CS294-170: Anna 2: Monotonic Boogaloo

Mingwei Samuel
mingwei@berkeley.edu
UC Berkeley
Berkeley, California, USA

Pranav Gaddamadugu
pranavsaig@berkeley.edu
UC Berkeley
Berkeley, California, USA

ABSTRACT

The ANNA key-value store proved that monotonicity and lattice composition are powerful design principles that can create a simple, understandable, and highly scalable distributed system. However, ANNA itself is not monotonic. Only the key-value state, at a high-level, is modelled as monotonic, while the rest of ANNA remains unreasoned about. Low-level implementation details may raise questions of Anna's monotonicity, while other whole components appear to be either clearly or possibly non-monotonic. Additionally, the monotonicity of data served by ANNA is not necessarily preserved once it leaves the system, so clients can easily break monotonicity.

In this project, we seek to reimplement ANNA in a way that is constructively monotonic. This process will help us find the limitations of purely monotonic design in terms of both expressivity and performance. Additionally, we hope to discover useful abstractions for representing distributed systems which can eventually be expanded into an easy-to-use cloud programming language.

1 INTRODUCTION

ANNA [11] is a partitioned, multi-master key-value store (KVS) which achieves high performance and elasticity via wait-free execution and supports several coordination-free consistency levels. ANNA's design is based on a simple architecture of shared-nothing actors which communicate through message-passing and each monotonically update internal state through the merging of composed-lattice-based data structures.

ANNA proved that *monotonicity* and *lattice composition* are incredibly powerful simplifying design concepts for distributed systems. However ANNA's monotonic design is limited to the key-value state stored within each actor. Monotonic guarantees depend on the correctness of ANNA's C++ implementation. Additionally, other parts of ANNA, such as key repartitioning, scale-down, logging, caching, and communication, appear to be either possibly or clearly non-monotonic.

This paper details our ongoing effort to reimplement ANNA in a way that is *constructively monotonic*. We hope to create a design in which we can easily and automatically verify correctness and monotonicity. Because of this, we chose to use the Rust programming language, which is as fast as C++ but also provides a powerful type system and strong safety guarantees, and is gaining popularity in both industry and academia [9]. We will use Rust's type system along with additional formal verification tools to verify and enforce the correctness of our implementation.

We are currently working in parallel on two separate reinplementations of ANNA. CRISPER, described in section 3.2, is a quick prototype built on Timely Dataflow [8] in which we get early performance results to guide our design in Rust. SPINACH, described in section 3.1, is a ground-up user library which provides low-level monotonic dataflow and lattice primitives. In SPINACH we hope to find abstractions that are both expressive enough to represent many distributed programs and modular enough to be automatically verified, and use this foundation to reimplement ANNA as well as other eventually consistent systems.

An additional goal of SPINACH is to guarantee monotonicity not just in the KVS backend but also in clients reading data, and in turn to any systems interacting with those clients, and so on. In this way, we hope SPINACH will be used across multiple machines, spreading monotonicity guarantees throughout a system.

In section 2 we review the theoretical background of monotonocity, lattices, and the CALM theorem, as well as related work. In section 3 we explain our work implementing CRISPER and the current abstractions in SPINACH. In section 4 we evaluate the performance of CRISPER and SPINACH and compare to the existing C++ ANNA implementation. And in section 5 we outline how we hope to automatically verify SPINACH programs and other future work.

1.1 Related Work

Existing high-level programming languages for distributed systems include BLOOM and LVARS. BLOOM is a high-level declarative programming language for distributed systems which can analyse a program's monotonicity and identify where coordination is needed [1]. BLOOM^L extends BLOOM with support for lattices-based datastructures [2]. LVARS is a monotonic-by-construction programming language based on lattices [6]. Our efforts are lower-level and aim to be more general purpose than these programming languages.

Much of the existing work in dataflow systems focuses on the execution details of running distributed dataflow programs. Our work does not focus on execution and we hope to not limit users to any specific dataflow execution model. Our work explores framing dataflow and reactive programming through monotonicity; we hope to constructively prove our dataflow programs as monotonic. Additionally, we hope to support dynamically changing dataflow graphs while still preserving these guarantees.

The connections we make between dataflow, reactive programming, and *state* are related to *differential dataflow* [7] and *partially-stateful* dataflow (used by Noria) [3]. The most recent implementations of both of these systems are written in Rust. These techniques are conventionally used to maintaining up-to-date materialized views of database queries.

2 THEORY

From the CALM theorem we know that monotonicity is the key to consistent, coordination-free distributed systems [4].

A function $f : S \longrightarrow T$ is *monotonic* if it preserves a partial ordering of the domain to a (possibly different) partial ordering of the codomain.

$$a \sqsubseteq_S b \quad \Longrightarrow \quad f(a) \sqsubseteq_T f(b) \qquad (\textit{monotonic})$$

The original stating of the CALM theorem uses subset ($\subseteq$) as the orderings of S and T in its definition of monotonicity; this definition is a generalization of that one. Intuitively, monotonicity means that new inputs never cause the function to retract previous conclusions, so following computations can speed ahead without need to backtrack or coordinate.

A *join semilattice* consists of a domain of possible states and a binary *join* operator, also known as the *least upper bound*. Dually, a *meet semilatice* consists of a domain and a binary *meet* operator or *greatest lower bound*. As these two structures are practically equivalent, we use *semilattice* to refer to both, and *merge* to refer to the binary operator.

A semilattice merge operator $\sqcup$ on a domain S must have the following three properties, for all $a, b, c \in S$:

$$a \sqcup (b \sqcup c) = (a \sqcup b) \sqcup c \qquad (\textit{associative})$$
$$a \sqcup b = b \sqcup a \qquad (\textit{commutative})$$
$$a \sqcup a = a \qquad (\textit{idempotent})$$

Together these are referred to as the *ACI properties*. These properties induce a partial ordering on S. Given $a, b \in S$, if the merge of a and b results in b then we can say "a comes before b" or "b dominates a":

$$a \sqsubseteq b \quad \equiv \quad a \sqcup b = b \qquad (\textit{semilattice partial order})$$

If the merge of a and b results in a new third value then a and b are incomparable.

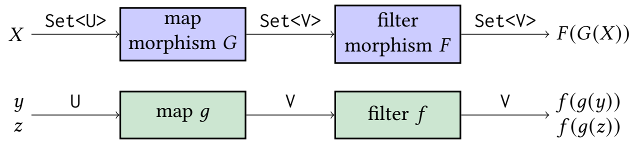

Figure 1: The bottom, in green, represents a simple dataflow program. The top, in blue, is a lifted view of the dataflow which can be thought of as operating on the complete set of inputs all at once.

A function $f : S \longrightarrow T$ from semilattice domain S to semilattice codomain T is a *morphism*[1] if it preserves merges, for all $a, b \in S$:

$$f(a \sqcup_S b) = f(a) \sqcup_T f(b) \qquad (\textit{morphism})$$

The semilattice order means morphisms are a special subset of monotonic functions which are *delta computable*. For example, if we have some state $z := f(a \sqcup b \sqcup c)$ and we want to extend this computation to a fourth value $d \in S$, we can compute $z' := z \sqcup f(d)$ which avoids recomputation on a, b, c. It turns out that this framing of computation has direct connections to the dataflow programming model.

2.1 Dataflow

Dataflow is a programming model where independent functions, called *operators*, are composed into a directed graph. Elements of data "flow" along the graph and are transformed by each operator. Operators such as `map`, `filter`, and `flatten` operate on single elements of data, while `fold` (also called `reduce` or `accumulate`) sits at the end of a chain of operators and combines all the arriving elements together into a single value.

It turns out this model of programming has a natural connection to lattice morphisms. Instead of thinking in terms of individual elements we can instead think of the *set* of all elements which are passed into an operator, across time. In this framing, each individual-element operator is a set-lattice morphism, and each individually arriving element is a delta.

In Figure 1 we see a simple example dataflow on the bottom in green, and a "lifted" view of the dataflow in terms of sets above in blue. Given individual inputs y and z the dataflow computes $f(g(y))$ and $f(g(z))$. If we view the output as a set this is equivalent to the lifted view of the computation, $F(G(y \cup z))$:

$$f(g(y)) \cup f(g(z)) \quad = \quad F(G(y \cup z))$$

This is a simple example, but we can view many dataflow programs in this way, as a composition of monotonic morphisms over set-union lattices. However, this can be extended

[1]Because both the domain and codomain are lattice spaces, *homomorphism* is a more precise term.

beyond just set-union lattices. At the output of the chain, we can `fold` the output elements together using any lattice merge function $\sqcup$ of our choosing, and the merge will be preserved over preceeding dataflow operations.

$$f(g(y)) \sqcup f(g(z)) \quad = \quad F(G(y \sqcup z))$$

Even more generally, the dataflow operations structurally preserve *any* final `fold` function (because the lifted view is directly derived from the per-element view). But when we limit the final `fold` to lattice merge functions we cause the preceding operators to become morphisms and guarantee that the dataflow is monotonic.

In the context of the CALM theorem, this provides a mathematical argument as to why the dataflow *model* has been so successful in distributed computing. The operators which combine to form a dataflow program are structure preserving and can be monotonic morphisms depending on how data is merged. If the data is merged in an order-preserving way, e.g. as lattices, that means the dataflow program can be coordination-free and highly scalable.

2.2 Reactive Programming

Reactive programming is a model where chains of computations are re-run when input values change. It is similar to the dataflow model in that data moves through a graph of operations. Unlike dataflow, computations are not run on independent elements (deltas) but instead computations are re-run on the same or slightly modified input values. Traditionally, reactive programming has been used to build UIs that are responsive to updates in data. However for our purposes it provides a model to handle non-morphism monotonic computations.

We refer to monotone functions which are *not* lattice morphisms as *monotone tricky*, abbreviated *MTT*. They are tricky because they do not satisfy the ACI properties of lattice and therefore require extra considerations to handle input reordering or duplication.

A simple example of an MTT function is set cardinality: given a set, return the number of elements in the set. This is clearly monotonic; the cardinality of the set grows as elements are added to the input. When we want to compute set cardinality on a the elements passing through a dataflow pipeline, each arriving element will probably increment the cardinality by one, however if an element is a duplicate of a previous element the cardinality remains unchanged; this function is not idempotent.

However, if we are given an entire set of elements then computing the cardinality is trivial. In a pipeline, this looks like *reactive programming*; each successive input is a slightly larger version of the same set. To transition from dataflow to reactive programming we use a `fold`, which requires internal state. Each arriving dataflow element causes the state to

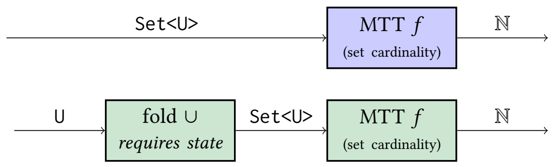

Figure 2: The pipeline before the fold (bottom left, in green) looks like dataflow. After the fold the pipeline looks like reactive programming. The fold does not have an effect on the lifted view (in blue), but after the fold the reactive pipeline and the lifted view match.

update, then a copy or reference to the state is passed along to the reactive pipeline. A representation of this is shown in figure 2.

Reactive pipelines are the key to ensuring monotonicity in dynamically extendable pipeline graphs. If we add operators to dataflow pipelines, then that operator misses seeing elements that have previously passed through the dataflow. This creates a clear race condition between elements in the dataflow and the addition of operators, and we cannot guarantee determinism or correctness. However, in a monotonic reactive pipeline each value is “larger” than all previous values, in a sense containing its whole past history. Therefore it is perfectly fine to add more operators onto a reactive pipeline. Because reactive pipelines depend on state, this means that dynamically extensible pipelines also require state, at least in this model.

Unlike dataflow, reactive pipelines require order guarantees. To be monotonic, each element passed through a reactive pipeline must be a monotonically larger version of the previous element, however if network delays reorder elements we must ensure that monotonicity is not violated. A communication channel which enforces order such as TCP is one way to prevent this. Another option is for the receiving end to store information about the largest lattice element seen so far and ignore any elements which are smaller. Interestingly, the later method maintains correctness even when messages are dropped, so it may be a useful abstraction for network communication.

We can relate this analysis back to the CALM theorem. Non-monotonic functions can be called *future-dependent* because any current conclusions could be invalidated by future inputs. Because of this, non-monotonic functions require *coordination* to control future inputs and avoid backtracking. Monotonic functions are *future-independent* and require no coordination. However monotone tricky functions depend on past inputs and can be called *past-dependent*, and require *state* in order to keep track of those past inputs. Finally, *morphisms* depend on neither the future nor the past and

Table 1: Time Dependence of Functions

Function	Depends on: Past	Depends on: Future	Note
Morphism	✗	✗	Requires neither state nor coordination.
Monotone tricky	✓	✗	Requires *state* to record the past.
Non-monotone	✓	✓	Requires *coordination* to control the future.

```rust
1  pub trait Merge {
2    type Domain;
3
4    fn merge_in(
5      target: &mut Self::Domain, delta: Self::Domain);
6
7    fn partial_cmp(a: &Self::Domain, b: &Self::Domain)
8      -> Option<Ordering>;
9  }
```

Figure 3: The merge trait. The main method, `merge_in`, takes a value and merges it in-place into `target`. Although the merge implicitly defines a partial order, a separate `partial_cmp` method is provided as an optimization to avoid unnecessary merging.

Table 2: Semilattice Merge Functions

Merge	Domain	Description
`Min`, `Max`	Totally-ordered domains, `Ord`.	Picks the smaller, larger element.
`Union`, `Intersect`	`HashSet<T>` or `BTreeSet<T>`.	Unions, intersects sets.
`MapUnion`[*]	`HashMap<K,V>` or `BTreeMap<K,V>`.	Unions keys, sub-merges intersecting values.
`Dominat-ingPair`[*†]	Pair tuples, `(X,Y)`	Picks the pair whose `X` dominates the other's. Otherwise sub-merges both `X` and `Y`.

[*]Higher-order merge functions require sub-merge(s) to be specified.
[†]`PairLattice` in the original Anna [11].

therefore do not require state nor coordination. Table 1 summarizes these dependencies. Overall more work is needed to formalize these ideas.

3 IMPLEMENTATION

In pursuit of a constructively monotonic design for Anna, we have developed Spinach and Crisper in Rust. The source code for which can be found below:

- https://github.com/MingweiSamuel/spinach
- https://github.com/d0cd/crisper

3.1 Spinach

Spinach is a Rust library which provides abstractions for composing semilattices and building monotonic dataflow and reactive pipelines. We aim for programs built using Spinach to be provably monotonic even when distributed across multiple machines. In this section we discuss the current abstractions used in Spinach and cover their benefits and limitations.

3.1.1 Semilattice Composition. Rather than directly composing semilattice objects, Spinach composes semilattices through the composition of semilattice *merge functions*. This avoids the overhead and awkwardness of wrapping instances in semilattice objects.

All merges implement the `Merge` trait, shown in figure 3. A merge specifies the domain on which it operates (line 2) and provides a binary merge operation (lines 4-5). Although mathematically a merge is symmetrical, as an optimization `merge_in` is asymmetrically; it takes in a value `delta` and merges it in-place into the referenced `target`. This avoids unnecessary copying of large semilattices.

Merges additionally provide a `partial_cmp` method (lines 7-8) which returns the partial ordering between two semilattice instances without modifying them. Technically the order can be fully determined by just the merge method; this is an optimization to avoid unnecessary cloning or merging of instances. In the future we may add more methods for optimization as needed, such as "subtraction" to compute the minimal delta needed to convert one semilattice instance to another when merging.

Spinach provides several semilattice merge types as listed in table 2. The provided merges are sufficient to implement a key-value store like Anna, and figure 4 gives an example of this. Additionally, users may provide custom merges by implementing the trait. Automatically verifying the correctness of both Spinach-provided and user-provided merges is future work, discussed further in section 5.

Merges are used exclusively in the `fold` operator, where state is accumulated and dataflow transitions to reactive programming. Limiting `fold`s to lattice merges ensures preceding operators can be modelled as lattice morphisms and are monotonic. We discuss operators in the next section.

```
1  MapUnion<HashMap<
2    String,
3    DominatingPair<Max<u32>, Max<String>>
4  >>
5
6  HashMap<String, ( u32, String )>
```

Figure 4: An example of a composed merge type for a "larger timestamp wins" hash map with string keys and integer-timestamped string values. Lines 1-4 show how semilattice merge functions are composed, while line 6 shows the resulting domain of the merge function.

```
pub trait Future {
  type Output;
  fn poll(&mut self, waker: Waker) -> Poll<Self::Output>;
}
pub enum Poll<T> {  Pending, Ready(T)  }
```

Figure 5: A simplified version of Rust's `Future` trait. Rust's `Futures` are not based on callbacks but instead on *polling*. An async runtime repeatedly polls each `Future` until it returns a final `Ready(T)` value. To avoid wasted work, the `Future` uses the provided `Waker` to notify the async runtime when it need to be re-polled.

```
pub trait Op {
  type Codomain;
  fn poll_next(&mut self, waker: Waker)
    -> Poll<Option<Self::Codomain>>;
}
```

Figure 6: A simplified version of Spinach's operator trait. `Op` is similar to Rust's `Future`, but can produce an unbounded number of values rather than just one. `Op` is effectively equivalent to the `Stream` asynchronous iterator trait used in many Rust libraries.

3.1.2 Operator Execution Model. Unlike Anna which uses an actor-based execution model, Spinach uses a more flexible model based around the program's pipeline graph. *Channel* operators explicitly delineate where a pipeline crosses threads, and special Rust traits mark which pipelines can be run in parallel on multiple threads. Using these tools the user can tightly control how a Spinach program is executed.

The implementation is based on Rust's async/await framework and `Future` trait, described briefly in figure 5. This provides us with user thread scheduling functionality and non-blocking execution. We currently use Tokio[2] as our async runtime and scheduler.

[2]https://github.com/tokio-rs/tokio

Dataflow operators are defined by implementing the `Op` trait (figure 6). An operator usually takes ownership of the previous operator and propagates poll calls down the chain. Rust *monomorphizes* these chains of operators, so it inlines them and optimizes them together.

When elements arrive at the beginning of a chain of operators, through some IPC or network channel, the async runtime is notified through the `Waker`. The runtime will schedule and soon poll the chain, *pulling* values through. Interestingly, this allows us to treat reactive pipelines in a special way. Dataflow pipelines return `Pending` responses when no new elements are available. However when a local *reactive* pipeline is polled it can always return a value because of the `fold` state. When the reactive state *actually* changes we notify the runtime via the `Waker` to poll succeeding operators, propagating the changes.

In general, the direction of control flow matches the direction of Rust ownership. In a pull-based pipelines, operators own *previous* operators and therefore, in terms of control flow, each operator can choose how and when to poll elements from *previous* operators. This is important for dynamic changes to the pipeline graph; it allows newly connected operators to immediately poll for the most recent value without having to wait for an update to arrive.

Previously we tried a push-based operator model where each operator owns and pushes to *following* operators, but this creates a control flow mismatch when new operators are connected—we would like to go *backwards* to get a value for the newly connected operator, however control flow goes *forwards*. We could add an alternative control or signalling mechanism to fix this, but that would create a far more complex abstraction.

3.1.3 Anna Implementation Plan. A KVS such as Anna implements two operations: a `PUT` for writing and a `GET` for reading.[3] Writing is straightforward: simply send a key and value into the dataflow, which will then handle and merge all writes together. However it is unclear how reading should operate in a dataflow context. Reads are dynamic; arbitrary keys can be read at arbitrary times, and reads are probing; they don't seem to follow the control flow of a dataflow graph. Additionally we want reads to preserve monotonicity, however a simple read "freezes" monotonic progress; a value copied out of the dataflow will no longer update.

We would like our "reads" to maintain monotonicity and continue updating. In fact, to implement some consistency Anna's client proxies need to wait for and re-query a key's value until its version reaches a certain threshold. To implement this, a read can be modelled as a *dynamic extension to the dataflow graph.* A new operator, or chain of operators, is

[3]`DELETE` is a special case of `PUT` which writes a gravestone value. We do not yet garbage collect these values.

added to read updates from the hash table, projecting them to the desired key, and sending them to the reader In this way a read operation is similar to the *subscribe* operation provided by pub-sub systems. A key difference between this and many pub-sub systems is that a read should immediately receive the current value rather than just waiting for the next value, to minimize latency.

The dynamic modification of the dataflow graph remains a tricky issue. Even if the dataflow execution is monotonic, if we are not careful modifications to the dataflow graph could be non-monotonic which would make the entire system monotonic. Additionally, clients may want to "unsubscribe" from updates at some point, we have to ensure that the removal of operators from the graph does not violate monotonicity as well. This remains a tricky problem requiring future work.

The reimplementation of ANNA using SPINACH remains a work-in-progress and we expect to encounter many more interesting issues in the future.

3.2 Crisper

CRISPER is an eventually consistent key-value store built on top of Timely Dataflow [8]. Moreover, CRISPER also uses the semilattice abstraction provided by SPINACH as its underlying representation of state. The system is intended to be an evaluation of dataflow as programming paradigm for distributed systems and is not intended for production. In this section, we discuss Timely Dataflow and the design for Crisper.

3.2.1 Timely Dataflow. Timely Dataflow is both a computational model and Rust library for building timely dataflows. The computational model is based on a graph in which stateful vertices send and receive logically timestamped messages along directed edges. Each worker or executor is given a global view of the dataflow graph and is responsible for executing every operator. Timestamps allows workers to track progress across epochs and and produce consistent output despite the presence of loops in the dataflow. Furthermore, Timely Dataflow provides an unopinionated view of coordination, allowing workers to impose a synchronous execution model, by tracking timestamps, but not requiring that they do so. In leaving flexibility to user, Timely Dataflow provides a versatile abstraction for building a variety of dataflow systems.

3.2.2 Design. CRISPER was designed after ANNA, in hopes of understanding how much of the system can be constructed as monotonic dataflow. That being said, there are components of the ANNA design that we omit in CRISPER. Specifically we do not consider key redistribution, multiple storage tiers, and autoscaling. Instead, CRISPER focuses on the core logic

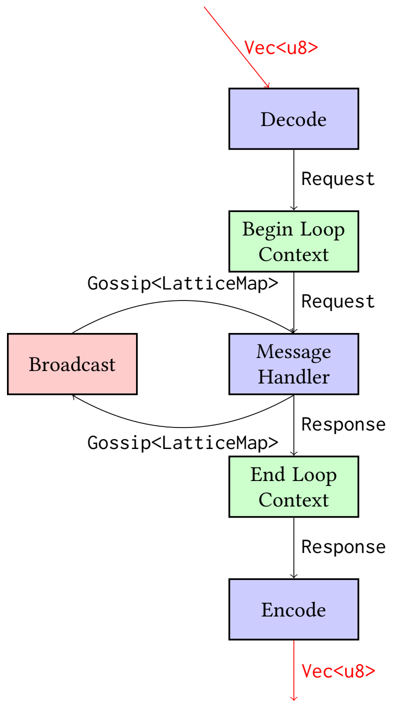

Figure 7: Overview of Crisper, described in Timely Dataflow.

of each worker thread, resulting in a system comparable to ANNA v0. The design of CRISPER is shown in Figure 7. The rectangular boxes represent explicit operators available in the Rust library for Timely Dataflow. Note that:

- The blue operators explicitly interact with the elements in the dataflow.
- The green operators denote loop contexts.
- The red operators explicitly communicate with the underlying workers.
- The black arrows indicate local movement of data.
- The red arrows indicate movement of data over the network.

The path of a request through the dataflow is as follows:

(1) The request arrives over the network as a sequence of bytes: `Vec<8>`.
(2) The `Decode` operator maps a `Vec<u8>` into a `Request` struct.
(3) The `Request` enters the loop context.
(4) The `Handle` operator contains a `LatticeMap<String, Lattice, MapUnionMerge>` lattice, which stores the data in the KVS. The `Request` either induces a read or is merged into the `LatticeMap` and produces a `Response`.
(5) The `Response` leaves the loop context.

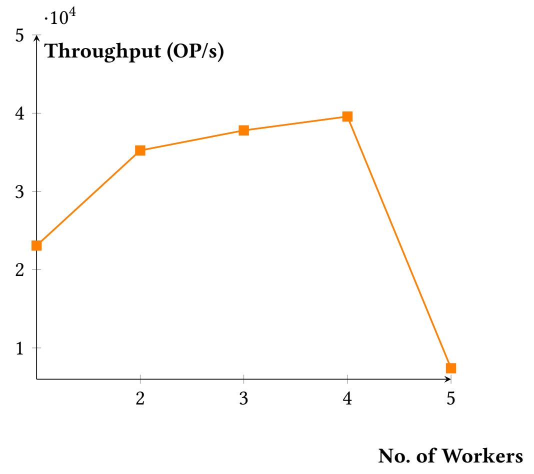

Figure 8: Throughput of PUT-heavy workload under a varying number of workers.

(6) The `Encode` operator maps a `Response` into a `Vec<u8>`.
(7) The sequence of bytes is sent over the network to the appropriate entity.

In order to guarantee eventual consistency in Crisper, lattice state is also sent to every worker in the dataflow via the `Broadcast` operator. When a `Gossip<LatticeMap>` is received from a `Broadcast` operator, it is merged into the worker's local lattice state. Note that, optimizations are made to ensure that redundant state is not sent between workers. In the 4 we will discuss the performance, limitations of, and lessons learned in building Crisper.

4 EVALUATION

In the section we present a selective evaluation of Crisper, aimed at identifying and understanding performance overheads. We then discuss some limitations of the Crisper design and Timely Dataflow and present a set of considerations for building a constructively monotonic systems.

4.1 Experimental Setup

We asses the behavior of Crisper under a PUT heavy workload, where 10 clients are instantiated, each attempting to PUT 100,000 unique keys-value pairs of 1KiB in size. We deploy Crisper on a 2017 MacbookPro, with 16GB RAM and 2.9 GHz Quad-Core Intel Core i7 processor. Crisper is evaluated with up to five workers and is deployed behind a local NGINX load balancer.

4.2 Results

The results of our experiment is shown in Figure 8. First, we make note that the drop in throughput when using 5 works is expected as our test machine only has 4 physical cores. Considering only the first four data points, we observe that although the throughput increases it does not appear to scale linearly. Note that the workers do not share state nor use any explicit synchronization mechanisms. Upon further investigation, it appears that the `Broadcast` operator, which requires serializing state and deserializing state, dominates the execution time. Furthermore, we also found that merging in state can take up to 1312ms in some cases! While we intend to instrument further analysis and more comprehensive benchmarks to fully understand the performance of Crisper, our initial findings leave us with some guidance for future work.

4.3 Lessons Learned

In this section we will describe the lessons we have learned in building Crisper. Broadly speaking, these ideas can be separated into lessons specific to Timely and those that apply to our larger goal of designing a framework for monotonic dataflow/reactive programming. We enumerate some of these ideas below.

- **L1.** Crisper uses the lattice abstractions provided by Spinach to realize a monotonic dataflow. To achieve safe usage, Spinach prevents users from `Clone`-ing lattices, however Timely requires that input to certain operators, including `Broadcast`, be clone-able. While arbitrary `Clone`-ing can result in monotonicity violations, it may be necessary in certain contexts. This seems to suggest that more work may be needed in designing a safe abstraction for `Clone`-ing lattices.
- **L2.** Crisper incurs a signification amount of overhead in serializing and deserializing its underlying state. This suggests that a fast method for serialization is needed for a performant implementation.
- **L3.** Crisper is implemented in a way that makes explicit use of as many features of Timely as possible. An artifact of this design is that gossiping state is placed on the main path of the dataflow, which results in significant overhead as the number of workers scale up. This indicates that optimizations could be made by moving gossip to the background via a dedicated runtime or some other mechanism.

5 VERIFICATION

In this section we will describe some of our initial efforts at leveraging a variety of verification techniques to enforce correctness. We will then transition to the discussion to future work in general.

```rust
use std::collections::HashSet;
use std::hash:Hash;

fn check_set_size_monotonicity<T: Eq + Hash>(
    a: HashSet<T>, b: HashSet<T>) -> ( usize, usize )
{
    ( a.len(), b.len() )
}
```

Figure 9: check_set_size_monotonicity. The function takes as input two HashSets and returns a tuple containing their respective sizes.

5.1 Problem Specification

Broadly speaking, the verification problem for a monotonic dataflow/reactive programming library can be broken down into the following questions.

- **Q1.** How do we prove monotonicity for primitive operations?
- **Q2.** How do we ensure that monotonic primitives are used correctly?

To illustrate **Q1.**, consider the code in Figure 9. While this is not an exact representation of the code we would like to verify, it serves as a useful proxy. We would like to verify that our implementation of set cardinality is indeed monotonic. To that end, one could check the conjuction of the following post-conditions on `check_set_size_monotonicity`.

$$\texttt{a.is\_subset(\&b)} \implies \texttt{result.0} \leq \texttt{result.1}$$

$$\texttt{b.is\_subset(\&a)} \implies \texttt{result.1} \leq \texttt{result.0}$$

Given that we have a method to verify primitive operations, it then follows to ask **Q2.**. In answering both of these verification techniques, we will have a method for building *verifiable*, constructively monotonic programs.

5.2 Initial Attempts

Our initial efforts at incorporating verification techniques are primarily centered around Prusti [10]. Prusti is a static verification tool for the *safe* subset of Rust that enables function verification, via preconditions, postconditions, and loop invariants. In using the tool, we quickly found out that many of our hand-written specifications for monotonicity would invoke unsafe Rust or use Rust features that are incompatible with Prusti. Among the other approaches we have considered, RustBelt [5], which leverages theorem provers to verify safe encapsulation of "unsafe" code.

6 FUTURE WORK

6.1 Spinach

In it's current form, Spinach will soon be enough to implement a rudimentary distributed KVS similar to Crisper, and we hope to soon benchmark this implementation against Anna and Crisper. The main missing feature is network support for communication between nodes, as well as the associated serialization and deserialization operations.

The operator abstractions used by Spinach are currently very rudimentary and we will need to add more types or traits to differentiate between dataflow and reactive pipelines. Finally, it is unclear how we will reason about the correctness of dynamic extension to the pipeline execution graph. We will undoubtedly run into problems such as these which will allow us to further refine and improve the abstractions provided by Spinach.

6.2 Crisper

Crisper was built to inform us about the requirements for a performant monotonic dataflow library. While the system has been built, the evaluation has yet to be completed. To that end, we plan on (2) developing representative benchmarks beyond PUT heavy workloads, (2) deploying Crisper in a distributed setting, and (3) rigorous profiling to identify further performance overheads.

6.3 Verification

There remains many unanswered questions regarding our verification approach. We see the most promise in using an interactive theorem prover like Coq as there are Rust projects in this domain [5]. There are however some disadvantages in using such tools, the most significant of which is a lack of domain expertise in theorem proving.

7 CONCLUSION

In this paper we propose an approach for implementing Anna, an eventually consistent key-value store in a constructively monotonic manner. To that end, we put discuss some ideas for constructing monotonic dataflow and reactive programs, using lattices as the underlying representation. We then present Spinach, a Rust library that provides abstractions for composing semilattices and building monotonic dataflow and reactive pipelines, and Crisper, an implementation of the core logic in Anna over Timely Dataflow. From Crisper, we have learned a few lessons that will hopefully guide further work on Spinach. We plan to continue to refine our ideas and extend the capabilities of Spinach.

ACKNOWLEDGMENTS

To Joseph M. Hellerstein and Matthew Milano for their guidance with this project and their hard work leading the CS294-170 *Programming the Cloud* class. And to our advisors, Alvin Cheung and Sanjit Seshia.

REFERENCES

[1] Peter Alvaro, Neil Conway, Joe Hellerstein, and William Marczak. 2011. Consistency Analysis in Bloom: a CALM and Collected Approach. *CIDR 2011 - 5th Biennial Conference on Innovative Data Systems Research, Conference Proceedings*, 249–260.

[2] Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. 2012. Logic and Lattices for Distributed Programming. In *Proceedings of the Third ACM Symposium on Cloud Computing* (San Jose, California) *(SoCC '12)*. Association for Computing Machinery, New York, NY, USA, Article 1, 14 pages. https://doi.org/10.1145/2391229.2391230

[3] Jon Gjengset, Malte Schwarzkopf, Jonathan Behrens, Lara Timbó Araújo, Martin Ek, Eddie Kohler, M. Frans Kaashoek, and Robert Morris. 2018. Noria: dynamic, partially-stateful data-flow for high-performance web applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 213–231. https://www.usenix.org/conference/osdi18/presentation/gjengset

[4] Joseph M. Hellerstein and Peter Alvaro. 2020. Keeping CALM. *Commun. ACM* 63, 9 (Aug. 2020), 72–81. https://doi.org/10.1145/3369736

[5] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (Dec. 2017), 34 pages. https://doi.org/10.1145/3158154

[6] Lindsey Kuper and Ryan R. Newton. 2013. LVars: Lattice-Based Data Structures for Deterministic Parallelism. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Functional High-Performance Computing* (Boston, Massachusetts, USA) *(FHPC '13)*. Association for Computing Machinery, New York, NY, USA, 71–84. https://doi.org/10.1145/2502323.2502326

[7] Frank McSherry, Derek Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential dataflow. In *Proceedings of CIDR 2013* (proceedings of cidr 2013 ed.).

[8] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: A Timely Dataflow System. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 439–455. https://doi.org/10.1145/2517349.2522738

[9] Jeffrey M. Perkel. 2020. Why scientists are turning to Rust. *Nature* 588, 7836 (Dec. 2020), 185–186. https://doi.org/10.1038/d41586-020-03382-2

[10] Alexander J. Summers. 2020. Prusti: Deductive Verification for Rust (Keynote). In *Proceedings of the 22nd ACM SIGPLAN International Workshop on Formal Techniques for Java-Like Programs* (Virtual, USA) *(FTfJP 2020)*. Association for Computing Machinery, New York, NY, USA, 1. https://doi.org/10.1145/3427761.3432348

[11] C. Wu, J. Faleiro, Y. Lin, and J. Hellerstein. 2018. Anna: A KVS for Any Scale. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 401–412. https://doi.org/10.1109/ICDE.2018.00044