Edelweiss: Automatic Storage Reclamation for Distributed Programming

Neil Conway, Peter Alvaro, Emily Andrews, Joseph M. Hellerstein
UC Berkeley
{nr, palvaro, e.andrews, hellerstein}@cs.berkeley.edu

ABSTRACT

Event Log Exchange (ELE) is a common programming pattern based on immutable state and messaging. ELE sidesteps traditional challenges in distributed consistency, at the expense of introducing new challenges in designing space reclamation protocols to avoid consuming unbounded storage.

We introduce Edelweiss, a sublanguage of Bloom that provides an ELE programming model, yet automatically reclaims space without programmer assistance. We describe techniques to analyze Edelweiss programs and automatically generate application-specific distributed space reclamation logic. We show how Edelweiss can be used to elegantly implement a variety of communication and distributed storage protocols; the storage reclamation code generated by Edelweiss effectively garbage-collects state and often matches hand-written protocols from the literature.

1. INTRODUCTION

“Blossom of snow may you bloom and grow,
bloom and grow forever.”
—Oscar Hammerstein, “Edelweiss”

Distributed and parallel systems are increasingly commonplace, but writing reliable programs for these environments remains stubbornly difficult. Both developers and academics have identified shared mutable state as a common source of problems: code that mutates shared state is hard to reason about and often requires expensive, fine-grained synchronization and careful coordination between processes to achieve consistent state and correct behavior.

To avoid these problems, both practitioners and academics have proposed writing systems by using immutable state whenever possible [16, 20, 29]. Rather than directly modifying shared state, processes instead accumulate and exchange immutable logs of messages or events, a model we call Event Log Exchange (ELE). Previously learned information is never replaced or deleted, but can simply be masked by recording new facts that indicate that the previous information is no longer useful.

By using event logs, ELE designs achieve a variety of familiar benefits from database research. For example, rather than determining a conservative order for modifications to shared state, ELE can allow operations to be applied in different orders at different replicas and reconciled later, reducing the need for coordination and increasing concurrency and availability. ELE allows simple mechanisms for fault tolerance and recovery via log replay, and provides a natural basis for system debugging and failure analysis.

ELE is an attractive approach to simplifying distributed programming, but it introduces its own complexities. If each process accumulates knowledge over time, the required storage will grow without bound. To avoid this, ELE designs typically include a background “garbage collection” or “checkpointing” protocol that reclaims information that is no longer useful. This pattern of logging and background reclamation is widespread in the distributed storage and data management literature, having been applied to many core techniques including reliable broadcast and update propagation [13, 14, 19, 22, 28, 34, 38, 45], group communication [17], key-value storage [3, 15, 45], distributed file systems [18, 26, 35], causal consistency [7, 23, 27], quorum consensus [21], transaction management [2, 11], and multi-version concurrency control [36, 39, 44].

Despite the similarity of these example systems, each design typically includes a storage reclamation scheme that has been developed from scratch and implemented by hand. Such schemes can be subtle and hard to get right: reclaiming garbage too eagerly is unsafe (because live data is incorrectly discarded), whereas an overly conservative scheme can result in hard-to-find resource leaks. Moreover, the conditions under which stored values can be reclaimed depends on program semantics; hence, a hand-crafted garbage collection procedure must be updated as the program is evolved, making software maintenance more difficult.

It would seem that ELE simply trades the difficulties of consistency for a different set of difficulties in space reclamation. In this paper, we make ELE significantly more attractive by removing the burden of space reclamation from the programmer. We present a collection of program analyses that allow background storage reclamation to be automatically generated from program source code.

We introduce Edelweiss, a sublanguage of Bloom [4, 10] that omits primitives for mutating or deleting data. Instead, Edelweiss programs describe how local knowledge contributes to the distributed computation. The system computes the complementary garbage collection logic: that is, it automatically and safely discards data that will never be useful in the future.

We validate our work by demonstrating a wide variety of communication and storage protocols implemented as Edelweiss programs with efficient, automatically generated reclamation. Our demonstrations in the paper include reliable unicast, reliable broadcast, a replicated key-value store, causal consistency, and atomic read/write.
2. BLOOM AND EDELWEISS

We begin by reviewing Bloom, a declarative language for distributed programming [4, 10]. We then present Edelweiss, a sublanguage of Bloom that disallows data mutation and deletion.

A Bloom cluster consists of a set of nodes. Each node has collections and rules. The collections define node state; nodes perform computation and communication by evaluating the rules over their local collections. There is no shared state; rather, nodes communicate via message passing. Each node executes a simple event handling loop: first, it accepts inbound messages and adds them to its local collections. Then the program’s rules are evaluated over those collections. This produces new values; some of those values denote outbound messages (as described below), which are sent. Finally, the node returns to listening for new inbound messages. An iteration of this loop is called a timestep. We assume every node has the same set of collections and rules.

The initial implementation of Bloom, called Bud, allows Bloom logic to be embedded inside Ruby programs. Hence, Bloom collections and rules are defined inside a stylized Ruby class. At each node, a small amount of Ruby code is used to instantiate the Bloom program and cause it to begin executing; more details are available on the Bloom website [10].

2.1 Data Model

A collection is a set of facts (tuples), akin to a relation in Datalog. Each collection has a schema, which defines the structure of the facts in the collection. The schema declares a set of column names; a subset of those columns forms the collection’s key. As in the relational model, the keys functionally determine the rest of the columns. That is, at any given node, a collection will not contain two different facts with the same values for their key columns. In line 6 of Figure 1, id is the key column, which functionally determines addr and val. The schema method (line 7) allows the schema of one collection to be reused for another, reducing redundancy.

Bloom provides several collection types with different semantics (Table 2). A table is a persistent collection: once a fact appears in a table, it remains until it is explicitly deleted. A scratch collection is akin to a view in SQL: semantically, scratches are recomputed at the start of every timestep.\(^1\) Hence, a fact remains in a scratch only as long as it can be derived from the persistent collections. Lastly, channel collections allow asynchronous communication. When a fact is inserted into a channel, the fact is delivered to a remote node at a non-deterministic future time. The network address to which the

\(^1\)The runtime may choose to materialize scratch collections to avoid the cost of repeated recomputation.

<table>
<thead>
<tr>
<th>Name</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>table</td>
<td>Persistent storage.</td>
</tr>
<tr>
<td>scratch</td>
<td>Transient storage.</td>
</tr>
<tr>
<td>channel</td>
<td>Asynchronous communication. A fact inserted into a channel is delivered to a remote Bloom node at a non-deterministic future time.</td>
</tr>
</tbody>
</table>

Table 2: Bloom collection types.

fact is sent is given by the channel’s location specifier column, which is prefixed with “@” in the schema. For example, line 5 in Figure 1 means that each fact in chn will be sent to the network address given by the fact’s second column (e.g., “example.org:1234”).

2.2 Rules

A rule has one or more input collections and a single output collection; the rule defines how the input collections are transformed before being included (via set union) in the output collection. A rule has the form:

<collection-identifier> <op> <collection-expression>

The left-hand side (lhs) is the name of the output collection and the right-hand side (rhs) is an expression that produces a collection. The rhs expression can include the usual relational operators (e.g., selection, projection, join, grouping, and negation), although Bloom adopts a syntax intended to be familiar to imperative programmers.

Bloom provides several operators that determine when the rhs will be included in the lhs (Table 3). The <= operator performs standard logical deduction: that is, the lhs and rhs are true at the same timestep. The <+ and <~ operators indicate that facts will be added to or removed from the lhs collection at the beginning of the next timestep. The <-> operator specifies that the rhs will be merged into the lhs collection at some non-deterministic future time. The lhs of a rule that uses <-> must be a channel. The <-> operator captures unreliable, asynchronous communication: channel messages might be delayed, reordered, or dropped.

2.2.1 Monotonicity and Persistence

An important property of a Bloom rule is whether it is monotonic: as new facts are added to the rule’s input collections, does the output collection strictly grow? As in Datalog, projection, selection, and join are monotonic, while aggregation and negation are not. As we will explain shortly, the behavior of negation is particularly important in Edelweiss. Given X, notin(Y) (the set difference of X with Y), observe that notin is monotone with respect to X but anti-monotone with respect to Y: that is, when new Y tuples appear, the output of the notin operator shrinks. We call X and Y the positive and negative inputs to the notin, respectively.

A persistent collection strictly grows over time, including from one timestep to the next. A table is persistent unless it appears on the lhs of a deletion rule (<-). A scratch is persistent if it is defined via monotone rules over persistent collections.

2.3 Edelweiss

Edelweiss is a sublanguage of Bloom that imposes the following restrictions on programs:

1. Deletion rules cannot be used (<- operator).
2. Channel messages are stored persistently. That is, the lhs of a rule that reads messages from a channel must be persistent.
3. RELIABLE UNICAST

As described in Section 2, channels provide asynchronous messaging. Although asynchronous communication matches the capabilities of the physical network, many applications find it convenient to use reliable unicast, in which a sender repeatedly transmits a message until it has been acknowledged by the recipient.

Figure 1 shows a naive reliable unicast program. Each message contains a unique ID, destination address, and payload. The `sbuf` collection is the sender-side buffer; communication is expressed by copying `sbuf` into the `chn` channel (line 11); the recipient persists delivered messages in `rbuf` (line 12). Note that because `sbuf` is persistent (as declared on line 6), Bloom’s semantics [5] dictate that a new `chn` message will be sent for every timestep at the sender.

Although it is concise and declarative, the naive reliable unicast program has two obvious shortcomings. First, an unbounded number of `chn` messages are derived. Although inefficient, this is not incorrect; Bloom collections have set semantics and the receiver-side logic is idempotent, which means that delivering the same message more than once has no effect. Second, the sender-side buffer `sbuf` grows without bound. This is unnecessary in practice: once a message has been successfully delivered to the recipient, it need not be retained by the sender.

We could address both problems by making the program more complex—for example, by arranging for receivers to emit acks and for senders to delete acknowledged messages. However, these modifications would not change the user-visible behavior of the program!

Acknowledgment and storage reclamation are not necessary for correctness—rather, they are only needed to help ensure that resources are used efficiently. We would like the best of both worlds: a concise, declarative program that has an efficient implementation. In the remainder of this section, we introduce a series of techniques that achieve this goal by allowing acks and storage reclamation to be introduced by safe, automatic program transformations from Edelweiss to an equivalent Bloom program.

3.1 Avoidance of Redundant Messages (ARM)

We begin by detailing ARM, an automatic program rewrite that avoids redundant communication between nodes. This requires identifying when delivering a message multiple times is redundant, and then rewriting the program to avoid duplicate transmissions.

In Edelweiss, detecting when duplicate channel deliveries are redundant is simple: the restrictions in Section 2.3 imply that once any message has been delivered, the receiver will persist it and subsequent attempts to send that message can safely be suppressed. Therefore, we can rewrite every Edelweiss rule that inserts messages into a channel to avoid inserting duplicates. For the program in Figure 1, avoiding duplicates would be easy if the sender could directly access the receiver’s buffer:

```plaintext
chn <= sbuf.notin(rbuf)
```

This approach is not possible because Edelweiss nodes can only communicate via message passing. However, a simple variant is possible: receivers can inform senders about messages that have

<table>
<thead>
<tr>
<th>Technique</th>
<th>Goal</th>
<th>Requirements</th>
<th>Mechanism</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avoidance of Redundant Messages (ARM)</td>
<td>Avoid sending duplicate messages</td>
<td>Receiver logic ignores duplicates</td>
<td>Add logic to send acks; avoid sending ack’ed messages</td>
<td></td>
</tr>
<tr>
<td>Positive Difference Reclamation (DR+)</td>
<td>Reclaim storage for X in X:notin(Y)</td>
<td>X, Y are persistent; logic downstream of X is reclaim-safe</td>
<td>Reclaim from X upon match in Y</td>
<td></td>
</tr>
<tr>
<td>Negative Difference Reclamation (DR−)</td>
<td>Reclaim storage for X and Y in X:notin(Y)</td>
<td>X, Y are persistent; logic downstream of X and Y are reclaim-safe; notin quals cover X’s keys</td>
<td>Create range collection for X’s keys; reclaim from X and Y upon match</td>
<td></td>
</tr>
<tr>
<td>Range Compression</td>
<td>Efficient storage of gap-free sequences</td>
<td>Column values contain one or more gap-free sequences; no non-key columns</td>
<td>range collection type</td>
<td></td>
</tr>
<tr>
<td>Punctuations [43]</td>
<td>Bounded storage for join input collections</td>
<td>Join appears as input to notin; punctuation matches join predicate</td>
<td>sealed collection type, supplied by user, or inferred from rule semantics</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Summary of mechanisms and analysis techniques in this paper.

Figure 1: Naive reliable unicast in Edelweiss.
been successfully delivered. Because such communication is asynchronous, the sender will only have a lower bound on the receiver’s state—but since the receiver ignores duplicate messages anyway, this does not harm correctness.

There are many ways in which senders can learn a conservative estimate of the receiver’s state, such as cumulative, timer-based acks (as in TCP) or “piggybacking” acks onto normal message traffic. For programs involving multiple senders and receivers, even more strategies are possible, such as epidemic gossip [13] or tree-based multicast. Any of these schemes could be used by ARM, since they all accomplish the same purpose of allowing senders to lower-bound the receiver’s state. For unicasting delivery, a simple scheme suffices: a receiver sends an ack whenever they receive a message.

The result of applying ARM to the naive unicast program is shown in Figure 2. ARM automatically introduces a new channel (line 9), which is used to send acks upon successful receipt of a sbuf message (line 15). Senders persist acks (line 16). Finally, ARM rewrites the rule that sends sbuf messages to avoid sending acknowledged messages (line 13). Note that ARM automatically infers that acks only need to contain message IDs (line 15), not the entire message. This is possible because id is the key of sbuf (line 5), which means that a given ID is associated with exactly one message.

3.2 Positive Difference Reclamation (DR+)

The ARM rewrite allows the simple unicast program in Figure 1 to avoid sending an unbounded number of messages, but the rewritten program in Figure 2 still does not reclaim acknowledged messages from sbuf. In fact, the program’s storage consumption has grown because the sender also persists the chn_approx collection. In this section, we introduce DR+, a program rewrite that automatically and safely reclaim storage, and show how it can be applied to sbuf; we address chn_approx in the following section.

DR+ exploits the semantics of set difference, which is expressed in Bloom using the notin operator. Consider X.noin(Y), where X and Y are persistent. Recall that X and Y are the “positive” and “negative” inputs to the notin, respectively; as new tuples arrive in Y, any matching tuples in X will no longer appear in the output of the notin. Moreover, because Y is persistent, any X tuple that has a match in Y will never appear in the output of the notin again.

For the purposes of this rule, X tuples with matches in Y will never contribute to program outcomes and no longer need to be stored.

To reclaim from X, Edelweiss needs to prove that doing so will not change the output of any other rule that references X. In many cases, this is easy to do: for example, if X appears on the rhs of a projection or selection rule with a persistent collection on its lhs, we can reclaim from X (intuitively, the rule makes a persistent “copy” of the X tuple). When X appears in more than one set difference rule, we can reclaim X tuples when the conjunction of the reclamation conditions of the rules are satisfied; we give an example in Section 5.1. Finally, reclaiming from collections that appear in joins is more complicated; we discuss this scenario in Section 6.2.

Returning to the reliable unicast example, we observe that sbuf is only referenced by a single rule, where it is used as the positive input to a notin operator (line 13 in Figure 2). Because chn_approx is persistent, DR+ will reclaim sbuf tuples that have matches in chn_approx. This is done by adding this rule to the program:

```
  sbuf <-(sbuf * chn_approx).lefts(id => id)
```

The rhs computes the equijoin of sbuf and chn_approx on id and returns the left join input (tuples from sbuf); the < operator removes the resulting sbuf tuples. This rule corresponds to our intuition that once the recipient has acknowledged successful delivery of a message, the message can safely be discarded by the sender.

Note that ARM and DR+ are independent program rewrites, but they work together profitably: ARM introduces set difference operations and DR+ exploits the semantics of set difference to safely and automatically reclaim storage.

3.3 Range Compression

Lastly, we need to address the storage used by the chn_approx collection. Unfortunately, DR+ is not useful because chn_approx does not appear as the positive input to a notin operator. Moreover, reclaiming tuples from chn_approx is problematic in principle: if we deleted such tuples, we would have no information at the sender to prevent redelivering acknowledged sbuf messages in the future.

Rather than reclaiming from chn_approx, can we instead represent the entire collection using a small amount of storage? Fortunately, this is feasible: recall that chn_approx only contains a single column, the message ID. Since IDs are assigned by a single sender, the sender can choose IDs from a gap-free, totally ordered sequence such as the natural numbers starting at some constant k. Because we expect all messages to eventually be delivered, chn_approx will eventually contain all the IDs from k to n. Hence, our task is much easier: we need to represent [k, . . . , n], which we can do by storing the smallest and largest elements of the set.

However, some elements of the set [k, . . . , n] might be missing from chn_approx at any given time. Hence, rather than a single pair [k, n], we use a set of pairs {[k0, k1], . . . , [km, km+1]}; each pair efficiently represents a gap-free range of numbers, while missing IDs are represented by gaps between the “high” element of one pair and the “low” element of the next. This data structure is a 1-dimensional range tree [8]; we call the compression technique it allows range compression.

Range compression can be viewed as a generalization of the “low water mark” used by reliable delivery schemes such as TCP, in which senders assign sequence numbers to packets and receivers send acks to indicate the prefix of the sequence they have received. Rather than requiring programmers to manipulate sequence numbers and use integer inequality, range compression achieves a similar degree of efficiency while allowing the program to deal with an unordered set of events. This has two benefits: first, range compression automatically handles situations in which IDs are omitted or delivered out-of-order, without requiring the programmer to explicitly track a “low water mark.” Second, set-oriented programs are convenient to develop, particularly in set-oriented languages such as Bloom.

In the current Edelweiss prototype, developers explicitly enable range compression by using a new collection type, range. For example, “range : chn_approx, [:id]” would replace line 8 in Figure 2. In addition, the Edelweiss runtime automatically applies
range compression to outband channel messages when profitable—this allows a single acknowledgment to describe the successful delivery of many chn messages. It would be possible to apply range compression to all collections by default, but we haven’t found the need to implement this yet.

4. RELIABLE BROADCAST

In the previous section, we showed how a declarative Edelweiss program for reliable unicast can be implemented efficiently. In the following sections, we show how the same techniques can be applied to a series of more complicated Edelweiss programs. We begin by generalizing reliable unicast to reliable broadcast and then in Section 5 we use reliable broadcast to build a replicated key-value store. In Section 6, we then extend the key-value store to provide causal consistency guarantees. Finally, Section 7 discusses how to implement atomic read/write registers. Importantly, all of these programs can be written in Edelweiss and implemented efficiently via extended versions of the techniques introduced in Section 3.

4.1 Fixed Membership

Figure 3 shows a naive reliable broadcast program. Any node can send a message by inserting into the log collection. The messages in the log are sent to every node in the group (line 11). When a node receives a message, it adds the message to its log (line 12); that node will re-broadcast the message in the future. Each message has a unique ID. To assign unique IDs without global coordination, a common technique is to use \( nodeid, seqnum \) pairs, where \( seqnum \) is a node-local sequence number. We assume the broadcast group contains a fixed set of nodes; we relax this assumption in Section 4.2.

The program in Figure 3 is simple—indeed, it closely resembles the pseudocode for the textbook reliable broadcast algorithm [32]—but as with naive reliable unicast (Figure 1), it suffers from unbounded messaging and storage.

4.1.1 Bounded Messaging

As with reliable unicast (Section 3.1), the ARM rewrite automatically avoids unbounded messaging by inserting an acknowledgment protocol. We omit the rewritten program for space reasons, but the same acking scheme can be used. To avoid sending acknowledged messages, line 11 is rewritten to:

\[
chn \leftarrow (node * log).pairs \{[n,1] n + 1\}
\]

...notin(chn_approx, 0 => :addr, 1 => :id)

As chn_approx grows, it matches tuples in the output of the join between node and log; our goal is to use tuples in chn_approx to reclaim from the join’s input collections. To do so, Edelweiss must reason about how the join’s inputs can grow over time. For example, to reclaim a tuple \( r \) from log, Edelweiss must ensure that all future join outputs that depend on \( r \) have already been produced and that all such output tuples have a match in chn_approx.

This can be done by adapting the concept of punctuation, which were first introduced for processing queries over unbounded data streams [43]. A punctuation is a guarantee that no more tuples matching a predicate will appear in a collection. For now, we consider a simple class of punctuations: the assertion that no more tuples will ever appear in a collection. Given \( (X * Y) \).notin(2), suppose we want to reclaim a tuple \( y \in Y \). A punctuation asserting that no more \( X \) tuples will arrive implies that we know about all the \( X \) tuples that will ever match \( y \). Hence, once we have seen a match in \( Z \) for all the current join results that depend on \( y \), we can safely be reclaimed. Of course, the symmetry of the join operator means that a similar argument allows reclamation from \( X \) given a punctuation on \( Y \).

Returning to reliable broadcast, Edelweiss can reclaim tuples from log given a punctuation that no new node tuples will appear. At the beginning of this section, we assumed that the broadcast group is fixed—hence, the necessary punctuation can safely be produced. As a syntactic convenience, Edelweiss defines a new collection type called sealed to hold a collection whose contents are fixed after the system has been initialized. Declaring that node is sealed (line 5 in Figure 3) allows the Edelweiss runtime to automatically emit a punctuation for the collection. We omit the complete compiler output for space reasons but give the main idea: punctuations are generated by the same node will form a gap-free sequence.

4.1.2 Acknowledgments and Logical Clocks

After applying ARM, each node persists two collections that grow over time: log, the set of messages, and chn_approx, which holds each node’s knowledge about the messages that have been received by the other nodes. The storage required for chn_approx can be reduced via range compression, as described in Section 3.3. Recall that for reliable unicast, chn_approx will eventually be range-compressed to a single value, effectively yielding a logical clock. With broadcast, chn_approx will contain a single “clock” value for each node in the broadcast group and hence behaves similarly to a vector clock [30]. That is, the combination of ARM and range compression essentially “discovers” the relationship between event histories [40] and logical clocks! Edelweiss allows programmers to simply manipulate sets of immutable events; it then automatically produces the corresponding “clock” management code.

4.1.3 Punctuations

To reclaim messages from log, we can use the DR+ rewrite introduced in Section 3.2. However, reclaiming broadcast messages is more complicated than reclaiming unicast messages—intuitively, a unicast message can be reclaimed as soon as it has been successfully delivered to the recipient, whereas a broadcast message can only be reclaimed once it has been delivered to every node in the group. This difference is manifest in the program:

\[
chn \leftarrow (node * log).pairs \{[n,1] n + 1\}
\]

...notin(chn_approx, 0 => :addr, 1 => :id)

Note that + concatenates tuples. Broadcast is expressed as Cartesian product—i.e., a join between node and log with no join predicate.

We implemented \( nodeid, seqnum \) pairs as a single 64-bit integer consisting of a 32-bit node ID prepended to a 32-bit sequence number. This is compatible with range compression, since multiple IDs generated by the same node will form a gap-free sequence.

The notin predicate checks for an equality match between the first two columns of the join result against the addr and id fields of chn_approx. Note that unlike with reliable unicast, acks include node addresses as well as message IDs. This is necessary because a message might be delivered successfully to some nodes but not others; moreover, ARM deduces this automatically because the key of chn contains both fields (line 7).
represented by tuples in “seal tables” that are defined automatically by Edelweiss. The rules generated by DR+ join against the seal table for node and thereby wait for a punctuation on node before reclaiming any tuples from log. Hence, by exploiting the fact that node is sealed, DR+ confirms that messages can safely be reclaimed once they have been successfully delivered to all nodes.

4.2 Dynamic Membership

Assuming a fixed broadcast group simplifies deciding when a log entry can be reclaimed. If we relax this assumption (by declaring that node is a table in line 5 of Figure 3), DR+ can no longer reclaim tuples from log. Indeed, reclaiming from log would be unsafe: if a new tuple appeared in node, the join between node and log (line 11) implies that all log messages should be delivered to the newly joined node. Hence, reclaiming from log would change user-visible program behavior.

To allow both dynamic membership and safe reclamation from the log table, we need to change the program to identify situations in which log messages should not be delivered to new nodes; such messages can then be reclaimed. We can achieve this using epochs: each epoch has a fixed set of members and each message identifies the epoch to which it belongs. To change the membership of the broadcast group, the system moves to a new epoch with a different set of members. Hence, once the membership of an epoch has been fixed and a message has been delivered to all the members of that epoch, that message can safely be reclaimed.

Figure 4 contains an Edelweiss program implementing this design. Note that this program is nearly identical to reliable broadcast with fixed membership, except that the Cartesian product between node and log (line 11 in Figure 3) has been replaced with an equipoint on epoch (line 11 in Figure 4). DR+ automatically exploits the equipoint predicate to enable reclamation using finer-grained rejections: given \((X \times Y).pairs: (k_1 \Rightarrow k_2).notin(Z)\) and a punctuation that asserts that no more \(Y\) tuples will arrive with \(k_2 = c\), all \(X\) tuples with \(k_1 = c\) are now eligible for reclamation. In general, DR+ can exploit rejections that match the join predicate: since a Cartesian product is essentially a join with no predicate, DR+ can only use whole-relation rejections for such operators.

Applying DR+ to the epoch-based broadcast program produces the expected results: given a punctuation asserting that no more node facts will be observed for epoch \(k\), the rules produced by DR+ automatically reclaim any message in epoch \(k\) that has been delivered to all the members of that epoch. The procedure for deciding to move to a new epoch is orthogonal to this program; a common approach is to use a separate (and more expensive) protocol based on distributed consensus [9, 33]. After a new epoch has been decided on, the consensus mechanism would then broadcast a corresponding punctuation, allowing Edelweiss to reclaim messages.

DR+ also allows reclamation from node using rejections on log: if we can guarantee that no more log facts will arrive for a given epoch, then once every log fact that has been delivered to some node \(n\), \(n\) can be reclaimed from node. This follows from the symmetry of the join predicate on line 11.

5. KEY-VALUE STORE

In this section, we use reliable broadcast to build a replicated key-value store (KVS). We show that Edelweiss automatically produces a safe, effective storage reclamation scheme for this program.

Using reliable broadcast to build a KVS is a well-known technique [15, 22, 45]. The store contains a set of keys and associated values. Clients submit insert and delete operations; replicas apply these operations to maintain their local view. Each insert operation has a unique ID and a delete operation contains the ID of its corresponding insertion. Following prior work [45], we allow multiple insertions of the same key with different IDs; all such key-value pairs are included in the view. Hence, if a key appears multiple times, a given deletion applies to only one of the IDs associated with that key. The KVS is fully replicated. Building this design using reliable broadcast is straightforward: a log of insert and delete operations is broadcast to all nodes, and the set of live keys at any given replica consists of every insertion that has no matching deletion. Figure 5 contains a simple Edelweiss program that implements this scheme.

A natural question is how to bound the storage required for operation logs. In prior work [15, 19, 28, 45], researchers proposed hand-crafted protocols that allow safe reclamation by tracking each node’s knowledge of the state of the other nodes. We show how a similar scheme can safely and automatically be produced by Edelweiss from the simple, declarative program in Figure 5.

5.1 Reclaiming Insertions

Because insert and delete operations are used differently, we need to employ two different reclamation strategies. Inserts are used in two places: the broadcast rule (line 14) and the rule to compute the current view (line 18). Applying ARM to the broadcast rule, we get:

\[
\text{ins} \leftarrow (\text{node} * \text{ins}.log).pairs \{n, 1 \mid \text{n.addr} + 1\}
\]

Observe that \(\text{ins}.log\) only appears as the positive input to two \text{notin} operators, which makes it a candidate for the DR+ rewrite. As discussed in Section 3.2, we can reclaim from a collection when the absence of a tuple from the collection would not change user-visible program behavior. In this case, an \(\text{ins}.log\) tuple can be discarded when it has a match in both \text{ins}_chn.approx and...
where \( X \) will ever match \( \text{notin} \) is, DR \( \notin \) range compression. As a heuristic, we use the collection type of the negative input to the rewritten as follows:
\[ X \\text{.notin}(Y, :A \Rightarrow :B) \]
Negative Deletion is observed. Associated with each insertion, which is reclaimed when a matching rewriting the program to split \( X \) into every node, and (b) it has been delivered to every collection is added, \( \text{flat_map} \) (line 15) is used to "unnest" the array in the column), and log entries are moved from log to safe when their dependencies are met. The \( \text{flat_map} \) method (line 15) is used to "unnest" the array in the \( \text{deps} \) column.

A replica's view should reflect all the safe, undominated writes it has observed. If there are two writes to the same key and neither dominates the other, the writes are concurrent. Some systems handle this situation by invoking a commutative merge function [27, 34]. We include both versions of the key in the view; a client can then read both versions and resolve the conflict by issuing a new write that dominates both previous versions of the key.

6.2 Write Operations
To extend the key-value store presented in Section 5 to support causal consistency, we begin by considering how to support write operations. As in the simple KVS, each replica broadcasts its log of write operations to the other replicas. However, replicas may need to buffer writes they receive until the dependencies of those writes have been satisfied. Figure 6 shows an Edelweiss program fragment that implements this scheme in lines 14–19. Each log entry has a set of dependencies (represented as a nested array in the \( \text{deps} \) column), and log entries are moved from log to safe when their dependencies are met. The \( \text{flat_map} \) method (line 15) is used to "unnest" the array in the \( \text{deps} \) column.

A replica's view should contain all the safe, undominated writes it has observed, so next we need to determine which writes in \( \text{safe} \) have been dominated. Our initial implementation looked for paths in the transitive closure of the dependency graph—that is, \( w \) dominates \( w' \) if \( w \rightarrow w' \). However, while this design was correct, it prevented dominated writes from being reclaimed by Edelweiss. On closer examination, we realized that in this scheme, reclaiming dominated writes is not permissible because a dominated write to
To implement causal consistency for reads, we first need to recompute a read's dependencies have been satisfied, the replica responds via the request by sending a message over the sent read operations explicitly. In Figure 7, a client initiates a read each replica uses the operation log to compute the safe that no other join results will depend on safe. Similarly, the join on line 22 matches the other join input, but Edelweiss supports several special cases that can be reclaimed as soon as they are produced: while logically the dom can be reclaimed once they have been dominated. Hence, Edelweiss taught us something surprising about our own program!

In recent work [27], Lloyd et al. avoid the need to retain the complete dependency graph by requiring that a write to key k must include a dependency on a previous write to k (if any exists). We make the same assumption in Figure 6; hence, we can identify dominated writes by looking for another write to the same key. Hence, Edelweiss taught us nothing surprising about our own program!

Edelweiss generates an effective reclamation scheme for this program. As expected, log entries are reclaimed once they have been delivered to all replicas and their dependencies have been met at the local replica. Tuples in dep can be reclaimed once their set of dominated writes grows over time. Edelweiss observes that a dominated write is only needed to remove tuples from safe. Hence dom and safe_dep facts can be immediately reclaimed.

Facts in safe can be reclaimed once they have been dominated. Note that safe appears in two joins (lines 21 and 22); Edelweiss must determine that reclaiming from safe will not change the results of either join. In general, this might require computations on the other join input, but Edelweiss supports several special cases that avoid the need for user-supplied computations for this program. On line 21, dep is produced by a flat_map operation involving the key of log; hence, Edelweiss can infer computations on the first column of dep. This matches our intuition that no new dependencies will be observed for a given write. Similarly, the join on line 22 matches the key column of safe with the key columns of safe_dep; hence, once a safe tuple s has a match in safe_dep, Edelweiss knows that no other join results will depend on s.

6.3 Read Operations

The KVS in Section 5 does not explicitly support read operations: each replica uses the operation log to compute the view collection, and clients read by (implicitly) examining the replica’s current view. To implement causal consistency for reads, we first need to recompute read operations explicitly. In Figure 7, a client initiates a read request by sending a message over the req_chn channel; when the read’s dependencies have been satisfied, the replica responds via the resp_chn channel. If a read request is unsafe (i.e., if it specifies dependences that are not satisfied by the local replica), the replica buffers the request until its dependencies have been met.

Edelweiss provides several features that simplify this program. First, ARM prevents unbounded resp_chn messages by inserting client-side acknowledgment logic (we omit the client code for brevity). Second, DR+ reclaims read_buf messages when the read’s dependencies have been satisfied. Finally, DR– reclaims read.resp tuples when the client’s acknowledgment is received. Perhaps more importantly, Edelweiss automatically handles the interactions between the reclamation conditions of all these rules and the safety and dominance rules in Figure 6, allowing the developer to focus on implementing correct application-level behavior.

7. READ/WRITE REGISTERS

In this section, we use Edelweiss to implement atomic readwrite registers [25], a common building block for distributed algorithms. An atomic register allows a single writer to interact with multiple concurrent reader processes and guarantees that the values returned by reads are consistent with a serial ordering of operations. Reading an atomic register reflects the latest value written (in contrast to the KVS presented in Section 5, in which each insertion for a given key adds to its list of values).

While traditional designs utilize mutable storage, we show how a mutable register interface can be implemented via an Edelweiss program that accumulates an immutable event log. We then extend the program to support atomic writes to multiple registers and multi-register reads that reflect a consistent snapshot. For all of these programs, Edelweiss automatically obtains the interactions between the reclamation conditions of all these rules and the safety and dominance rules in Figure 6, allowing the developer to focus on implementing correct application-level behavior.

7.1 Single-register Writes

Figure 8 contains an implementation of atomic registers in Edelweiss. As in the KVS program (Section 5), the current register values are computed as a view over an append-only event log. The collection write_log records the history of writes to a set of registers. The atomic register model assumes a single writer per register [25], so each entry in write_log for a particular register has exactly one predecessor—the previously written value—which it supersedes. The dom table contains those write IDs that are dominated by a “more recent” entry in write_log (line 17)—i.e., those IDs that are the predecessor (or prev_wid) of another record. Line 18 defines the view live as the subset of records in write_log that...
are not dominated by a more recent record. The single writer assumption implies that live contains exactly one record at any time for a given register.

To write to a register, a client inserts into the write collection. If the write has not yet been applied to the write log, this generates a write_event (line 13). We then insert a new record into write_log containing the new value along with the ID of the previous write to that register, or 0 if this is the first write (lines 14–16).

The program has three persistent collections: write, write_log, and dom. Edelweiss uses DR+ to reclaim records from write as soon as they are reflected in write_log. DR- can be used to reclaim from both write_log and dom. As discussed in Section 5.2, DR- exploits the fact that the notin predicate on line 18 only uses the key column of write_log. Hence, Edelweiss knows that once a write_log entry has been dominated, both the write_log and dom facts can be reclaimed—as long as we can prevent any duplicate write_log tuples from appearing. To enable this, Edelweiss automatically creates a range collection that stores all the write IDs ever observed—fortunately, this set can be effectively range compressed.

7.2 Multi-register Writes

Next, we show how to support atomic updates to multiple keys (Figure 9). That is, we allow writes to be supplied for multiple keys and then eventually committed, at which point all the associated writes are applied atomically.

Clients insert values into write as before, but these values are not applied to write_log until a commit record exists for their batch (lines 14–15). As in the atomic register program, we assume that at most one value for every register exists in commit_event at any time. For each such register, a fact is inserted into write_log reflecting the last effective write for that register (live.wid) as its prev_wid (lines 16–18).

Edelweiss utilizes the DR+ rewrite and client-supplied punctuations to synthesize garbage collection logic for this program. Lines 14–15 join write with commit, but the lefts operator preserves only records from write into the notin operator. Hence, DR+ recognizes that it can “push up” the notin operator into the join and reclaim redundant records from write as soon as they are reflected in write_log. In order to reclaim records from commit, however, we need to rule out the possibility of a write record appearing after its corresponding commit record has been deleted. If the client seals write.batch as part of batch commit—a promise to produce no future writes within that batch—DR+ uses this additional information to generate rules that safely reclaim from commit.

7.3 Snapshot Reads

Ensuring that all writes within a batch become visible atomically is not sufficient to guarantee consistent reads of multiple registers. Consider the following history in which batch T2 commits after T1:

- T1: \(W(x = 1), W(y = 1)\)
- T2: \(W(x = 2), W(y = 2)\)

Without any synchronization, a multi-register read \((R_1)\) could view a state not produced by a serial ordering of write batches:

\[R_1: R(x = 1), R(y = 2)\]

We could rule out this anomaly by forcing multi-register reads to participate in a concurrency control scheme with write batches and commit according to a serializable ordering. In a workload in which reads are common or long-running, however, such a scheme can have undesirable effects, interfering with write batches by causing them to wait or abort. An alternative approach is to use a multiversioning scheme in which read batches do not interact with writes, but nevertheless perceive a snapshot of the store consistent with a serial ordering of writes [36, 39, 44]. This requirement implies that it is not necessarily safe to reclaim entries in the log as soon as they are dominated by a more recent write—these entries must persist
8. EVALUATION

In this section, we evaluate two aspects of Edelweiss. First, we verify that the storage reclamation logic produced by Edelweiss works effectively. Second, we evaluate the quantitative and qualitative benefits of programming in Edelweiss. We show that Edelweiss enables significant reductions in code size and complexity.

8.1 Storage Reclamation

To validate that Edelweiss is effective at safely and automatically reclaiming storage, we study how the causal KVS (Section 6) behaves in two scenarios. First, we show that Edelweiss automatically discards dominated writes. Second, we report the behavior of the causal KVS during a network partition, confirming the expected behavior that partitions prevent storage from being reclaimed [27].

8.1.1 Dominated Writes

As discussed in Section 6, Edelweiss automatically infers that a write operation in the causal KVS can be discarded when (a) the write has been replicated to all nodes, and (b) the write has been dominated by another write, because dominated writes no longer contribute to the replica’s current view. To verify this behavior, we created a single-site KVS and measured the storage requirements of the system over time. We submitted write operations at a constant rate (50 writes per second), but varied the percentage of writes that updated a previously written key. As the fraction of updates increases, the number of undominated writes in the replica’s view decreases, and hence we expect the program to require less storage.

Figure 11 reports the results of this experiment for several update percentages. As the fraction of updates in the workload increases, more dominated writes are observed and hence Edelweiss automatically reclaims more stored tuples.

8.1.2 Network Partitions

Next, we consider how the reclamation protocol generated by Edelweiss behaves during network partitions. When the network is partitioned, write operations cannot be replicated to all replicas; hence, those operations must be retained until the partition heals, temporarily increasing storage requirements.

To study this behavior, we created a simple causal KVS cluster with two replicas, A and B. A single client continuously submits writes to replica A. Every write is a dominating update, so when the network is connected we expect the storage required at each replica to remain constant over time. We then simulated two network partitions by dropping all channel messages sent between A and B.

Figure 12 reports the storage required by replica A for this experiment. The network is connected for the first 30 seconds of the experiment; as expected, the replica’s storage requirements do not increase. Starting at 30 seconds and continuing for the next 50 seconds, we simulated a network partition. The reclamation logic generated by Edelweiss does not allow write operations to be discarded until they have been successfully replicated, and hence the number of tuples retained by replica A grows.

After 80 seconds, the partition is healed. Replica A promptly sends its backlog of write operations to B, and the ARM-generated acknowledgment logic at B informs A that the writes have been delivered successfully. Edelweiss can then safely discard those write operations, leading to an immediate drop in storage. As the experiment continues, a similar pattern of behavior can be observed: storage remains stable as new writes arrive while the network is connected, then grows during the partition that begins at 140 seconds, and finally shrinks again once the partition is healed.

8.2 Program Size and Complexity

Next, we consider the code size and qualitative complexity for several common distributed algorithms implemented using Edelweiss.
We observed this first-hand: in the initial version of the key-value store, we eventually realized that reclaiming deletions in this program when (logically) deleted keys should be omitted from the view—the communities. As noted in Section 1, the pattern of operation logging implementation of dominated writes in the causal KVS. Edelweiss helped us identify a similar logic error in our initial store, we arranged for delete operations to specify a key to be safe reclamation, the result is a storage leak rather than data loss.

This also means that if the program’s semantics do not allow safe reclamation, the result is a storage leak rather than data loss. We observed this first-hand: in the initial version of the key-value store, we arranged for delete operations to specify a key to be removed, rather than an insertion ID. As a result, Edelweiss was unable to reclaim delete operations. While we were initially puzzled, we eventually realized that reclaiming deletions in this program would be unsafe in principle: the program allows multiple insertions with the same key (and different IDs), so reclaiming deletions would change the behavior of the program. As described in Section 6.2, Edelweiss helped us identify a similar logic error in our initial implementation of dominated writes in the causal KVS.

<table>
<thead>
<tr>
<th>Description</th>
<th>Input</th>
<th>Rewritten</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reliable unicast (§3)</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>Reliable broadcast, fixed (§4.1)</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>Reliable broadcast, epoch-based (§4.2)</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>Causal broadcast (N/A)</td>
<td>6</td>
<td>14</td>
</tr>
<tr>
<td>Request-response pattern (N/A)</td>
<td>7</td>
<td>16</td>
</tr>
<tr>
<td>Key-value store (§5)</td>
<td>5</td>
<td>23</td>
</tr>
<tr>
<td>Key-value store with causal consistency (§6)</td>
<td>19</td>
<td>62</td>
</tr>
<tr>
<td>Atomic registers (§7.1)</td>
<td>4</td>
<td>11</td>
</tr>
<tr>
<td>Atomic registers, multi-key writes (§7.2)</td>
<td>4</td>
<td>17</td>
</tr>
<tr>
<td>Atomic registers, snapshot reads (§7.3)</td>
<td>8</td>
<td>23</td>
</tr>
</tbody>
</table>

Table 4: Code size comparison.

In Table 4, the “Input” column shows the number of rules in each Edelweiss program, while the “Rewritten” column shows the number of rules in the corresponding Bloom program that is produced automatically by the Edelweiss compiler. That is, the rewritten programs include mechanisms for knowledge propagation and storage reclamation that are inferred automatically by Edelweiss. In some cases, the rewritten programs contain a small number of redundant rules that could be avoided by a careful Bloom developer. Similarly, a developer might choose to ignore reclamation conditions that are supported by our rewrites—for example, if punctuations for a certain collection will never be supplied, reclamation rules that depend on those punctuations can be omitted. Nevertheless, when examining the rewritten programs by hand, we found they had a similar structure to hand-crafted acknowledgment and reclamation mechanisms we have written in the past.

All of the programs in Table 4 are concise, particularly in comparison to implementations using traditional imperative languages: for example, a recent causally consistent key-value store prototype required about 13,000 lines of C++ code [27]. Nevertheless, the Edelweiss programs are smaller than their Bloom counterparts by a factor of two or more. Perhaps more importantly, Edelweiss relieves programmers of the need to reason about when storage can safely be deallocated. Instead, the programmer specifies when information remains useful to their application and Edelweiss produces a reclamation scheme that is consistent with those requirements. For example, in the KVS (Section 5), the Edelweiss program specifies when (logically) deleted keys should be omitted from the view—the programmer does not need to consider when the associated insert and delete operations should be physically reclaimed.

This also means that if the program’s semantics do not allow safe reclamation, the result is a storage leak rather than data loss. We observed this first-hand: in the initial version of the key-value store, we arranged for delete operations to specify a key to be removed, rather than an insertion ID. As a result, Edelweiss was unable to reclaim delete operations. While we were initially puzzled, we eventually realized that reclaiming deletions in this program would be unsafe in principle: the program allows multiple insertions with the same key (and different IDs), so reclaiming deletions would change the behavior of the program. As described in Section 6.2, Edelweiss helped us identify a similar logic error in our initial implementation of dominated writes in the causal KVS.

9. RELATED AND FUTURE WORK

Our work on Edelweiss is related to prior work by several research communities. As noted in Section 1, the pattern of operation logging with periodic background reclamation appears frequently in distributed storage and data management. Proposed designs typically differ along a few dimensions, such as how “common knowledge” about the state of other nodes is represented, how this knowledge is communicated (e.g., bundled with log messages [15, 45] or sent via a separate mechanism [7, 28]), and the criteria for determining that an operation is “stable” and can be discarded (e.g., some systems use wall-clock time, waiting a period of time in which replica synchronization is likely to have occurred [13]; in others, the system explicitly tracks the logical clocks of all nodes, which requires all nodes to be available to allow log entries to be reclaimed [45]).

Garbage collection has been extensively studied by the programming language community for both single-site and distributed [1] programs. Traditional garbage collection is applied to a reference graph: subgraphs that are not reachable from one or more “root” vertices can safely be reclaimed. This relies on the property that such objects will never be reachable in the future, which is a special-case of the kind of “henceforth no longer useful” properties exploited by Edelweiss. It would be interesting to see how naturally Edelweiss could be enhanced to synthesize traditional GC schemes.

This paper focuses on systems in which knowledge can eventually be discarded; there is a related design pattern in which nodes accumulate knowledge and then periodically summarize or reorganize it, e.g., in the form of a “checkpoint.” Examples include write-ahead logging in database systems [31], log-structured file systems [37], and rollback-based recovery in distributed systems [42]. We are working to extend our analysis to support checkpointing.

This paper also relates to efforts to provide principled foundations for eventually consistent systems. In prior work, we proposed the CALM Theorem, which shows that monotonic logic programs are deterministic (“confluent”) and hence eventually consistent [4]. This shares similarities to Edelweiss and the way in which the ELE model sidesteps consistency concerns. However, most of the programs in this paper use non-monotonic operators (particularly negation) and are not confluent. We are currently exploring how to harmonize the notion of CALM consistency with ELE and provide consistency analysis for Edelweiss programs.

In this paper, we assume that nodes accumulate knowledge as a set of facts that grows over time, but this can be generalized from sets to join semilattices to represent other kinds of growth, e.g., integers that increase numerically or Boolean values that move from false to true [12]. Lattices often require a form of periodic garbage collection to restore efficiency [41]; extending Edelweiss to lattices is a natural direction for future work.

Our development of reclamation techniques for Edelweiss has been somewhat ad hoc and driven by the practical programs we have studied. Given a program for which the current Edelweiss prototype does not reclaim storage, it is often unclear whether the problem lies in the program or in the Edelweiss implementation—that is, could a more sophisticated analysis successfully reclaim storage for the program or is the program “un-reclaimable” in principle? Both theoretical and practical developments would be useful here. First, we would like to formally characterize the class of Edelweiss programs that can be evaluated with bounded storage. Second, we could enhance Edelweiss to provide feedback to developers about how program semantics influence storage requirements. For example, Edelweiss could describe the circumstances under which a given fact can be reclaimed, or generate an execution trace in which prematurely reclaiming a fact leads to incorrect program behavior.

10. CONCLUSION

Edelweiss demonstrates for the first time that the benefits of the ELE pattern for distributed programming do not require custom
state reclamation code. The fact that this result arose from a basis in Bloom is not coincidental. Rewrites like Difference Reclamation were inspired directly from the use of a declarative, set-oriented language. Moreover, the clear data dependencies in a declarative language made our analysis code easy to write. It is an open question whether our techniques can be transferred to (immutable versions of) more popular imperative languages, which could be quite useful in practice. Meanwhile, it is our experience that distributed programming design patterns like ELE are quite well-served by declarative distributed languages, and we believe that the design and analysis of such languages is a fruitful direction for further exploration.

Acknowledgments
This paper benefited from discussions with William Marczak and Mooly Sagiv. We would also like to thank Andrew R. Gross, Evan Sparks, and the reviewers for their helpful feedback on this paper. This work was supported by the AFOSR (grant FA95500810352), NSERC, NSF (grants IIS-0803690 and IIS-0963922), and gifts from EMC, Microsoft Research, and NTT.

11. REFERENCES