records are batched together for efficiency. In this case, the execution order only needs to be logged at batch boundaries. In the allreduce case, the message ordering is deterministic, so it need not be recorded at all.

Fortunately, the message *content* in data processing applications is often the output of a deterministic computation performed by the sender. Thus, it can be perfectly recomputed, assuming the same inputs and sender state. This allows for a key optimization: recording the *lineage* instead of the raw data. In particular, we reliably store a pointer to the application data, called an *object*, and a concise description of the computation, called a *task*. Each task can take as input a process's local state and one or more objects, and can generate objects (return values) as well as other tasks (nested functions). The lineage of an object comprises the task that created it and the lineage of each of the task's arguments. Since object values are deterministic, we can cache multiple immutable copies of the object across nodes. As in previous systems [36], this comes at the cost of having to recompute objects during recovery if all copies are lost. For small enough objects, the data can optionally be inlined in the task specification.

In some cases, nondeterminism is actually key to application functionality and performance. In particular, the ability to dynamically execute tasks based on data availability at runtime is essential for low latency in applications where a single process executes tasks from multiple other processes, such as in stream processing (Fig. 2b). In this case, if another process sees the result, then the task execution order must be recorded in the lineage and made durable in case of failure. Otherwise, the nondeterministic process may recover to a state inconsistent with the witness, or *orphan process* [18]. For instance, in Fig. 3a, processes *A* and *B* submit tasks concurrently to *C*, where they can be executed in any order, and the result is seen by *D*. If *C* were to fail, we must replay the tasks from *A* and *B* in the same order as before to guarantee consistency with *D*.

However, there are also applications where it is sufficient for a process to execute tasks in a deterministic order. For example, communication primitives like allreduce are *fully deterministic* because every process receives tasks from one other process. Determinism can also be enforced for a process with multiple callers if tasks are always executed in a specific order, e.g., round-robin. While this is more restrictive to applications, it does allow for more efficient logging, since it is not necessary to record task execution order. For instance, in Fig. 4a, *C* is the only process to submit tasks to *A*, so the order of tasks that *A* executes is deterministic and need not be recorded. Note that it is possible to mix different logging levels in a single application, i.e. one process may execute tasks dynamically while others are fully deterministic.

In rare cases, a process may also execute nondeterministic events *during* a task. For instance, a stream processing application with strict latency requirements could choose to release outputs based on a timer. To support this case, we also allow the application to record such events as part of the task description so that they can be replayed exactly after a failure. While we provide system support, it is up to the application to identify and replay such events; in practice, we expect this to be done in application-level libraries.

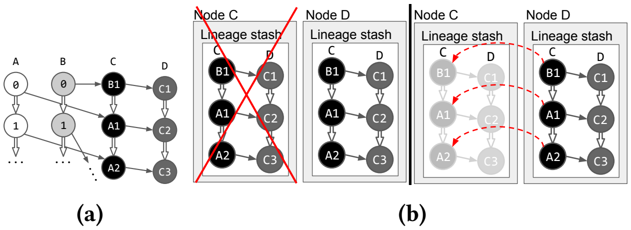


Figure 3. (a) A nondeterministic application and **(b)** a failure scenario showing why lineage must be forwarded. Because *C* executes tasks from *A*, *B* in a nondeterministic order, it must retrieve its lineage from *D* after a failure, shown by the red dashed arrows.

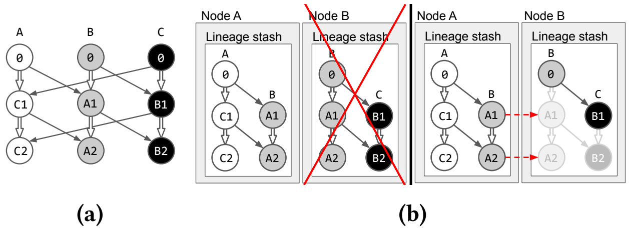


Figure 4. (a) A deterministic application and **(b)** a failure scenario showing what lineage must be remembered. To recover *B* after a failure, *A* simply resubmits (red dashed arrows) its previous tasks.

How should we log information? While recording the lineage rather than the data greatly reduces the cost of logging, the rate at which tasks are generated can still be very high for fine-grained, decentralized applications. Thus, storing a task description reliably must be done off of the critical path. The main idea behind the lineage stash is to use a *causal logging* approach, where instead of storing the lineage reliably before the task is executed, we *forward the lineage of each of the task's inputs with the task invocation*. This way, the node executing the task has all the information to reconstruct the task's inputs, if necessary. Each node remembers the lineages of tasks that it generated or received for execution in a local store, called the *lineage stash*.

Only the nondeterministic events must be reliably recorded for recovery correctness. Thus, in deterministic applications, the lineage need not be forwarded during normal operation because it can be deterministically recreated after a failure. Each process only needs to *remember* the tasks that it has submitted, by storing them in its local lineage stash. If a process fails, then it recovers by simply asking the remaining processes to resubmit their stashed tasks. Figure 4b shows this for the deterministic application example in Fig. 4a: during normal operation, *A* remembers the tasks that it submits to *B* in its local lineage stash. When *B* dies, *A* resubmits its stashed tasks (*A*1, *A*2) to recover *B*.

In nondeterministic applications, each process must also *forward* the lineage that it has seen so far. This is because task descriptions are updated during execution based on nondeterministic events, such as the order of task execution. If a process fails, then it must recover the most recent copy of its tasks. For example, in Fig. 3a, because *C* executes tasks in a nondeterministic order, it must also forward its own lineage to *D* during execution. This is so that if *C* dies, as in Fig. 3b, *D* can resubmit these tasks to *C* in the correct order.

While this simple scheme of remembering and forwarding recent lineage removes the lineage store from the critical path of a task's execution, unfortunately it will not scale for realistic applications as the lineage can become very large and forwarding it can be prohibitive. Thus, timely flushing of the local stash is critical to maintaining predictably low task latency. Traditionally in rollback recovery systems, each process can asynchronously flush its volatile log to an individual stable storage system, which may be remote [17, 18, 30]. However, this requires each process to garbage-collect its stable storage and can lead to an unpredictably large storage footprint if a task is forwarded many times.

We simplify garbage collection by asynchronously flushing each stash to a global but physically decentralized (sharded) stable storage system, in which each task has a unique identifier and any process may read or append to any task (Section 4.2.2). This means that only a single copy of each task is reliably stored, and garbage collection of the stable storage can be handled by a single background process, which erases tasks previous to the last global checkpoint. Although it is not our focus in this work, this also facilitates logging for *stateless* tasks that are not bound to a specific process.

Because processes can die before they flush their lineage, task descriptions can be lost entirely before they are written to the global store. However, we guarantee that during normal execution, if a task is not yet in the global store, then all nodes that execute dependent tasks must have the task in their stash. Therefore, if we lose all nodes that have stashed a particular task, we can still guarantee consistency because no live node will have seen the result of the task.

How do we recover the logs? To recover a failed process to a globally consistent state, we must retrieve and re-execute its lineage. As noted above, for deterministic processes, it is enough for the other processes to remember the lineage of tasks that they have submitted so far. These can be resubmitted along with any stashed lineage during recovery (Fig. 4b).

For nondeterministic processes, we must also recover the initial execution order, as well as any nondeterministic events that occured during a task's execution. For example, in Fig. 3a, if *C* fails and its lineage has not yet been written to the global store, it must retrieve its lineage from *D* before it can accept further tasks from *A* and *B*. For an arbitrary application, the relevant lineage could reside at any process, so a failed process would have to retrieve and reconcile a subgraph of its lineage from every other process [17]. This can be expensive and complicated, especially when there are multiple simultaneous failures.

We can simplify the lineage recovery protocol with the global lineage store. Upon a failure, each process simply flushes its local stash to the global store and replies to the recovering process once all writes have been acknowledged. The recovering process can then retrieve its lineage by walking the task dependencies in the global store, starting from the tasks resubmitted by the other processes. Because the global store is indexed by task rather than process, retrieving the lineage is likely slower than if the process's log were stored contiguously. While this may affect recovery performance, it allows for a simple recovery protocol; our implementation required only 125 lines of code (Section 5).

We can further optimize the recovery protocol by leveraging a property common to decentralized data processing applications: most processes send tasks to only a small set of other processes, which changes infrequently. Thus, we only need to contact this set, which is often much smaller than the total set of processes. For example, in Fig. 3b, *C* only needs to retrieve lineage from *D*.

4 Lineage Stash Implementation

First, we expand on the architecture presented in Fig. 2a and introduce the lineage stash protocols, which we present in Section 4.2 along with their guarantees. A process can send a task to a remote process using the lineage stash protocols for forwarding and remembering lineage (Section 4.2.1). Processes can also send or receive objects to or from remote processes through their in-memory object stores. We assume a fail-stop model: if a node fails, then its object store and lineage stash will be lost.

Each lineage stash flushes to a logically centralized *global lineage store*, a reliable key-value store that maps task ID to specification. All operations are over a single task, and we do not assume sequential consistency across tasks, i.e., operations on different tasks from the same node may be processed in any order. This allows us to shard the global store by task ID for horizontal scalability, as in Fig. 2a. Each node communicates with the global store independently to flush its local stash and retrieve lineage during recovery (Section 4.2.2). As an optimization, each node can request an acknowledgement when a given task has been written to the global store.

4.1 Definitions

Next, we describe the lineage structure. All processes, tasks, and objects are assigned a unique identifier (ID), which can be deterministically recomputed during recovery. The task ID is a hash of the sender and receiver IDs and the number

Field	Type	Description
id	TaskID	hash(*receiver, sender, taskCounter*)
version	int	# of updates to the task specification
receiver	ProcessID	ID of process that receives the task
sender	ProcessID	ID of process that sent the task
taskCounter	int	# of tasks sent from *sender* to *receiver*
applicationLog	string[]	nondeterministic events during task
parentId	TaskID	*id* of task that submitted this task
predecessorId	TaskID	hash(*receiver, sender, taskCounter* − 1)
argumentIds	ObjectID[]	object IDs of task arguments
dependencies	TaskID[]	[*parentId, predecessorId*] + *argumentIds*

Table 1. Task specification (*version*, *predecessorId* and *applicationLog* may be updated after task creation to record nondeterminism)

of tasks sent between the pair so far (Table 1), The object ID is a concatenation of the ID of the task that created it and the number of objects that the task has created so far.

The task specification (Table 1) can be monotonically updated to record nondeterministic events. The predecessorId is initially the ID of the previous task that the sender submitted to the destination process. It may be overwritten once, before execution, to the task that the destination process executed immediately beforehand, to reflect the task order. During task execution, the application can also append nondeterministic events to the applicationLog. Each of these updates increments the task's version. To define global consistency, we first define a total order on tasks with the same ID. This also makes it safe to flush any version of a task; the global store simply rejects older versions.

Definition 4.1 (Task order)**.** For tasks T and T' where $T.id = T'.id$, $T \leq T'$ if $T.version \leq T'.version$, $T.applicationLog$ is a prefix of $T'.applicationLog$, and either $T.predecessorId = T'.predecessorId$ or $T.predecessorId$ and $T.version$ are equal to their initial values (Table 1).

Definition 4.2 (Lineage)**.** The lineage of a task T consists of T itself and the lineage of all of its dependencies (Table 1). For convenience, we will also say that the lineage of an object is the lineage of the task that created it, and the lineage of a process is the lineage of the last task that it executed.

$$Lineage(T) = \{T\} \bigcup_{T' \in T.dependencies} Lineage(T')$$

Recovery correctness is defined via global consistency [12], i.e., every message received by a process is also reflected in the sender's history. In terms of lineage, this means that for every task that a process has executed, if the same task appears in some other process's lineage, then the process that executed the task must have the most recent version.

Definition 4.3 (Lineage consistency)**.** For any processes p, q, if $T_p.id = T_q.id$, p executed T_p, and $T_q \in Lineage(q)$, then $T_q \leq T_p$.

Lineage consistency after failure of a process p is guaranteed as follows: if T is the last task executed by p that is in the lineage of any live process, then all tasks in $Lineage(T)$ are either in a live process's stash or in the global store. We ensure this property by forwarding *uncommitted lineage* with each submitted task.

```
def GetUncommittedLineage(stash, T):
  lineage = {}
  for D in T.dependencies:
    if D in stash:
      lineage.add(D)
      lineage.update(GetUncommittedLineage(stash, D))
  return lineage
```

(a) Getting uncommitted lineage from the local stash.

```
def AddUncommittedLineage(stash, T, lineage):
  for D in T.dependencies:
    if D in lineage and D not in stash:
      stash.add(D); AddUncommittedLineage(stash, D, lineage)
```

(b) Receiving uncommitted lineage in the local stash.

Figure 5. Lineage stash methods for getting and receiving a task's uncommitted lineage (Definition 4.4). A practical implementation can easily avoid forwarding duplicate lineage by recording which tasks have been sent to which nodes.

```
def SubmitTask(T):
  stash.add(T); FlushTask(T)
  AssignTask(T, T.receiver, GetUncommittedLineage(stash, T)
    if P.NONDETERMINISTIC else {})
```

(a) Submit a task and forward uncommitted lineage.

```
def AssignTask(T, P, uncommitted_lineage):
  if P.NONDETERMINISTIC:
    T.predecessorId = P.lastTaskId; P.lastTaskId = T.id
    T.version += 1; FlushTask(T)
  stash.add(T)
  AddUncommittedLineage(stash, T, uncommitted_lineage)
```

(b) Assign a task and add the forwarded lineage.

Figure 6. Node methods for task execution. AssignTask also records nondeterministic execution order by updating the task's predecessorId. Nondeterministic events during task execution are recorded by appending to the task's applicationLog (not shown).

Definition 4.4 (Uncommitted lineage of T)**.** The tasks in $Lineage(T)$ that are not yet committed in the global store.

4.2 Protocol

4.2.1 Forwarding Lineage

We describe the protocol for submitting a task from one process to another, first without flushing. For processes hosted by separate nodes, this requires a minimum of one message to send the task itself. We design the lineage stash protocol so that all additional information needed for recovery correctness, i.e. the task's lineage, can be computed locally by the sender and piggy-backed on this message.

As described in Section 3, only the nondeterministic events need to be forwarded to receiving nodes. If the application is

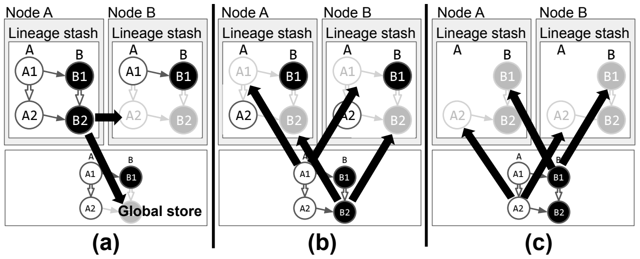


Figure 7. Forwarding and flushing lineage. **(a)** Task $A2$ submits task $B2$, forwards the uncommitted lineage ($A2$) to B, and asynchronously flushes $B2$. **(b)** A and B receive commit acknowledgements for $A1$ and $B2$. $A1$ can be evicted because it has no dependencies, but $B2$ cannot. **(c)** A and B receive commit acknowledgements for the remaining tasks and it is safe to evict all tasks.

```
def FlushTask(T):
  global_store.Write(T, TryEvict)
def TryEvict(stash, T):
  if T.version >= stash[T.id].version:
    for D in T.dependencies:
      if D in stash: return
    stash.erase(T)
```

Figure 8. Lineage stash methods for flushing to the global store. `FlushTask` writes a task asynchronously to the global store with the callback `TryEvict`. Once a task (or a newer version) is committed and its dependencies have been evicted, it is evicted in `TryEvict` (`TryEvict` also tries to evict any dependent tasks, not shown).

deterministic, i.e. the task specifications are immutable, then it is only necessary to *remember* tasks that have been submitted so far, by adding these tasks to the local stash (Fig. 6a).

For nondeterministic processes, the sender process retrieves the task's lineage from its local stash (Fig. 5a) and forwards the result along with the task (Fig. 6a). As in previous work [17], it is only necessary to forward *new* uncommitted lineage that has not yet been forwarded to the receiving process (not shown). Next, the receiver adds the forwarded lineage to its own stash (Fig. 5b) before assigning the task (Fig. 6b). If an added task is already present in the receiver's lineage stash, the more recent version is used (not shown). The task submission protocol is illustrated in Fig. 7a.

The `SubmitTask` and `AssignTask` procedures maintain the invariant that all lineage is *durable* during normal execution, assuming no flushing yet, proved by induction on the global state (all process and lineage stash state).

Invariant 1 (Lineage durability without flushing)**.** *For each process p and task T that p has executed or submitted, T's lineage is in p's local stash.*

4.2.2 Flushing the Stash

To prevent lineage stashes from growing indefinitely, each process flushes its stash to the shared global store. Because task versions are ordered, each lineage stash could safely flush any task that it sees. However, to avoid overloading the global store with many writes of the same task, we choose to instead flush a task every time it is updated, i.e., its `version` (Table 1) is incremented. When a task is submitted (Fig. 6a), the sender asynchronously flushes the initial version of the task, as A does for $B2$ in Fig. 7a. When the execution order is nondeterministic, the node updates the assigned task's specification to reflect its predecessor task and flushes again (`AssignTask` in Fig. 6b). If a task executes a nondeterministic event, the node adds the application-provided entry to the task `applicationLog` and flushes again (not shown).

Each node receives commit acknowledgements for particular task versions from the global store and evicts tasks from its stash accordingly (`TryEvict` in Fig. 8) . `TryEvict` only evicts a task if it has been committed and if its dependencies have also been evicted from the local stash. This is to guarantee that for every task still in the local stash, there is a connected subgraph in the stash that contains the task's uncommitted lineage, to ensure `GetUncommittedLineage` correctness when flushing is enabled.

As an example, in Fig. 7b, both processes receive acknowledgements for tasks $A1$ and $B2$. Note that these acknowledgements can arrive in any order since we do not assume sequential consistency from the global store. Both stashes can evict $A1$ because $A1$ does not have any dependencies. However, $B2$ cannot be evicted yet because it has uncommitted dependencies $A1$ and $B1$. This ensures that `GetUncommittedLineage` will correctly return the uncommitted lineage for $B2$ or any future tasks dependent on $B2$.

The `FlushTask` and `TryEvict` procedures maintain the same invariant as above, but for *uncommitted* lineage. The remaining lineage is in the global store and therefore durable.

Invariant 2 (Lineage durability with flushing)**.** *For each process p and task T that p has executed or submitted, T's* ***uncommitted*** *lineage is in p's local stash.*

4.2.3 Recovery Protocol

During the recovery protocol, the failed process retrieves and re-executes the lineage of the last task that it executed before failure that exists in another live process's lineage. Note that this may differ from the last task that the failed process actually executed but is enough to guarantee global consistency.

First, each process that submitted a task to the failed process resubmits its last submitted task. Step 5 in Figure 9b shows this for operator C from the application example in Fig. 3a. For a deterministic process, the lineage does not change after it is generated, so the resubmission step with Invariant 2 is enough to guarantee global lineage consistency (after re-execution). Invariant 2 implies that any uncommitted lineage will be forwarded with the resubmitted task. All other lineage can be retrieved from the global store.

When the execution is nondeterministic, the failed process must also retrieve the *latest* version of each task that it executed. We adopt a standard causal logging procedure [17],

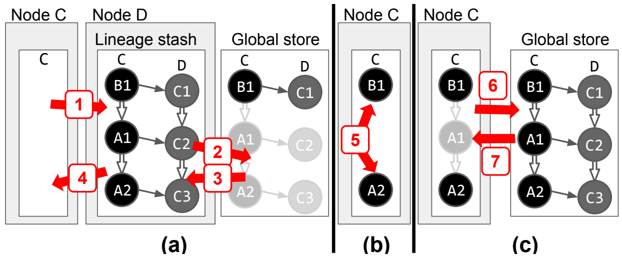


Figure 9. Recovery procedure for the nondeterministic process from Fig. 3 in detail. **(a)** (1) *C* contacts downstream process *D*, (2) *D* flushes its lineage, (3) *D* receives all acknowledgements, (4) *D* replies to *C*. **(b)** Processes *A* and *B* (not shown) resubmit their last submitted tasks (*A*2, *B*1) to *C*. This may happen concurrently with steps 1-4. **(c)** After steps 1–5, *C* recovers the lineage of *A*2 and *B*1, which includes the initial execution order, from the global store.

in which the failed process contacts other processes to retrieve their stashed lineage. However, rather than have the processes reply directly with the uncommitted lineage, each process instead flushes its entire local stash to the global store (via `FlushTask`, Fig. 8), waits for all tasks to commit (via `TryEvict`, Fig. 8), then acknowledges to the recovering process, which can then retrieve the flushed lineage from the global store (Fig. 9). Using Invariant 2 and the fact that the global store only accepts writes for higher task versions, this guarantees lineage consistency (after re-execution).

The recovering process can then re-execute its tasks based on the lineage retrieved from the global store, for both deterministic and nondeterministic applications. The process may resubmit tasks that have already executed, which get added to its stash as during normal execution to guarantee that Invariant 2 will hold after recovery completes. Receiving processes can easily deduplicate these tasks with a counter.

4.3 Failure Model

The protocols in Section 4.2 guarantee exactly-once semantics within an application. They can also support *end-to-end* exactly once semantics, e.g., if a sink operator in a stream processing application outputs to an external system. The lineage stash can support a task that outputs to the external world by first flushing the task and its local uncommitted lineage (Section 4.2.2), then waiting for the commit acknowledgements from the global store. This guarantees that if the operator fails later on, it will replay its execution in the same order and restore to a state consistent with the external world. In contrast, a global checkpointing approach alone must take a checkpoint during every such interaction to guarantee that the execution will not be rolled back [1, 19].

Like other rollback recovery systems, we target a fail-stop model [18]. Each node's local state (processes, in-memory object store, and lineage stash) can be rebuilt after a failure, possibly on a different physical node. As in previous causal logging work [4], we also allow the user to configure the maximum number of times that an uncommitted task is forwarded to lower-bound f, the number of simultaneous failures tolerated. f may be greater than the task forwarding limit depending on application properties: communication structure (e.g., in acyclic graphs [4]) and the mix of deterministic versus nondeterministic processes. We discuss further application-specific failure handling considerations here.

Checkpointing. Long-running applications must still take checkpoints to bound re-execution time after a failure. In theory, the application can take inconsistent checkpoints with the lineage stash. However, as prior work has shown [19], it is much simpler to take globally consistent checkpoints, to avoid coordination between processes for garbage collection of the global store. Since many applications today, e.g., distributed training [2] and stream processing [10], already provide support for efficient global checkpointing, we recommend adopting these methods to simplify and roughly bound recovery. The lineage stash can be used in conjunction to further guarantee exact replay, to reduce recovery time and runtime overheads for end-to-end exactly-once semantics.

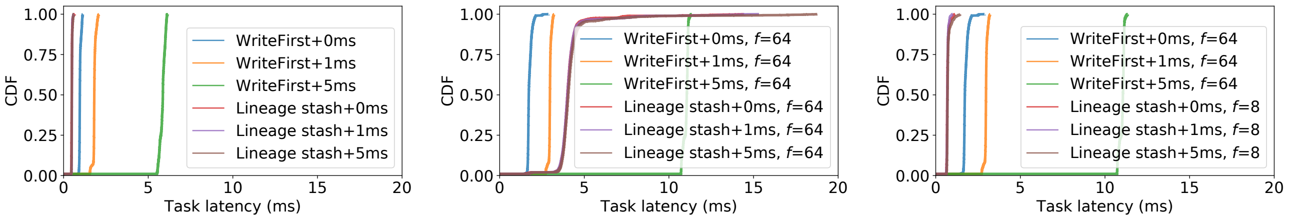
Intermediate State. In general, logging approaches collect state during execution in order to reduce recovery overheads and garbage-collect the state after a checkpoint. Therefore, in every logging approach, it is possible for this intermediate state to exceed storage capacity.

For the lineage stash, there are three types of intermediate state. First, for the lineage in the local stash, the node can apply backpressure on the local processes until enough tasks have been flushed, via the protocol in Section 4.2.2. Second, for the lineage in the global store, the options are to scale up the capacity (e.g., by adding shards), force an application checkpoint, or fall back to a global rollback in case of failure. Third, there are the objects in the local in-memory store. This is unique to the lineage stash because we decouple the object metadata (i.e., the lineage) from the object data. The options are similar: spill to external storage, force a checkpoint, or evict some objects and fall back to a global rollback.

5 Evaluation

We study the performance of the lineage stash compared to a `WriteFirst` method, which persists tasks to a global store before execution. We also evaluate the performance of the lineage stash on two end-to-end applications, distributed model training with ring allreduce and stream processing, and show that the lineage stash can provide faster recovery than a checkpoint-only solution with little to no additional runtime overhead. In summary, we study:

1. What is the latency overhead of the `WriteFirst` method compared to the lineage stash?
2. How can an application maintain a stable amount of uncommitted lineage?
3. How does the lineage stash benefit data processing applications vs a global checkpoint-only approach?



(a) Deterministic. **(b)** Nondeterministic, unlimited forwarding. **(c)** Nondeterministic, forward up to 8 nodes.

Figure 10. Task latency for deterministic and nondeterministic applications, with lineage stash vs WriteFirst. A ring of 64 processes is instantiated, one on each node. Each process submits no-op tasks with a unique token to its successor. Task latency is the time before the process receives its token again divided by the number of processes. For Fig. 10c, we forward an uncommitted task up to f=8 times.

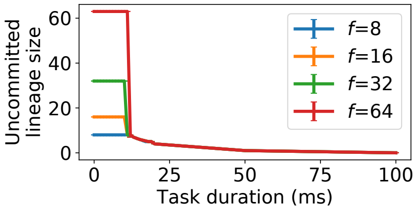


Figure 11. Median (and first and third quartiles) size of the forwarded uncommitted lineage, varying task duration for different values of f, the maximum number of concurrent failures tolerated. Above 10ms tasks, the uncommitted lineage size is stable.

We ran all experiments on Amazon EC2 (instance types inline). We implemented the lineage stash in 1k LoC (C++) on Ray [27], a low-latency system for distributed dynamic dataflows that normally uses WriteFirst. The recovery protocol for nondeterministic applications (Section 4.2.3) was implemented in an additional 125 LoC. For each Ray cluster, we used one non-replicated Redis instance per global store shard and one m5.8xlarge node separate from the workers to host the shards. In benchmarks that simulate global store write latency, we modified Ray to submit writes on a timer.

5.1 Microbenchmarks

Task latency distribution. We measure the latency of the lineage stash relative to WriteFirst. We also simulate global store latencies of +1 and +5ms. In Fig. 10, each process in a ring of 64 processes simultaneously submits a no-op task to its successor in the ring with a unique token, and we measure task latency based on the round-trip time of each token. Because of the ring structure, every task's lineage includes nearly every other task executed so far.

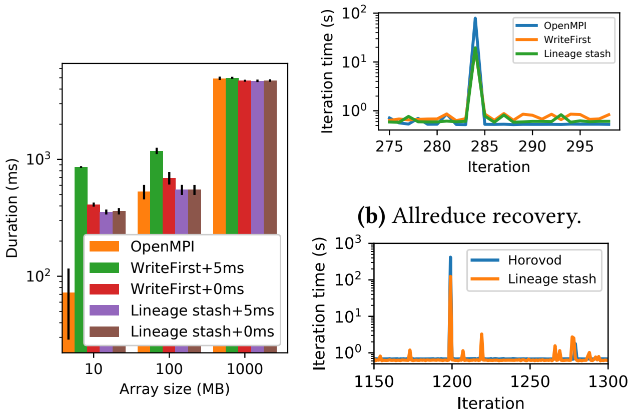
Figure 10a shows the latency distribution for applications with deterministic lineage. While WriteFirst can achieve p50=0.96ms and p99=1.12ms latency, it suffers greatly with simulated delays of +1 and +5ms (maximum of >6ms). Meanwhile, the lineage stash achieves p50=0.48ms and p99=0.58ms latency, even with +5ms of simulated delay. This is because each node only needs to remember uncommitted tasks that it submitted in its local stash (Section 3).

For nondeterministic applications, both systems flush each task a second time, before dispatch to the process, to record the execution order (Section 4.2). For the lineage stash, in Figs. 10b and 10c, we forward uncommitted tasks infinitely many and up to 8 times, to tolerate up to f=64 (the number of nodes) or f=8 simultaneous failures, respectively.

As expected, logging execution order doubles WriteFirst latency compared to deterministic applications (p50=1.72ms at +0ms; p50=11.07ms at +5ms). For the lineage stash with f=64 (Fig. 10b), the latency is much higher than for deterministic applications (p50=4ms and p99=11ms at +5ms). This is because every process has a path to every other process, which causes the uncommitted lineage to grow too large when the task duration is too short relative to the global store latency. Once we limit the number of times a task can be forwarded (Fig. 10c), latency is stable. The lineage stash's p50 latency at +5ms delay is 0.70ms, 15× lower than WriteFirst's at +5ms and lower even than WriteFirst's at +0ms.

Uncommitted lineage. The amount of forwarded uncommitted lineage depends on: (1) the global store latency, (2) the task arrival rate, (3) f, the number of simultaneous failures tolerated, and (4) the application structure (Section 4.3). For instance, if a process submits one task every T seconds to another process and the global store latency is $10T$, then we expect each task to forward an average of 10 tasks.

In Fig. 11, we vary task duration as a proxy for task arrival rate and report the forwarded lineage size, per submitted task. We also vary the maximum number of times an uncommitted task can be forwarded, to demonstrate how to cap the forwarded lineage at the cost of only tolerating f failures (Section 4.3). The workload is a ring of 64 nondeterministic processes as in Fig. 10, with a simulated global store latency of 100ms. This communication structure is challenging for the lineage stash because each process has a path to every other process, so each task must be forwarded to f other nodes to tolerate f failures. Also, the global store



(a) Allreduce latency.

(b) Allreduce recovery.

(c) SGD recovery.

Figure 12. (a) Allreduce duration on 64 workers (m5.2xlarge), averaged over 20 trials (with std. deviation). WriteFirst and the lineage stash use ring allreduce, with simulated global store latency as labeled. **(b)** Allreduce recovery time for lineage stash vs WriteFirst vs OpenMPI, with checkpoints to disk every 150 iterations. We kill and restart a worker at iteration 284. **(c)** Distributed SGD on the lineage stash vs Horovod v0.16.1, on 16 p3.8xlarge. Both use TensorFlow v1.12 on Resnet-101 with synthetic data and batch size 64. The lineage stash uses the same ring allreduce as in Section 5.2.1. Each worker checkpoints the model to disk every 640 iterations (~7min). We kill and restart a worker at iteration 1200.

	Allreduce		Distributed SGD		Streaming WC	
	OpenMPI	LS	Horovod	LS	Flink	LS
Mean latency w/o failure	530	550	684	674	79	92
Mean latency during failure	79,012	19,557	417,655	124,296	8,869	435

Table 2. Summary of mean latencies in milliseconds during normal operation and during recovery for Ray with the lineage stash (LS) compared to baseline systems on a variety of applications. For latency during a failure in streaming (Section 5.2.3), we take the mean of all reported latencies between the failure time to when the latency for new inputs converges to normal operation. For the other applications, we report the maximum latency.

should have much lower latency in practice, but we configure this to accurately show the effect of millisecond task durations.

Below task duration 11ms, the forwarded lineage in all cases grows unbounded and is capped only by f. This has consequences on the task latency: 61 forwarded tasks translates to 3.4ms latency, versus 1.1ms latency for 8.8 forwarded tasks. Interestingly, no matter the value of F, all configurations converge on 8-9 forwarded tasks at task duration 11ms. This suggests that for a given application structure and global store latency, there is a maximum task arrival rate under which the uncommitted lineage will remain stable.

5.2 End-to-end Applications

5.2.1 Ring allreduce

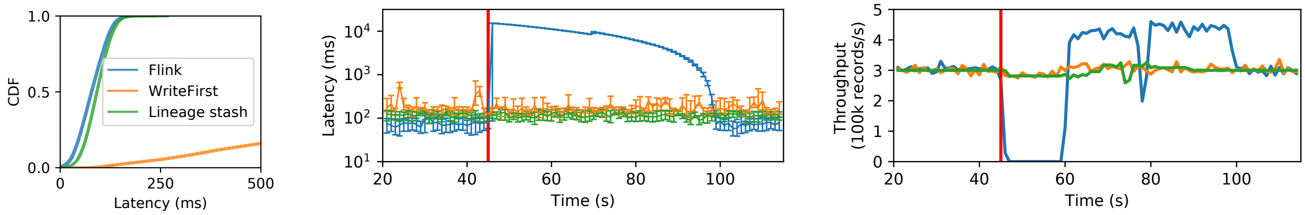
Allreduce is an important collective communication routine commonly used in high-performance computing in which all processes start with an input element and end with the reduced sum of the inputs. Ring allreduce is an implementation optimized for large arrays, in which a ring of P processes exchange inputs over $2(P-1)$ rounds of communication with P messages (tasks) each. The runtime of this algorithm is especially important for machine learning, where it is used in data-parallel synchronous distributed training to exchange gradients between copies of the model. Ring allreduce can be written as a deterministic application on the lineage stash. Also, because the application data is large, we cache all object data in Ray's per-node shared-memory store.

In Fig. 12a, we compare the runtime of ring allreduce on the lineage stash against the same implementation but with WriteFirst and against OpenMPI v1.10 [20]. We show that the latency with the lineage stash is comparable to that of OpenMPI and consistently lower than WriteFirst. On 100MB arrays, the mean duration on the lineage stash is 550ms versus 530ms on OpenMPI. The lineage stash outperforms OpenMPI on 1GB arrays but is 5× worse on 10MB, in both cases possibly because of OpenMPI's use of a different allreduce algorithm. Meanwhile, the lineage stash is 1.26× faster than the WriteFirst method on 100MB. With a global store delay of 5ms, the lineage stash iteration time stays constant, since it is insensitive to global store latency, while the WriteFirst iteration time increases to 1184ms.

We also compare recovery in Fig. 12b on an application that iteratively calls allreduce on a 100MB array on 64 workers. We checkpoint the allreduce data to disk every 150 iterations (~1min), kill and restart a node near iteration 280, and measure the time to recover all of the allreduce outputs since the last checkpoint. For OpenMPI, we restart the benchmark from the latest checkpoint on failure. For the lineage-based systems, the failed process retrieves all lost allreduce outputs since the last checkpoint from the remaining nodes' in-memory stores and replays the last allreduce iteration from the lineage. Figure 12b shows that the lineage stash (and WriteFirst) achieves 4× better recovery time than OpenMPI with only a small runtime overhead during normal operation (Table 2).

5.2.2 Distributed Training

Data-parallel distributed training is an increasingly important workload in which many copies of a model train on different batches of a dataset. In synchronous training, all workers iteratively compute a local gradient (in 100s of ms on GPUs), sum gradients with allreduce, and apply the summed gradient to their model copy. Thus, fast allreduce is critical for distributed training throughput. While distributed training is often long-running, meaning that fast recovery may



(a) Latency without failure.

(b) Latency during failure.

(c) Throughput during failure.

Figure 13. **(a)** Latency CDF for a streaming wordcount on 32 m5.xlarge workers at 400k records/s (4M words/s). Latency is sampled once every 1000 records. Both systems used a parallelism of 32 (per source, map, reduce, sink) and checkpoints to disk every 30s. **(b, c)** Failure and recovery for streaming wordcount on 32 m5.xlarge nodes at 300k records/s, checkpoints to disk every 30s. A worker is killed and restarted at t=~45s (vertical red line), ~15s after the first checkpoint. We report **(b)** median latencies seen by a single sink (with 1st and 3rd quartiles), x-axis is the record timestamp, and **(c)** total throughput, x-axis is physical time. The throughput drop at t=~80s is due to checkpointing.

be less important than in an online application, we show that the lineage stash can provide faster recovery than state-of-the-art systems with no perceivable runtime overhead.

In Fig. 12c, we compare distributed stochastic gradient descent (SGD) on Ray with the lineage stash vs Horovod v0.16 [32] (both with Tensorflow [2] v0.12) and show that we can achieve a similar mean iteration time of 674ms, compared to 684ms on Horovod, during normal operation (Table 2). Also, we show that we can recover from the failure at iteration 1200 in 124s, more than 3× faster than Horovod (417s). Approximately half of the lineage stash's recovery time is due to TensorFlow initialization, which could be reduced with a standby worker, while the rest is spent recovering and reapplying the lost gradients to the restored model.

5.2.3 Stream Processing

In this section, we measure the benefits of the lineage stash for an online stream processing workload. Stream processing at scale requires low-latency scheduling across many nodes. Since applications are also long-running, the chance of a failure is high, so reducing downtime during recovery is critical. Finally, these applications often interact with the external world, which in general cannot be rolled back, so exact replay is important for end-to-end exactly-once semantics.

We implement a streaming wordcount application on top of Ray with and without the lineage stash, with one long-running actor per mapper and reducer instance. Each actor batches records in the stream and submits one task per batch to a downstream actor. Mapper tasks compute over an input batch and contain only the lineage (no application data), while reducer tasks contain inlined task arguments. Reducers execute tasks *nondeterministically*, i.e. they process tasks from the mappers in order of arrival. This order is recorded in the lineage, as described in Section 4.1. To test the overhead of the lineage stash for nondeterministic processes, we record the latency for each reducer at a different node, so that the reducer must forward any uncommitted lineage to a remote node. We also implement asynchronous, globally consistent checkpointing, using the same algorithm as Flink [10].

Latency without failures. In Fig. 13a, we show that the lineage stash on Ray can achieve similar latencies as Flink (v1.8.1) at a throughput of 400k records/s (4M words/s) on 32 nodes. The p50 and p90 latency for Flink is 79ms and 125ms, respectively, vs. 92ms and 132ms for the lineage stash. Meanwhile, `WriteFirst` cannot keep up with the target throughput because the global store is a bottleneck.

Recovery time. In Figs. 13b and 13c, we run the same workload at 300k records/s and kill a worker ~15s after the first checkpoint. For both systems, we immediately restart the worker so that Flink has enough resources to continue.

For Flink, because the entire job must roll back and play forward again, new records are blocked by recovery and throughput drops to 0 (t=48-60s in Fig. 13c). Once all lost work has been replayed, at t=61s in Fig. 13c, the system can process new records that entered the stream during recovery. Because the system is overprovisioned for the target load, the system is able to use the extra capacity to eventually catch up to the input stream, returning to normal throughput at t=101s in Fig. 13c. Note that the higher the expected load during normal operation, the more the system must be overprovisioned for failure, or else the system will never catch up with the input stream after recovery. The records that are processed during this period (t=48-100s in Fig. 13b) all experience higher latency than normal (>15s) since their processing was blocked by the global rollback (Table 2).

For Ray with and without the lineage stash, the failed node has one source, mapper, reducer, and sink, each of which is replayed after the failure. The mapper can skip most tasks during replay since it is stateless, but the reducer must recompute its state from its last checkpoint. While new records scheduled to the recovering operators are delayed by task replay, those scheduled to intact operators can be safely processed. Thus, the total throughput drops only slightly after the failure, to ~280k records/s (t=48-65s in Fig. 13c). Once the failed operators have finished re-execution, they process the new records (t=66-80s in Fig. 13c). During this period, the total throughput increases (to ~320k records/s), as in Flink, but much less additional capacity is needed. Also, although the maximum per-record latency is about the same as for Flink, since the maximum work replayed by any single process is the same, most of the record latencies during recovery (t=48-80s in Fig. 13b) are actually the same as during normal operation, since they were not blocked by recovery (Table 2).

6 Related work

Message-passing systems. Because almost any distributed application could be logically viewed as a message-passing system [18], there are many framework examples, including parallel computing frameworks [20], distributed training frameworks [2, 32], low-latency data processing frameworks [9, 28], and actor frameworks [6, 8]. Out of these systems, the ones that provide explicit fault tolerance support [2, 9, 28, 32] use global checkpointing alone, most likely because this is the simplest to implement and understand and adds low and predictable runtime overhead. Previous work has studied techniques for asynchronous global checkpointing [12] that are optimize runtime overheads for particular applications, such as stream processing [9, 25]. However, in general, a global checkpoint-only approach introduces higher recovery overheads, as well as high runtime overhead when end-to-end exactly once semantics are needed [18], i.e., when outputting to the external world.

Causal logging [5, 17] is a general class of techniques in which processes log nondeterministic events asynchronously and piggyback volatile records onto messages to other processes. Potentially because of protocol complexity and difficulty in guaranteeing low runtime overhead in practice, causal logging is not used in any practical application that we are aware of. A primary difficulty in any logging approach, causal or otherwise, is that all possible sources of nondeterminism must be logged, which is complicated for a general application that can make system calls, share memory, etc. Our primary contribution is in identifying distributed data processing as a promising application for causal logging and describing how to efficiently capture the necessary nondeterministic events. We also present a system architecture for the stable log storage system that reflects the design of modern cloud storage systems, which are often highly available and horizontally scalable but guarantee only eventual consistency and do not promise low latency [11, 13, 16].

Lineage-based systems. MapReduce [15], Apache Hadoop [35], and Apache Spark [36] implement a *bulk synchronous parallel* model in which the user specifies data parallelism through a lineage graph of coarse-grained transformations that apply the same operation to each item in an arbitrarily sized dataset. A centralized scheduler then schedules tasks in each stage to execute over a data partition. For fault tolerance, the lineage is stored reliably at a centralized location, usually the scheduler, on the critical path of task execution. Drizzle [34] amortizes the scheduler overhead for applications where the lineage is known a priori, as in stream processing [37]. However, this does not solve the problems inherent to BSP systems, namely that the job must proceed in synchronous stages and each stage must be statically sized (e.g., the static microbatch size in Spark Streaming [37]).

CIEL [29], Ray [27], and Noria [21] are examples of lineage-based systems that support dynamic dataflows, but again with synchronous logging to a centralized location. More importantly, none of these systems support exact replay of nondeterministic execution. They target only computations that can be rolled back and replayed without side effects on the external world. Noria guarantees exactly-once semantics for client reads, but at the cost of rolling back and replaying all computation downstream of the failed node.

Transactional systems. Lineage, or *provenance*, is a powerful concept in database systems that can enable debugging and data auditing [14, 22]. The lineage stash could similarly enable debugging, especially for asynchronous distributed algorithms, but we focus here on fault tolerance. In this way, it is more closely related to the logging techniques used in databases to enforce transactional semantics, i.e. ACID.

The most common method is "write-ahead logging", where changes are durably logged before the transaction commits [26]. This technique is widely applicable and has been used to reduce recovery time and guarantee exactly-once semantics for large-scale stream processing in MillWheel [3], at the cost of higher latency during execution. MillWheel writes all operator state and intermediate records to a persistent storage system [13] on the critical path of execution, while the lineage stash logs only the lineage to persistent storage and does so off of the critical path.

7 Conclusion

We introduce the lineage stash, a causal logging technique for simultaneously achieving predictably low latency during normal execution and rapid recovery after a failure. While others [9, 18, 34] have shown that there is a fundamental tradeoff between these axes, we show here that the tradeoff need not affect the application. We achieve this by recording lineage off the critical path of task execution and replaying the lineage to reconstruct lost data after a failure. We evaluate the concept empirically on end-to-end applications in machine learning and stream processing, and show how the lineage stash enables large-scale, online data processing with fine-grained dynamic dataflows.

Acknowledgments

We thank Akshay Narayan, Malte Schwarzkopf, Michael Whittaker, Vasiliki Kalavri, and members of the RISELab at UC Berkeley for their helpful feedback. We also thank our shepherd, Lorenzo Alvisi, and the anonymous SOSP reviewers for their guidance. This research is supported in part by NSF CISE Expeditions Award CCF-1730628 and gifts from Alibaba, Amazon Web Services, Ant Financial, Arm, CapitalOne, Ericsson, Facebook, Google, Intel, Microsoft, Scotiabank, Splunk and VMware as well as by NSF grant DGE-1106400. John Liagouris was partially supported by a Swiss NSF "Scientific Exchanges" grant.

References

[1] An Overview of End-to-End Exactly-Once Processing in Apache Flink (with Apache Kafka, too!). https://flink.apache.org/features/2018/03/01/end-to-end-exactly-once-apache-flink.html.

[2] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., 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).

[3] Akidau, T., Balikov, A., Bekiroğlu, K., Chernyak, S., Haberman, J., Lax, R., McVeety, S., Mills, D., Nordstrom, P., and Whittle, S. Millwheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment 6*, 11 (2013), 1033–1044.

[4] Alvisi, L., and Marzullo, K. Trade-offs in implementing causal message logging protocols. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing* (1996), Citeseer, pp. 58–67.

[5] Alvisi, L., and Marzullo, K. Message logging: Pessimistic, optimistic, causal, and optimal. *IEEE Transactions on Software Engineering 24*, 2 (1998), 149–159.

[6] Armstrong, J., Virding, R., Wikström, C., and Williams, M. Concurrent programming in ERLANG.

[7] Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing 37*, 1 (1996), 55–69.

[8] Bykov, S., Geller, A., Kliot, G., Larus, J. R., Pandya, R., and Thelin, J. Orleans: Cloud computing for everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing* (2011), ACM, p. 16.

[9] Carbone, P., Ewen, S., Fóra, G., Haridi, S., Richter, S., and Tzoumas, K. State management in Apache Flink: Consistent stateful distributed stream processing. *Proc. VLDB Endow. 10*, 12 (Aug. 2017), 1718–1729.

[10] Carbone, P., Fóra, G., Ewen, S., Haridi, S., and Tzoumas, K. Lightweight asynchronous snapshots for distributed dataflows. *CoRR abs/1506.08603* (2015).

[11] Cattell, R. Scalable sql and nosql data stores. *Acm Sigmod Record 39*, 4 (2011), 12–27.

[12] Chandy, K. M., and Lamport, L. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst. 3*, 1 (Feb. 1985), 63–75.

[13] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS) 26*, 2 (2008), 4.

[14] Cheney, J., Chiticariu, L., Tan, W.-C., et al. Provenance in databases: Why, how, and where. *Foundations and Trends® in Databases 1*, 4 (2009), 379–474.

[15] Dean, J., and Ghemawat, S. MapReduce: Simplified data processing on large clusters. *Commun. ACM 51*, 1 (Jan. 2008), 107–113.

[16] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS operating systems review* (2007), vol. 41, ACM, pp. 205–220.

[17] Elnozahy, E. N. *Manetho: fault tolerance in distributed systems using rollback-recovery and process replication.* PhD thesis, Rice University, 1994.

[18] Elnozahy, E. N., Alvisi, L., Wang, Y., and Johnson, D. B. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv. 34*, 3 (2002), 375–408.

[19] Elnozahy, E. N., and Zwaenepoel, W. On the use and implementation of message logging. In *Proceedings of IEEE 24th International Symposium on Fault-Tolerant Computing* (1994), IEEE, pp. 298–307.

[20] Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R. H., Daniel, D. J., Graham, R. L., and Woodall, T. S. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting* (Budapest, Hungary, September 2004), pp. 97–104.

[21] Gjengset, J., Schwarzkopf, M., Behrens, J., Araújo, L. T., Ek, M., Kohler, E., Kaashoek, M. F., and Morris, R. Noria: dynamic, partially-stateful data-flow for high-performance web applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (Carlsbad, CA, Oct. 2018), USENIX Association, pp. 213–231.

[22] Gulzar, M. A., Interlandi, M., Yoo, S., Tetali, S. D., Condie, T., Millstein, T., and Kim, M. Bigdebug: Debugging primitives for interactive big data processing in spark. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)* (2016), IEEE, pp. 784–795.

[23] Hwang, J.-H., Balazinska, M., Rasin, A., Cetintemel, U., Stonebraker, M., and Zdonik, S. High-availability algorithms for distributed stream processing. In *21st International Conference on Data Engineering (ICDE'05)* (2005), IEEE, pp. 779–790.

[24] Koo, R., and Toueg, S. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on software Engineering*, 1 (1987), 23–31.

[25] Kwon, Y., Balazinska, M., and Greenberg, A. Fault-tolerant stream processing using a distributed, replicated file system. *Proc. VLDB Endow. 1*, 1 (Aug. 2008), 574–585.

[26] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS) 17*, 1 (1992), 94–162.

[27] Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Elibol, M., Yang, Z., Paul, W., Jordan, M. I., and Stoica, I. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (Carlsbad, CA, 2018), USENIX Association.

[28] Murray, D. G., McSherry, F., Isaacs, R., Isard, M., Barham, P., and Abadi, M. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 439–455.

[29] Murray, D. G., Schwarzkopf, M., Smowton, C., Smith, S., Madhavapeddy, A., and Hand, S. CIEL: A universal execution engine for distributed data-flow computing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2011), NSDI'11, USENIX Association, pp. 113–126.

[30] Rao, S., Alvisi, L., and Vin, H. M. The cost of recovery in message logging protocols. *IEEE Transactions on Knowledge and Data Engineering 12*, 2 (2000), 160–173.

[31] Schlichting, R. D., and Schneider, F. B. Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems (TOCS) 1*, 3 (1983), 222–238.

[32] Sergeev, A., and Del Balso, M. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799* (2018).

[33] Shabtay, L., and Segall, A. On the memory overhead of distributed snapshots. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 1994), PODC '94, ACM, pp. 401–.

[34] Venkataraman, S., Panda, A., Ousterhout, K., Ghodsi, A., Armbrust, M., Recht, B., Franklin, M., and Stoica, I. Drizzle: Fast and adaptable stream processing at scale. In *Proceedings of the Twenty-Sixth ACM Symposium on Operating Systems Principles* (2017), SOSP '17, ACM.

[35] White, T. *Hadoop: The Definitive Guide.* O'Reilly Media, Inc., 2012.

[36] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. 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* (2012), USENIX Association, pp. 2–2.

[37] Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., and Stoica, I. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 423–438.