Checking Invariant Confluence, In Whole or In Parts

Michael Whittaker
UC Berkeley
Berkeley, CA
mjwhittaker@berkeley.edu

Joseph M. Hellerstein
UC Berkeley
Berkeley, CA
hellerstein@berkeley.edu

ABSTRACT

Strongly consistent distributed systems are easy to reason about but face fundamental limitations in availability and performance. Weakly consistent systems can be implemented with very high performance but place a burden on the application developer to reason about complex interleavings of execution. Invariant confluence provides a formal framework for understanding when we can get the best of both worlds. An invariant confluent object can be efficiently replicated with no coordination needed to preserve its invariants. However, actually determining whether or not an object is invariant confluent is challenging.

In this paper, we establish conditions under which a commonly used sufficient condition for invariant confluence is both necessary and sufficient, and we use this condition to design a general-purpose interactive invariant confluence decision procedure. We then take a step beyond invariant confluence and introduce a generalization of invariant confluence, called segmented invariant confluence, that allows us to replicate non-invariant confluent objects with a small amount of coordination. We implement these formalisms in a prototype called Lucy and find that our decision procedures efficiently handle common real-world workloads including foreign keys, escrow transactions, and more.

1. INTRODUCTION

When an application designer decides to replicate a piece of data, they have to make a fundamental choice between weak and strong consistency. Replicating the data with strong consistency using a technique like distributed transactions [7] or state machine replication [14] makes the application designer's life very easy. To the developer, a strongly consistent system behaves exactly like a single-threaded system running on a single node, so reasoning about the behavior of the system is simple [12]. Unfortunately, strong consistency is at odds with performance. The CAP theorem and PACELC theorem tell us that strongly consistent systems suffer from higher latency at best and unavailability at worst [9, 1]. On the other hand, weak consistency models like eventual consistency [24], PRAM consistency [17], causal consistency [2], and others [19, 20] allow data to be replicated with high availability and low latency, but they put a tremendous burden on the application designer to reason about the complex interleavings of operations that are allowed by these weak consistency models. In particular, weak consistency models strip an application developer of one of the earliest and most effective tools that is used to reason about the execution of programs: application invariants [13, 5] such as database integrity constraints [11]. Even if every transaction executing in a weakly consistent system individually maintains an application invariant, the system as a whole can produce invariant-violating states.

Is it possible for us to have our strongly consistent cake and eat it with high availability too? Can we replicate a piece of data with weak consistency but still ensure that its invariants are maintained? Yes... sometimes. Bailis et al. introduced the notion of *invariant confluence* as a necessary and sufficient condition for when invariants can be maintained over replicated data without the need for any coordination [3]. If an object is invariant confluent with respect to an invariant, we can replicate it with the performance benefits of weak consistency and (some of) the correctness benefits of strong consistency.

Unfortunately, to date, the task of identifying whether or not an object actually is invariant confluent has remained an exercise in human proof generation. Bailis et al. manually categorized a set of common objects, transactions, and invariants (e.g. foreign key constraints on relations, linear constraints on integers) as invariant confluent or not. Hand-written proofs of this sort are unreasonable to expect from programmers. Ideally we would have a general-purpose program that could automatically determine invariant confluence for us. **The first main thrust of this paper is to make invariant confluence checkable:** to design a general-purpose invariant confluence decision procedure, and implement it in an interactive system.

Unfortunately, designing such a general-purpose decision procedure is impossible because determining the invariant confluence of an object is undecidable in general. Still, we can develop a decision procedure that works well in the common case. For example, many prior efforts have developed decision procedures for *invariant closure*, a sufficient (but not necessary) condition for invariant confluence [16, 15]. The existing approaches check whether an object is invari-

The original version of this paper is entitled "Interactive Checks for Coordination Avoidance" and was published in PVLDB Vol. 12.

 This is a minor revision of the paper entitled "Interactive Checks for Coordination Avoidance", published in PVLDB, Vol. 12, No. 1, ISSN 2150-8097. DOI: https://doi.org/10.14778/3275536.3275538

ant closed. If it is, then they conclude that it is also invariant confluent. If it's not, then the current approaches are unable to conclude anything about whether or not the object is invariant confluent.

In this paper, we take a step back and study the underlying reason *why* invariant closure is not necessary for invariant confluence. Using this understanding, we construct a set of modest conditions under which invariant closure and invariant confluence are in fact *equivalent*, allowing us to reduce the problem of determining invariant confluence to that of determining invariant closure. Then, we use these conditions to design a general-purpose interactive invariant confluence decision procedure.

The second main thrust of this paper is to partially avoid coordination even in programs that require it, by generalizing invariant confluence to a property called *segmented invariant confluence*. While invariant confluence characterizes objects that can be replicated *without any* coordination, segmented invariant confluence allows us to replicate non-invariant confluent objects with only *occasional* coordination. The main idea is to divide the set of invariant-satisfying states into *segments*, with a restricted set of transactions allowed in each segment. Within a segment, servers act without any coordination; they synchronize only to transition across segment boundaries. This design highlights the trade-off between application complexity and coordination-freedom; more complex applications require more segments which require more coordination.

Finally, we present Lucy: an implementation of our decision procedures and a system for replicating invariant confluent and segmented invariant confluent objects. Using Lucy, we find that our decision procedures can efficiently handle a wide range of common workloads. For example, in Section 6, we apply Lucy to foreign key constraints and escrow transactions. Lucy processes these workloads in less than half a second, and no workload requires more than 66 lines of code to specify.

2. INVARIANT CONFLUENCE

Informally, a replicated object is **invariant confluent** with respect to an invariant if every replica of the object is guaranteed to satisfy the invariant despite the possibility of different replicas being concurrently modified or merged together [3]. In this section, we make this informal notion of invariant confluence precise.

We begin by introducing our system model of replicated objects in which a distributed object and an invariant are replicated across a set of servers. Clients send transactions to servers, and servers execute transactions so long as they maintain the invariant. Servers execute transactions without coordination, but to avoid state divergence, servers periodically gossip with one another and merge their replicas.

2.1 System Model

A **distributed object** $O = (S, \sqcup)$ consists of a set S of states and a binary merge operator $\sqcup : S \times S \to S$ that merges two states into one. A **transaction** $t : S \to S$ is a function that maps one state to another. An **invariant** I is a subset of S. Notationally, we write $I(s)$ to denote that s satisfies the invariant (i.e. $s \in I$) and $\neg I(s)$ to denote that s does not satisfy the invariant (i.e. $s \notin I$).

Example 1. $O = (\mathbb{Z}, \max)$ is a distributed object consisting of integers merged by the max function; $t(x) = x + 1$ is a transaction that adds one to a state; and $\{x \in \mathbb{Z} \mid x \geq 0\}$ is the invariant that states x are non-negative.

In our system model, a distributed object O is replicated across a set $p_1, \ldots, p_n$ of n servers. Each server p_i manages a replica $s_i \in S$ of the object. Every server begins with a start state $s_0 \in S$, a fixed set T of transactions, and an invariant I. Servers repeatedly perform one of two actions.

First, a client can contact a server p_i and request that it execute a transaction $t \in T$. p_i speculatively executes t, transitioning from state s_i to state $t(s_i)$. If $t(s_i)$ satisfies the invariant—i.e. $I(t(s_i))$—then p_i commits the transaction and remains in state $t(s_i)$. Otherwise, p_i aborts the transaction and reverts to state s_i.

Second, a server p_i can send its state s_i to another server p_j with state s_j causing p_j to transition from state s_j to state $s_i \sqcup s_j$. Servers periodically merge states with one another in order to keep their states loosely synchronized. Note that unlike with transactions, servers *cannot* abort a merge; server p_j must transition from s_j to $s_i \sqcup s_j$ whether or not $s_i \sqcup s_j$ satisfies the invariant.

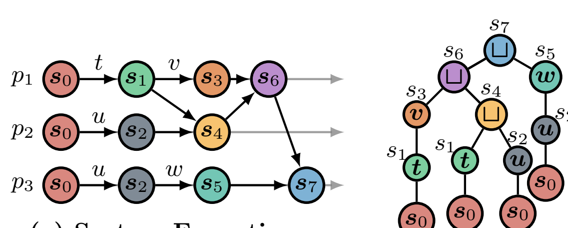
Informally, O is **invariant confluent** with respect to s_0, T, and I, abbreviated (s_0, T, I)**-confluent**, if every replica $s_1, \ldots, s_n$ is guaranteed to always satisfy the invariant I in every possible execution of the system.

2.2 Expression-Based Formalism

To define invariant confluence formally, we represent a state produced by a system execution as a simple expression generated by the grammar

$$e ::= s \mid t(e) \mid e_1 \sqcup e_2$$

where s represents a state in S and t represents a transaction in T. As an example, consider the system execution in Figure 1a in which a distributed object is replicated across servers p_1, p_2, and p_3. Server p_3 begins with state s_0, transitions to state s_2 by executing transaction u, transitions to state s_5 by executing transaction w, and then transitions to state s_7 by merging with server p_1. Similarly, server p_1 ends up with state s_6 after executing transactions t and v and merging with server p_2. In Figure 1b, we see the abstract syntax tree of the corresponding expression for state s_7.



(a) System Execution

(b) Expression

Figure 1: A system execution and corresponding expression

We say an expression e is (s_0, T, I)**-reachable** if it corresponds to a valid execution of our system model. Formally, we define $\text{reachable}_{(s_0,T,I)}(e)$ to be the predicate that satisfies the following conditions:

- $\text{reachable}_{(s_0,T,I)}(s_0)$.
- For all expressions e and for all transactions t in the set T of transactions, if $\text{reachable}_{(s_0,T,I)}(e)$ and $I(t(e))$, then $\text{reachable}_{(s_0,T,I)}(t(e))$.
- For expressions e_1 and e_2, if $\text{reachable}_{(s_0,T,I)}(e_1)$ and $\text{reachable}_{(s_0,T,I)}(e_2)$, then $\text{reachable}_{(s_0,T,I)}(e_1 \sqcup e_2)$.

Similarly, we say a state $s \in S$ is (s_0, T, I)-reachable if there exists an (s_0, T, I)-reachable expression e that evaluates to s. Returning to Example 1 with start state $s_0 = 42$, we see that all integers greater than or equal to 42 (i.e. $\{x \in \mathbb{Z} \mid x \geq 42\}$) are (s_0, T, I)-reachable, and all other integers are (s_0, T, I)-unreachable.

Finally, we say O is **invariant confluent** with respect to s_0, T, and I, abbreviated **(s_0, T, I)-confluent**, if all reachable states satisfy the invariant:

$$\{s \in S \mid \text{reachable}_{(s_0,T,I)}(s)\} \subseteq I$$

3. INVARIANT CLOSURE

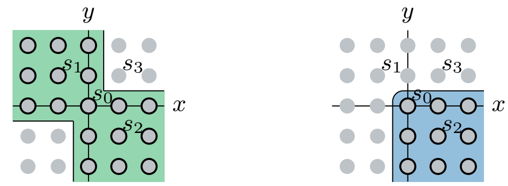
Our ultimate goal is to write a program that can automatically decide whether a given distributed object O is (s_0, T, I)-confluent. Such a program has to automatically prove or disprove that every reachable state satisfies the invariant. However, automatically reasoning about the possibly infinite set of reachable states is challenging, especially because transactions and merge functions can be complex and can be interleaved arbitrarily in an execution. Due to this complexity, existing systems that aim to automatically decide invariant confluence instead focus on deciding a sufficient condition for invariant confluence—dubbed **invariant closure**—that is simpler to reason about [16, 15]. In this section, we define invariant closure and study why the condition is sufficient but not necessary. Armed with this understanding, we present conditions under which it is both sufficient and necessary.

We say an object $O = (S, \sqcup)$ is **invariant closed** with respect to an invariant I, abbreviated **I-closed**, if invariant satisfying states are closed under merge. That is, for every state $s_1, s_2 \in S$, if $I(s_1)$ and $I(s_2)$, then $I(s_1 \sqcup s_2)$.

Theorem 1. *Given an object $O = (S, \sqcup)$, a start state $s_0 \in S$, a set of transactions T, and an invariant I, if $I(s_0)$ and if O is I-closed, then O is (s_0, T, I)-confluent.*

Theorem 1 states that invariant closure is sufficient for invariant confluence. Intuitively, our system model ensures that transaction execution preserves the invariant, so if merging states also preserves the invariant and if our start state satisfies the invariant, then inductively it is impossible for us to reach a state that doesn't satisfy the invariant.

This is good news because checking if an object is invariant closed is more straightforward than checking if it is invariant confluent. Existing systems typically use an SMT solver like Z3 to check if an object is invariant closed [8, 4, 10]. If it is, then by Theorem 1, it is invariant confluent. Unfortunately, invariant closure is *not* necessary for invariant confluence, so if an object is *not* invariant closed, these systems cannot conclude that the object is *not* invariant confluent. The reason why invariant closure is not necessary for invariant confluence is best explained through an example.



(a) Invariant **(b) Reachable points**

Figure 2: An illustration of Example 2

Example 2. Let $O = (\mathbb{Z} \times \mathbb{Z}, \sqcup)$ consist of pairs (x, y) of integers where $(x_1, y_1) \sqcup (x_2, y_2) = (\max(x_1, x_2), \max(y_1, y_2))$. Our start state $s_0 \in \mathbb{Z} \times \mathbb{Z}$ is $(0, 0)$. Our set T of transactions consists of two transactions: $t_{x+1}((x, y)) = (x + 1, y)$ which increments x and $t_{y-1}((x, y)) = (x, y - 1)$ which decrements y. Our invariant $I = \{(x, y) \in \mathbb{Z} \times \mathbb{Z} \mid xy \leq 0\}$ consists of all points (x, y) where the product of x and y is non-positive.

The invariant and the set of reachable states are illustrated in Figure 2 in which we draw each state (x, y) as a point in space. The invariant consists of the second and fourth quadrant, while the reachable states consist only of the fourth quadrant. From this, it is immediate that the reachable states are a subset of the invariant, so O is invariant confluent. However, letting $s_1 = (-1, 1)$ and $s_2 = (1, -1)$, we see that O is not invariant closed. $I(s_1)$ and $I(s_2)$, but letting $s_3 = s_1 \sqcup s_2 = (1, 1)$, we see $\neg I(s_3)$.

In Example 2, s_1 and s_2 witness the fact that O is not invariant closed, but s_1 is *not reachable*. This is not particular to Example 2. In fact, it is fundamentally the reason why invariant closure is not equivalent to invariant confluence. Invariant confluence is, at its core, a property of reachable states, but invariant closure is completely ignorant of reachability. As a result, invariant-satisfying yet unreachable states like s_1 are the key hurdle preventing invariant closure from being equivalent to invariant confluence. This is formalized by Theorem 2.

Theorem 2. *Consider an object $O = (S, \sqcup)$, a start state $s_0 \in S$, a set of transactions T, and an invariant I. If the invariant is a subset of the reachable states (i.e. $I \subseteq \{s \in S \mid \text{reachable}_{(s_0,T,I)}(s)\}$), then*

$$(I(s_0) \text{ and } O \text{ is } I\text{-closed}) \iff O \text{ is } (s_0, T, I)\text{-confluent}$$

The forward direction of Theorem 2 follows immediately from Theorem 1. The backward direction holds because any two invariant satisfying states s_1 and s_2 must be reachable (by assumption), so their join $s_1 \sqcup s_2$ is also reachable. And because O is (s_0, T, I)-confluent, all reachable points, including $s_1 \sqcup s_2$, satisfy the invariant.

4. INTERACTIVE DECISION PROCEDURE

Theorem 2 tells us that if all invariant satisfying points are reachable, then invariant closure and invariant confluence are equivalent. In this section, we present the interactive invariant confluence decision procedure shown in Algorithm 1, that takes advantage of this result.

A user provides Algorithm 1 with an object $O = (S, \sqcup)$, a start state s_0, a set of transactions T, and an invariant I. The user then interacts with Algorithm 1 to iteratively eliminate unreachable states from the invariant. Meanwhile, the algorithm leverages an invariant closure decision procedure to either (a) conclude that O is or is not (s_0, T, I)-confluent

Algorithm 1 Interactive invariant confluence decision procedure

```
// Return if O is (s_0, T, I)-confluent.
function IsInvConfluent(O, s_0, T, I)
    return I(s_0) and Helper(O, s_0, T, I, {s_0}, ∅)

// R is a set of (s_0, T, I)-reachable states.
// NR is a set of (s_0, T, I)-unreachable states.
// I(s_0) is a precondition.
function Helper(O, s_0, T, I, R, NR)
    closed, s_1, s_2 ← IsIclosed(O, I − NR)
    if closed then return true
    Augment R, NR with random search and user input
    if s_1, s_2 ∈ R then return false
    return Helper(O, s_0, T, I, R, NR)
```

or (b) provide counterexamples to the user to help them eliminate unreachable states. After all unreachable states have been eliminated from the invariant, Theorem 2 allows us to reduce the problem of invariant confluence directly to the problem of invariant closure, and the algorithm terminates. We now describe Algorithm 1 in detail. An example of how to use Algorithm 1 on Example 2 is given in Figure 3.

IsInvConfluent assumes access to an invariant closure decision procedure IsIclosed(O, I). IsIclosed(O, I) returns a triple (closed, s_1, s_2). closed is a boolean indicating whether O is I-closed. If closed is true, then s_1 and s_2 are null. Otherwise, s_1 and s_2 are a counterexample witnessing the fact that O is not I-closed. That is, $I(s_1)$ and $I(s_2)$, but $\neg I(s_1 \sqcup s_2)$ (e.g., s_1 and s_2 from Example 2). As we mentioned earlier, we can (and do) implement the invariant closure decision procedure using an SMT solver like Z3 [8].

IsInvConfluent first checks that s_0 satisfies the invariant. s_0 is reachable, so if it does not satisfy the invariant, then O is not (s_0, T, I)-confluent and IsInvConfluent returns false. Otherwise, IsInvConfluent calls a helper function Helper that—in addition to O, s_0, T, and I—takes as arguments a set R of (s_0, T, I)-reachable states and a set NR of (s_0, T, I)-unreachable states. Like IsInvConfluent, Helper(O, s_0, T, I, R, NR) returns whether O is (s_0, T, I)-confluent (assuming R and NR are correct). As Algorithm 1 executes, NR is iteratively increased, which removes unreachable states from I until I is a subset of $\{s \in S \mid \text{reachable}_{(s_0,T,I)}(s)\}$.

First, Helper checks to see if O is $(I - NR)$-closed. If IsIclosed determines that O is $(I - NR)$-closed, then by Theorem 1, O is $(s_0, T, I - NR)$-confluent, so

$$\{s \in S \mid \text{reachable}_{(s_0,T,I-NR)}(s)\} \subseteq I - NR \subseteq I$$

Because NR only contains (s_0, T, I)-unreachable states, then the set of (s_0, T, I)-reachable states is equal to set of $(s_0, T, I - NR)$-reachable states which, as we just showed, is a subset of I. Thus, O is (s_0, T, I)-confluent, so Helper returns true.

If IsIclosed determines that O is *not* $(I - NR)$-closed, then we have a counterexample s_1, s_2. We want to determine whether s_1 and s_2 are reachable or unreachable. We can do so in two ways. First, we can randomly generate a set of reachable states and add them to R. If s_1 or s_2 is in R, then they are reachable. Second, we can prompt the user to tell us directly whether the states are reachable or unreachable.

In addition to labelling s_1 and s_2 as reachable or unreachable, the user can also refine I by augmenting R and NR arbitrarily (see Figure 3 for example). In this step, we also make sure that $s_0 \notin NR$ since we know that s_0 is reachable.

After s_1 and s_2 have been labelled as (s_0, T, I)-reachable or not, we continue. If both s_1 and s_2 are (s_0, T, I)-reachable, then so is $s_1 \sqcup s_2$, but $\neg I(s_1 \sqcup s_2)$. Thus, O is not (s_0, T, I)-confluent, so Helper returns false. Otherwise, one of s_1 and s_2 is (s_0, T, I)-unreachable, so we recurse.

Helper recurses only when one of s_1 or s_2 is unreachable, so NR grows after every recursive invocation of Helper. Similarly, R continues to grow as Helper randomly explores the set of reachable states. As the user sees more and more examples of unreachable and reachable states, it often becomes easier and easier for them to recognize patterns that define which states are reachable and which are not. As a result, it becomes easier for a user to augment NR and eliminate a large number of unreachable states from the invariant. See Figure 3, for example. Once NR has been sufficiently augmented to the point that $I - NR$ is a subset of the reachable states, Theorem 2 guarantees that the algorithm will terminate after one more call to IsIclosed.

5. SEGMENTED INVARIANT CONFLUENCE

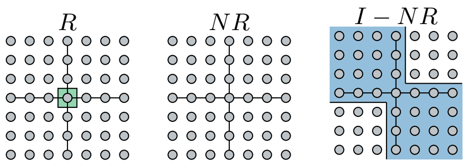
If a distributed object is invariant confluent, then the object can be replicated without the need for any form of coordination to maintain the object's invariant. But what if the object is *not* invariant confluent? In this section, we present a generalization of invariant confluence called **segmented invariant confluence** that can be used to maintain the invariants of non-invariant confluent objects, requiring only a small amount of coordination.

The main idea behind segmented invariant confluence is to segment the state space into a number of segments and restrict the set of allowable transactions within each segment in such a way that the object is invariant confluent *within each segment* (even though it may not be globally invariant confluent). Then, servers can run coordination-free within a segment and need only coordinate when transitioning from one segment to another. We now formalize segmented invariant confluence, describe the system model we use to replicate segmented invariant confluent objects, and introduce a segmented invariant confluence decision procedure.

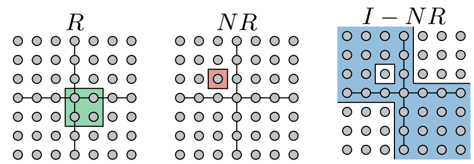
5.1 Formalism

Consider a distributed object $O = (S, \sqcup)$, a start state $s_0 \in S$, a set of transitions T, and an invariant I. A segmentation $\Sigma = (I_1, T_1), \ldots, (I_n, T_n)$ is a sequence of n segments (I_i, T_i) where every T_i is a subset of T and every $I_i \subseteq S$ is an invariant. Note that Σ is a sequence, not a set. The reason for this will become clear in the next subsection. O is **segmented invariant confluent** with respect to s_0, T, I, and Σ, abbreviated (s_0, T, I, Σ)**-confluent**, if the following conditions hold:

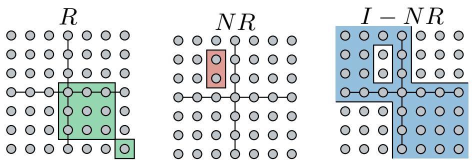
- The start state satisfies the invariant (i.e. $I(s_0)$).
- I is covered by the invariants in Σ (i.e. $I = \cup_{i=1}^{n} I_i$). Note that invariants in Σ do *not* have to be disjoint. That is, they do not have to partition I; they just have to cover I.
- O is invariant confluent within each segment. That is, for every $(I_i, T_i) \in \Sigma$ and for every state $s \in I_i$, O is (s, T_i, I_i)-confluent.



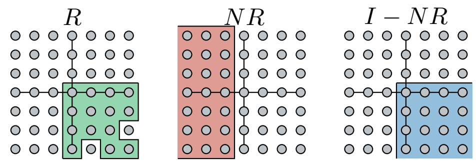
(a) IsInvConfluent determines $I(s_0)$ and then calls Helper with $R = \{s_0\}$, $NR = \emptyset$, and $I = \{(x, y) \mid xy \le 0\}$.



(b) Helper determines that O is not $(I - NR)$-closed with counterexample $s_1 = (-1, 1)$ and $s_2 = (1, -1)$. Helper randomly generates some (s_0, T, I)-reachable points and adds them to R. Luckily for us, $s_2 \in R$, so Helper knows that it is (s_0, T, I)-reachable. Helper is not sure about s_1, so it asks the user. After some thought, the user tells Helper that s_1 is (s_0, T, I)-unreachable, so Helper adds s_1 to NR and then recurses.



(c) Helper determines that O is not $(I - NR)$-closed with counterexample $s_1 = (-1, 2)$ and $s_2 = (3, -3)$. Helper randomly generates some (s_0, T, I)-reachable points and adds them to R. $s_1, s_2 \notin R, NR$, so Helper ask the user to label them. The user puts s_1 in NR and s_2 in R. Then, Helper recurses.



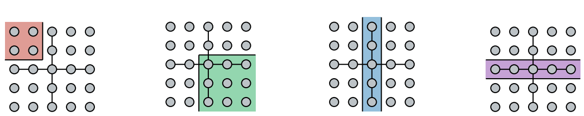
(d) Helper determines that O is not $(I - NR)$-closed with counterexample $s_1 = (-2, 1)$ and $s_2 = (1, -1)$. Helper randomly generates some (s_0, T, I)-reachable points and adds them to R. $s_2 \in R$ but $s_1 \notin R, NR$, so Helper asks the user to label s_1. The user notices a pattern in R and NR and after some thought, concludes that every point with negative x-coordinate is (s_0, T, I)-unreachable. They update NR to $-\mathbb{Z} \times \mathbb{Z}$. Then, Helper recurses. Helper determines that O is $(I - NR)$-closed and returns true!

Figure 3: An example of a user interacting with Algorithm 1 on Example 2. Each step of the visualization shows reachable states R (left), non-reachable states NR (middle), and the refined invariant $I - NR$ (right) as the algorithm executes.

Example 3. Consider again the object $O = (\mathbb{Z} \times \mathbb{Z}, \sqcup)$, transactions $T = \{t_{x+1}, t_{y-1}\}$, and invariant $I = \{(x, y) \mid xy \le 0\}$ from Example 2, but now let the start state s_0 be $(-42, 42)$. O is *not* (s_0, T, I)-confluent because the points $(0, 42)$ and $(42, 0)$ are reachable, and merging these points yields $(42, 42)$ which violates the invariant. However, O is (s_0, T, I, Σ)-confluent for $\Sigma = (I_1, T_1)$, (I_2, T_2), (I_3, T_3), (I_4, T_4) where

$$
\begin{aligned}
I_1 &= \{(x, y) \mid x < 0, y > 0\} & T_1 &= \{t_{x+1}, t_{y-1}\} \\
I_2 &= \{(x, y) \mid x \ge 0, y \le 0\} & T_2 &= \{t_{x+1}, t_{y-1}\} \\
I_3 &= \{(x, y) \mid x = 0\} & T_3 &= \{t_{y-1}\} \\
I_4 &= \{(x, y) \mid y = 0\} & T_4 &= \{t_{x+1}\}
\end{aligned}
$$

Σ is illustrated in Figure 4. Clearly, s_0 satisfies the invariant, and I_1, I_2, I_3, I_4 cover I. Moreover, for every $(I_i, T_i) \in \Sigma$, we see that O is I_i-closed, so O is (s, T_i, I_I)-confluent for every $s \in I_i$. Thus, O is (s_0, T, I, Σ)-confluent.



(a) (I_1, T_1). **(b)** (I_2, T_2). **(c)** (I_3, T_3). **(d)** (I_4, T_4).

Figure 4: An illustration of Example 3

5.2 System Model

Now, we describe the system model used to replicate a segmented invariant confluent object without any coordination within a segment and with only a small amount of coordination when transitioning between segments. As before, we replicate an object O across a set $p_1, \ldots, p_n$ of n servers each of which manages a replica $s_i \in S$ of the object. Every server begins with s_0, T, I, and Σ. Moreover, at any given point in time, a server designates one of the segments in Σ as its **active segment**. Initially, every server chooses the first segment $(I_i, T_i) \in \Sigma$ such that $I_i(s_0)$ to be its active segment. We'll see momentarily the significance of the active segment.

As before, servers repeatedly perform one of two actions: execute a transaction or merge states with another server. Merging states in the segmented invariant confluence system model is identical to merging states in the invariant confluence system model. A server p_i sends its state s_i to another server p_j which *must* merge s_i into its state s_j. Transaction execution in the new system model, on the other hand, is more involved. Consider a server s_i with active segment (I_i, T_i). A client can request that p_i execute a transaction t. We consider what happens when $t \in T_i$ and when $t \notin T_i$.

If $t \notin T_i$, then p_i initiates a round of global coordination to execute t. During global coordination, every server temporarily stops processing transactions and transitions to state $s = s_1 \sqcup \ldots \sqcup s_n$, the join of every server's state. Then, every server speculatively executes t transitioning to state $t(s)$. If $t(s)$ violates the invariant (i.e. $\neg I(t(s))$), then every server aborts t and reverts to state s. Then, p_i replies to the client. If $t(s)$ satisfies the invariant (i.e. $I(t(s))$), then every server commits t and remains in state $t(s)$. Every server then chooses the first segment $(I_i, T_i) \in \Sigma$ such that $I_i(t(s))$ to be the new active segment. Note that such a segment is guaranteed to exist because the segment invariants cover I. Moreover, Σ is ordered, as described in the previous subsection, so every server will deterministically pick the same active segment. In fact, an invariant of the system model is that at any given point of normal processing, every server has the same active segment.

Otherwise, if $t \in T_i$, then p_i executes t immediately and without coordination. If $t(s_i)$ satisfies the *active* invariant (i.e. $I_i(t(s_i))$), then p_i commits t, stays in state $t(s_i)$, and replies to the client. If $t(s_i)$ violates the *global* invariant (i.e. $\neg I(t(s_i))$), then p_i aborts t, reverts to state s_i, and replies to the client. If $t(s_i)$ satisfies the global invariant but violates the active invariant (i.e. $I(t(s_i))$ but $\neg I_i(t(s_i))$), then p_i reverts to state s_i and initiates a round of global coordination to execute t, as described in the previous paragraph.

This system model guarantees that all replicas of a segmented invariant confluent object always satisfy the invariant. All servers begin in the same initial state and with the same active segment. Thus, because O is invariant confluent with respect to every segment, servers can execute transactions within the active segment without any coordination and guarantee that the invariant is never violated. Any operation that would violate the assumptions of the invariant confluence system model (e.g. executing a transaction that's not permitted in the active segment or executing a permitted transaction that leads to a state outside the active segment) triggers a global coordination. Globally coordinated transactions are only executed if they maintain the invariant. Moreover, if a globally coordinated transaction leads to another segment, the coordination ensures that all servers begin in the same start state and with the same active segment, reestablishing the assumptions of the invariant confluence system model.

5.3 Interactive Decision Procedure

In order for us to determine whether or not an object O is (s_0, T, I, Σ)-confluent, we have to determine whether or not O is invariant confluent within each segment $(I_i, T_i) \in \Sigma$. That is, we have determine if O is (s, T_i, I_i)-confluent confluent for every state $s \in I_i$. Ideally, we could leverage Algorithm 1, invoking it once per segment. Unfortunately, Algorithm 1 can only be used to determine if O is (s, T_i, I_i)-confluent for a *particular* state $s \in I_i$, not for *every* state $s \in I_i$. Thus, we would have to invoke Algorithm 1 $|I_i|$ times for every segment (I_i, T_i), which is clearly infeasible given that I_i can be large or even infinite.

Instead, we develop a new decision procedure that can be used to determine if an object is (s, T, I)-confluent for every state $s \in I$. To do so, we need to extend the notion of reachability to a notion of coreachability and then generalize Theorem 2. Two states $s_1, s_2 \in I$ are **coreachable** with respect to a set of transactions T and an invariant I, abbreviated (T, I)**-coreachable**, if there exists some state $s_0 \in I$ such that s_1 and s_2 are both (s_0, T, I)-reachable.

Theorem 3. *Consider an object $O = (S, \sqcup)$, a set of transactions T, and an invariant I. If every pair of states in the invariant are (T, I)-coreachable, then*

$$O \text{ is } I\text{-closed} \iff O \text{ is } (s, T, I)\text{-confluent for every } s \in I$$

The proof of the forward direction is exactly the same as the proof of Theorem 1. Transactions always maintain the invariant, so if merge does as well, then every reachable state must satisfy the invariant. For the reverse direction, consider two arbitrary states $s_1, s_2 \in I$. The two points are (T, I)-coreachable, so there exists some state s_0 from which they can be reached. O is (s_0, T, I)-confluent and $s_1 \sqcup s_2$ is (s_0, T, I)-reachable, so it satisfies the invariant.

Algorithm 2 Interactive invariant confluence decision procedure for arbitrary start state $s \in I$

```
// Return if O is (s, T, I)-confluent for every s ∈ I.
function IsInvConfluent(O, T, I)
    return Helper(O, T, I, ∅, ∅)

// R is a set of (T, I)-coreachable states.
// NR is a set of (T, I)-counreachable states.
function Helper(O, T, I, R, NR)
    closed, s1, s2 ← IsIClosed(O, I, NR)
    if closed then return true
    Augment R, NR with random search and user input
    if (s1, s2) ∈ R then return false
    return Helper(O, T, I, R, NR)
```

Using Theorem 3, we develop Algorithm 2: a natural generalization of Algorithm 1. Algorithm 1 iteratively refines the set of *reachable* states whereas Algorithm 2 iteratively refines the set of *coreachable* states, but otherwise, the core of the two algorithms is the same. Now, a segmented invariant confluence decision procedure, can simply invoke Algorithm 2 once on each segment.

Example 4. Let $O = (\mathbb{Z}^3 \times \mathbb{Z}^3, \sqcup)$ be an object that separately keeps positive and negative integer counts (dubbed a PN-Counter [23]), replicated on three machines. Every state $s = (p_1, p_2, p_3), (n_1, n_2, n_3)$ represents the integer $(p_1 + p_2 + p_3) - (n_1 + n_2 + n_3)$. To increment or decrement the counter, the ith server increments p_i or n_i respectively, and $\sqcup$ computes an element-wise maximum. Our start state $s_0 = (0,0,0), (0,0,0)$; our set T of transactions consists of increment and decrement; and our invariant I is that the value of s is non-negative.

Applying Algorithm 1, IsIClosed returns false with the states $s_1 = (1,0,0), (0,1,0)$ and $s_2 = (1,0,0), (0,0,1)$. Both are reachable, so O is not (s_0, T, I)-confluent and Algorithm 1 returns false. The culprit is concurrent decrements, which we can forbid in a simple one-segment segmentation $\Sigma = (I, T^+)$ where T^+ consists only of increment transactions. Applying Algorithm 2, IsIClosed again returns false with the same states s_1 and s_2. This time, however, the user recognizes that the two states are not (T^+, I)-coreachable. The user refines NR with the observation that two states are coreachable if and only if they have the same values of n_1, n_2, n_3. After this, IsIClosed (and thus Helper) returns true, and Algorithm 2 terminates.

6. EVALUATION

In this section, we describe and evaluate Lucy: a prototype implementation of our decision procedures and system models. A more complete evaluation can be found in [25]. Lucy includes a Python implementation of the interactive decision procedures described in Algorithm 1 and Algorithm 2. Users specify objects, transactions, invariants, and segmentations in Python. Lucy also includes a C++

implementation of the invariant confluence and segmented invariant confluence system models.

We now evaluate the practicality and efficiency of our decision procedure prototypes. Specifically, we show that specifying objects is not too onerous and that our decision procedures' latencies are small enough to be used comfortably in an interactive way [18].

Example 5 (Foreign Keys)**.** A 2P-Set $X = (A_X, R_X)$ is a set CRDT composed of a set of additions A_X and a set of removals R_X [23]. We view the state of the set X as the difference $A_X - R_X$ of the addition and removal sets. To add an element x to the set, we add x to A_X. Similarly, to remove x from the set, we add it to R_X. The merge of two 2P-sets is a pairwise union (i.e. $(A_X, R_X) \sqcup (A_Y, R_Y) = (A_X \cup A_Y, R_X \cup R_Y)$).

We can use 2P-sets to model a simple relational database with foreign key constraints. Let object $O = (X, Y) = ((A_X, R_X), (A_Y, R_Y))$ consist of a pair of two 2P-Sets X and Y, which we view as relations. Our invariant $X \subseteq Y$ (i.e. $(A_X - R_X) \subseteq (A_Y - R_Y)$) models a foreign key constraint from X to Y. We ran our decision procedure on the object with initial state $((\emptyset, \emptyset), (\emptyset, \emptyset))$ and with transactions that allow arbitrary insertions and deletions into X and Y. After less than a tenth of a second, the decision procedure produced a reachable counterexample witnessing the fact that the object is not invariant confluent. A concurrent insertion into X and deletion from Y can lead to a state that violates the invariant. This object is not invariant confluent and therefore not invariant closed. Thus, existing systems that depend on invariant closure as a sufficient condition are unable to conclude definitively that the object is *not* invariant confluent.

We also reran the decision procedure, but this time with insertions into X and deletions from Y disallowed. In less than a tenth of a second, the decision procedure correctly deduced that the object is now invariant confluent. These results were manually proven in [3], but our tool was able to confirm them automatically in a negligible amount of time.

Example 6 (Escrow Transactions)**.** Escrow transactions are a concurrency control technique that allows a database to execute transactions that increment and decrement numeric values with more concurrency than is otherwise possible with general-purpose techniques like two-phase locking [21]. The main idea is that a portion of the numeric value is put in escrow, after which a transaction can freely decrement the value so long as it is not decremented by more than the amount that has been escrowed. We show how segmented invariant confluence can be used to implement escrow transactions.

Consider again the PN-Counter $s = (p_1, p_2, p_3), (n_1, n_2, n_3)$ from Example 4 replicated on three servers with transactions to increment and decrement the PN-Counter. In Example 4, we found that concurrent decrements violate invariant confluence which led us to a segmentation which prohibited concurrent decrements. We now propose a new segmentation with escrow amount k that will allow us to perform concurrent decrements that respect the escrowed value. The first segment $(\{(p_1, p_2, p_3), (n_1, n_2, n_3) \mid p_1, p_2, p_3 \geq k \wedge n_1, n_2, n_3 \leq k\}, T)$ allows for concurrent increments and decrements so long as every $p_i \geq k$ and every $n_i \leq k$. Intuitively, this segment represents the situation in which every server has escrowed a value of k. Each server can decrement freely, so long as they don't exceed their escrow budget of k. The second segment is the one presented in Example 4 which prohibits concurrent decrements. We ran our decision procedure on this example and it concluded that it was segmented invariant confluent in less than a tenth of a second and without any human interaction.

Further Decision Procedure Evaluation. In [25], we also specify workloads involving Example 1, an auction application, and TPC-C. Lucy processes all of these workloads, as well as the workloads described above, in less than half a second, and no workload requires more than 66 lines of Python code to specify. This shows that the user burden of specification is not too high and that our decision procedures are efficient enough for interactive use.

System Model Evaluation. In addition to our decision procedures, we also evaluate the performance of distributed objects deployed with segmented invariant confluence [25]. Namely, we show that segmented invariant confluent replication can achieve an order of magnitude higher throughput compared to linearizable replication, but the throughput improvements decrease as we increase the fraction of transactions that require coordination. For example, with 5% coordinating transactions, segmented invariant confluent replication performs over an order of magnitude better than linearizable replication; with 50%, it performs as well; and with 100%, it performs two times worse.

7. RELATED WORK

RedBlue consistency [16], is a consistency model that sits between causal consistency and linearizability. In [16], Li et al. introduce invariant safety as a sufficient (but not necessary) condition for RedBlue consistent objects to be invariant confluent. Invariant safety is an analog of invariant closure. In [15], Li et al. develop sophisticated techniques for deciding invariant safety that involve calculating weakest preconditions. These techniques are complementary to our work and can be used to improve the invariant closure subroutine used by our decision procedures.

The homeostasis protocol [22], a generalization of the demarcation protocol [6], uses program analysis to avoid unnecessary coordination between servers in a *sharded* database (whereas invariant confluence targets *replicated* databases).

Explicit consistency [5] is a consistency model that combines invariant confluence and causal consistency, similar to RedBlue consistency with invariant safety. Balegas et al. also describe a variety of techniques—like conflict resolution, locking, and escrow transactions [21]—that can be used to replicate workloads that do not meet their sufficient conditions. Segmented invariant confluence is a formalism that can be used to model simple forms of these techniques.

In [10], Gotsman et al. discuss a hybrid token based consistency model that generalizes a family of consistency models including causal consistency, sequential consistency, and RedBlue consistency. The token based approach allows users to specify certain conflicts that are not possible with segmented invariant confluence. However, segmented invariant confluence also introduces the notion of invariant segmentation, which cannot be emulated with the token based approach. For example, it is difficult to emulate escrow transactions with the token based approach.

8. CONCLUSION

This paper revolved around two major contributions. First, we found that invariant closure fails to incorporate a notion of reachability, and using this intuition, we developed conditions under which invariant closure and invariant confluence are equivalent. We implemented this insight in an interactive invariant confluence decision procedure that automatically checks whether an object is invariant confluent, with the assistance of a programmer. Second, we proposed a generalization of invariant confluence, segmented invariant confluence, that can be used to replicate non-invariant confluent objects with a small amount of coordination while still preserving their invariants.

9. ACKNOWLEDGMENTS

A big thanks to Alan Fekete, Peter Alvaro, Alvin Cheung, Alexandra Meliou, Anthony Tan, and Cristina Teodoropol. This research is supported in part by DHS Award HSHQDC-16-3-00083, NSF CISE Expeditions Award CCF-1139158, and gifts from Alibaba, Amazon Web Services, Ant Financial, CapitalOne, Ericsson, GE, Google, Huawei, Intel, IBM, Microsoft, Scotiabank, Splunk and VMware.

10. REFERENCES

[1] D. Abadi. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. *Computer*, 45(2):37–42, 2012.

[2] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.

[3] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Coordination avoidance in database systems. *PVLDB*, 8(3):185–196, 2014.

[4] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. Preguiça, M. Najafzadeh, and M. Shapiro. Putting consistency back into eventual consistency. In *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2015.

[5] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. Preguiça, M. Najafzadeh, and M. Shapiro. Towards fast invariant preservation in geo-replicated systems. *ACM SIGOPS Operating Systems Review*, 49(1):121–125, 2015.

[6] D. Barbará-Millá and H. Garcia-Molina. The demarcation protocol: A technique for maintaining constraints in distributed database systems. *The VLDB Journal*, 3(3):325–353, 1994.

[7] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)*, 13(2):185–221, 1981.

[8] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[9] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002.

[10] A. Gotsman, H. Yang, C. Ferreira, M. Najafzadeh, and M. Shapiro. 'cause i'm strong enough: Reasoning about consistency choices in distributed systems. *ACM SIGPLAN Notices*, 51(1):371–384, 2016.

[11] P. W. Grefen and P. M. Apers. Integrity control in relational database systemsâĂŤan overview. *Data & Knowledge Engineering*, 10(2):187–223, 1993.

[12] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[13] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[14] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[15] C. Li, J. Leitão, A. Clement, N. Preguiça, R. Rodrigues, and V. Vafeiadis. Automating the choice of consistency levels in replicated systems. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 281–292, 2014.

[16] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 265–278, 2012.

[17] R. J. Lipton and J. S. Sandberg. Pram: A scalable shared memory. Technical Report TR-180-88, Computer Science Department, Princeton University, August 1988.

[18] Z. Liu and J. Heer. The effects of interactive latency on exploratory visual analysis. *IEEE transactions on visualization and computer graphics*, 20(12):2122–2131, 2014.

[19] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 401–416. ACM, 2011.

[20] S. A. Mehdi, C. Littley, N. Crooks, L. Alvisi, N. Bronson, and W. Lloyd. I can't believe it's not causal! scalable causal consistency with no slowdown cascades. In *NSDI*, pages 453–468, 2017.

[21] P. E. O'Neil. The escrow transactional method. *ACM Transactions on Database Systems (TODS)*, 11(4):405–430, 1986.

[22] S. Roy, L. Kot, G. Bender, B. Ding, H. Hojjat, C. Koch, N. Foster, and J. Gehrke. The homeostasis protocol: Avoiding transaction coordination through program analysis. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1311–1326. ACM, 2015.

[23] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. *A comprehensive study of convergent and commutative replicated data types*. PhD thesis, Inria–Centre Paris-Rocquencourt; INRIA, 2011.

[24] W. Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[25] M. Whittaker and J. M. Hellerstein. Interactive checks for coordination avoidance. *Proceedings of the VLDB Endowment*, 12(1):14–27, 2018.