Interactive Checks for Coordination Avoidance

Technical Report. September 24, 2018.

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) a general-purpose interactive invariant confluence decision procedure and (b) a novel sufficient condition that can be checked automatically. 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 implemented these formalisms in a prototype called Lucy and found that our decision procedures efficiently handle common real-world workloads including foreign keys, rollups, escrow transactions, and more. We also found that segmented invariant confluent replication can deliver up to an order of magnitude more throughput than linearizable replication for low contention workloads and comparable throughput for medium to high contention workloads.

This version of the paper is a technical report that includes some additional information that is not present in our main publication. Additional text (like this) is annotated with a red bar along its left side.

PVLDB Reference Format:
Michael Whittaker and Joseph M. Hellerstein. Interactive Checks for Coordination Avoidance. *PVLDB*, 12(1): 14-27, 2018.
DOI: https://doi.org/10.14778/3275536.3275538

Proceedings of the VLDB Endowment, Vol. 12, No. 1
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3275536.3275538

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 (e.g., [12, 35]) or state machine replication (e.g., [41, 28, 32, 38]) 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 [25]. 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 [20, 13, 1]. On the other hand, weak consistency models like eventual consistency [46], PRAM consistency [31], causal consistency [2], and others [33, 34] 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 [26, 10] such as database integrity constraints [22, 23]. 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 [8]. 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 can 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 [30, 29]. The existing approaches check whether an object is invariant 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 and a new sufficient condition for invariant confluence, dubbed *merge reducibility*. Merge reducibility covers some cases that are not covered by invariant closure, and it can be checked automatically.

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, and vice-versa.

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 7, we apply Lucy to foreign key constraints, escrow transactions, an auction application, and the TPC-C benchmark. Lucy processes these workloads in less than half a second, and no workload requires more than 66 lines of code to specify. Moreover, we find that segmented invariant confluent replication can achieve 10x to 100x more throughput than linearizable replication for low-coordination workloads.

In closing, here is an outline of the paper and of our contributions: We propose a novel expression-oriented definition of invariant confluence that is both formal and simple (Section 2). We develop an understanding of why invariant closure is not necessary for invariant confluence and use this understanding to develop conditions under which it is both necessary and sufficient (Section 3). We exploit these conditions to design a general-purpose interactive decision procedure for invariant confluence (Section 4). We again exploit these conditions to design a novel non-trivial sufficient condition for invariant confluence, called merge reducibility. We present segmented invariant confluence: a generalization of invariant confluence that uses a small amount of coordination to maintain invariants for replicated objects that are otherwise not invariant confluent (Section 6). We evaluate our methods using a prototype implementation called Lucy (Section 7).

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 [8]. 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 accompanying invariant is replicated across a set of servers. Clients send transactions to servers, and a server executes a transaction so long as it maintains the invariant. Servers execute transactions without coordination, but to avoid state divergence, servers periodically gossip with one another and merge their replicas. After we introduce the system model, we present a formal definition of invariant confluence.

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 $\{x \in \mathbb{Z} \,|\, x \geq 0\}$ is the invariant that states x are non-negative.

Note that by modelling a transaction t as a function $S \to S$, we focus exclusively on the effects that a transaction has on the object (i.e. "writes" to the object). Transactions are also free to read the value of the object, but these reads are not captured by our model because, as we'll see, they do not affect invariant confluence. For example, we could model any read-only transaction as a function t where $t(s) = s$ for every $s \in S$.

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 replicated 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[1]. 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.

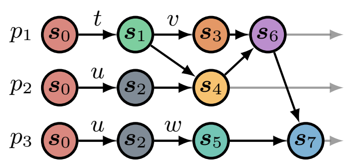
Informally, O is **invariant confluent** with respect to s_0, T, and I, abbreviated (s_0, T, I)**-confluent**, if every replica $s_1, \ldots, s_n$ is guaranteed to always satisfy the invariant I in every possible execution of the system.

2.2 Expression-Based Formalism

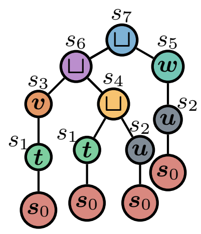
To define invariant confluence formally, we represent a state produced by a system execution as a simple expression generated by the grammar

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

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

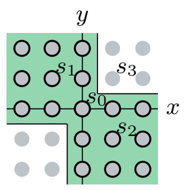
Figure 1: A system execution and corresponding expression

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 smallest predicate that satisfies the following equations:

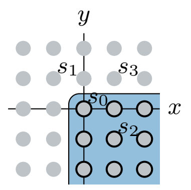
- $\text{reachable}_{(s_0,T,I)}(s_0)$.
- For all expressions e and for all transactions t in the set T of transactions, if $\text{reachable}_{(s_0,T,I)}(e)$ and $I(t(e))$, then $\text{reachable}_{(s_0,T,I)}(t(e))$.
- For all expressions e_1 and e_2, if $\text{reachable}_{(s_0,T,I)}(e_1)$ and $\text{reachable}_{(s_0,T,I)}(e_2)$,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

[1]Notably, if O is a CRDT—i.e. O is a semilattice and every transaction $t \in T$ is inflationary—then this periodic merging ensures that O is strongly eventually consistent [43].



(a) Invariant



(b) Reachable points

Figure 2: An illustration of Example 2

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 [30, 29]. 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, recall that 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 [17, 9, 21]. 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.

Example 2. Let $O = (\mathbb{Z} \times \mathbb{Z}, \sqcup)$ consist of pairs (x, y) of integers where $(x_1, y_1) \sqcup (x_2, y_2) = (\max(x_1, x_2), \max(y_1, y_2))$.

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($O$, $s_0$, $T$, $I$, $\{s_0\}$, $\emptyset$)

// $R$ 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, $s_1$, $s_2$ $\leftarrow$ IsIclosed($O$, $I - NR$)
    if closed then return true
    Augment $R, NR$ with random search and user input
    if $s_1, s_2 \in R$ then return false
    return Helper($O$, $s_0$, $T$, $I$, $R$, $NR$)
```

Our start state $s_0 \in \mathbb{Z} \times \mathbb{Z}$ is the point $(0, 0)$. 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 $I = \{(x, y) \in \mathbb{Z} \times \mathbb{Z} \,|\, xy \leq 0\}$ 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. $I \subseteq \{s \in S \,|\, reachable_{(s_0,T,I)}(s)\}$), 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.

4.1 The Decision Procedure

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 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. If closed is false, then 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 [17].

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

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

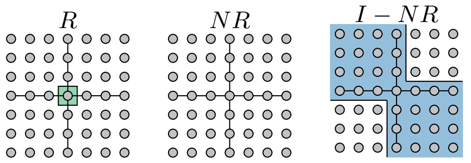
$$\{s \in S \,|\, \text{reachable}_{(s_0,T,I-NR)}(s)\} \subseteq I - NR \subseteq I$$

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.

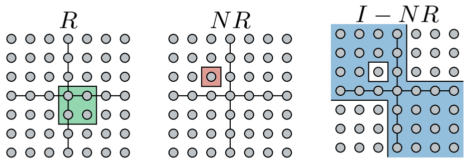
If IsIclosed determines that O is *not* $(I - NR)$-closed, then we have a counterexample s_1, s_2. We want to determine whether s_1 and s_2 are reachable or unreachable. We can do so in two ways. First, we can randomly generate a set of reachable states and add them to R. If s_1 or s_2 is in R, then we know they are reachable. Second, we can prompt the user to tell us directly whether or not 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 an example). In this step, we also make sure that $s_0 \notin NR$ since we know that s_0 is reachable.

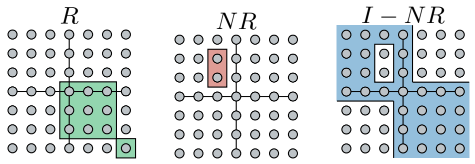
After s_1 and s_2 have been labelled as (s_0, T, I)-reachable or not, we continue. If both s_1 and s_2 are (s_0, T, I)-reachable, then so is $s_1 \sqcup s_2$, but $\neg I(s_1 \sqcup s_2)$. Thus, O is not (s_0, T, I)-confluent, so Helper returns false. Otherwise, one of s_1 and s_2 is (s_0, T, I)-unreachable,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



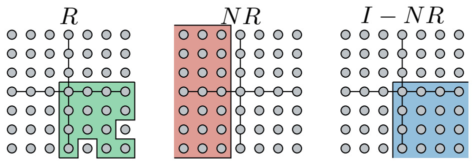
(a) IsInvConfluent determines $I(s_0)$ and then calls Helper with $R = \{s_0\}$, $NR = \emptyset$, and $I = \{(x, y) \mid xy \leq 0\}$.



(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.

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. 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.

4.2 Limitations

Our interactive invariant confluence decision procedure has two limitations. First, Algorithm 1 requires an invariant closure decision procedure, but determining invariant closure is undecidable in general.

For example, let $O_p = (S, \sqcup)$ where S is the set of all programs and $s_1 \sqcup s_2 = p$ for some fixed program p. Letting I be the set of all programs that terminate, determining if O_p is I-closed is tantamount to determining if p terminates.

In practice, we can implement an invariant closure decision procedure using an SMT solver like Z3 that works well on simple objects, invariants, and merge operators (e.g., integers, tuples, infinite sets, bit vectors, linear constraints, basic arithmetic, tuple projection, basic set operations, bit arithmetic). But, SMT solvers are mostly unable to analyze more complex constructs (e.g., finite lists [27], graphs, recursive algebraic data types, nonlinear constraints, merge operators that contain loops or recursion).

Second, Algorithm 1 relies on a user to identify unreachable states. As we saw in Figure 3, the set of unreachable states can sometimes be clear, especially if there's a noticeable pattern in the set of reachable states. However, if the set of transactions is large or complex or if the merge operator is complex, then reasoning about unreachable states can be difficult. Unlike with reachable states—where verifying that a state is reachable only requires thinking of a single way to reach the state—verifying that a state is unreachable requires a programmer to reason about a large number of system executions and conclude that *none* of them can lead to the state. In the future, we plan on exploring ways to help a user reason about unreachable states and ways to discover sets of unreachable states automatically.

4.3 Tolerating User Error

Algorithm 1 is an *interactive* decision procedure. It requires that a user classify states as reachable or unreachable. To err is human, so what happens when a user incorrectly classifies a state? There are two possible errors that can be made, and Algorithm 1 can be made robust to both.

A user can label an unreachable state as reachable. In this case, Helper might erroneously find s_1 and $s_2 \in R$ and return false, concluding that O is not (s_0, T, I)-confluent even when it is. This is inconvenient, but not catastrophic. We can modify Algorithm 1 so that Helper requires that whenever a user labels a state s as (s_0, T, I)-reachable, they must also provide an (s_0, T, I)-reachable expression e that evaluates to s. Here, e acts a proof that certifies that s is indeed reachable. This increases the burden on the user but completely eliminates this type of user error.

A user can label a reachable state as unreachable. In this case, IsIclosed$(O, I - NR)$ might return true, even though O is not (s_0, T, I)-confluent. Thus, a user might falsely believe their distributed object to be (s_0, T, I)-confluent even though it isn't, and eventually one replica of their distributed object might enter a state that violates the invariant. We can mitigate this in two ways. First, we can aggressively expand R automatically. If a user ever labels a state s as unreachable, but $s \in R$, we can notify the user of their mistake. Second, HELPER returns true when O is $(I - NR)$-closed for some NR, so O *is* $(s_0, T, I - NR)$-confluent (even though it might not be (s_0, T, I)-confluent). Thus, when a user replicates their distributed object across a set of servers, they can deploy with the invariant $I - NR$ instead of I. If NR is correct and only contains unreachable states, then deploying with $I - NR$ is equivalent to deploying with I. If NR is incorrect and contains some (s_0, T, I)-reachable states, then some of these states are artificially made unreachable, but the system is still guaranteed to never produce a state that violates I.

5. MERGE REDUCTION

In Section 3, we discussed how invariant confluence is fundamentally a property of reachability and that invariant closure is sufficient but not necessary for invariant confluence because it fails to incorporate any notion of reachability. Using this intuition, we established Theorem 2 and then exploited the theorem in Algorithm 1. In this section, we again take advantage of this intuition to develop a new sufficient condition for invariant confluence that can be checked without user interaction and that covers some cases not covered by invariant closure.

An expression $e = t_1(t_2(\ldots(t_n(s))\ldots))$ is **merge-free** if does not contain any merges (i.e. it is generated by the grammar $e ::= s \mid t(e)$). An object $O = (S, \sqcup)$ is **merge-reducible** with respect to a start state $s_0 \in S$, a set of transactions T, and an invariant I, abbreviated (s_0, T, I)-**merge reducible**, if for every pair e_1 and e_2 of merge-free (s_0, T, I)-reachable expressions, there exists some merge-free (s_0, T, I)-reachable expression e_3 that evaluates to the same state as $e_1 \sqcup e_2$. Intuitively, if O is merge-reducible, we can replace $e_1 \sqcup e_2$ (which has one merge) with e_3 (which has no merges) to obtain an equivalent expression with fewer merges.

Theorem 3. *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 (s_0, T, I)-merge reducible, then O is (s_0, T, I)-confluent.*

The proof of Theorem 3 is a straightforward induction. We begin with an (s_0, T, I)-reachable expression e and repeatedly replace any subexpression that merges two merge-free subexpressions with an equivalent merge-free reachable subexpression (which we can do because O is merge-reducible). We repeat this process until e has been completely replaced with an equivalent merge-free reachable expression e'. Because $I(s_0)$ and because our system model only executes transactions that preserve the invariant, e' (and hence e) is guaranteed to satisfy the invariant. Thus, all reachable states satisfy the invariant, so O is invariant confluent.

Merge-reducibility is a sufficient condition for invariant confluence, but unlike with invariant closure, it is not straightforward to automatically determine if an object is merge-reducible. In Theorem 4, we outline a sufficient condition for merge-reducibility that is straightforward to determine automatically.

Theorem 4. *Given an object $O = (S, \sqcup)$, a start state $s_0 \in S$, a set of transactions T, and an invariant I, if the following criteria are met, then O is (s_0, T, I)-merge reducible (and therefore (s_0, T, I)-confluent).*

1. *O is a join-semilattice.*
2. *For every $t \in T$, there exists some $s_t \in S$ such that for all $s \in S$, $t(s) = s \sqcup s_t$. That is, every transaction t is of the form $t(s) = s \sqcup s_t$ for some constant s_t.*
3. *For every pair of transactions $t_1, t_2 \in T$ and for all states $s \in S$, if $I(s)$, $I(t_1(s))$, and $I(t_2(s))$, then $I(t_1(s) \sqcup t_2(s))$.*
4. *$I(s_0)$.*

To show that O is (s_0, T, I)-merge reducible, we consider two merge-free reachable expressions

$$e_1 = t_1(\ldots(t_n(s_0))\ldots) \quad \text{and} \quad e_2 = u_1(\ldots(u_m(s_0))\ldots)$$

Then, we inductively use criterion (3) with criteria (1) and (2) to show that

$$e_3 = t_1(\ldots(t_n(u_1(\ldots(u_m(s_0))\ldots)))\ldots)$$

is reachable and evaluates to $e_1 \sqcup e_2$.

Theorem 1 states that invariant closure is a sufficient condition for invariant confluence, and Theorem 4 states that criteria (1) – (4) are sufficient conditions for invariant confluence. How do these sufficient conditions relate to one another? Clearly, not all invariant closed objects are semilattices, so invariant closure does not imply criteria (1) – (4). Conversely, there are some objects that satisfy criteria (1) – (4) that are not invariant closed. Here's one example.

Example 3. Let $O = (\mathcal{P}(\mathbb{N}), \cup)$ where $\mathcal{P}(\mathbb{N})$ is the power set of the natural numbers. Our start state $s_0 = \{0\}$ is the set of 0. Let $t_Y(X) = X \cup Y$ be the transaction that unions Y with its argument X. Our set $T = \{t_Y \mid Y \subseteq \mathbb{N}\}$ of transactions consists of all possible t_Y. Our invariant I consists of all non-empty sets X that contain only even or only odd elements. That is, $I = \{X \subseteq 2\mathbb{N} \mid X \neq \emptyset\} \cup \{X \subseteq 2\mathbb{N} + 1 \mid X \neq \emptyset\}$.

Criteria (1), (2), (3) and (4) are all satisfied. However, O is not I-closed. Let $s_1 = \{0\}$ and $s_2 = \{1\}$. Then, $I(s_1)$ and $I(s_2)$, but letting $s_3 = s_1 \cup s_2 = \{0, 1\}$, $\neg I(s_3)$.

Here's why criterion (3) is satisfied. If s is an arbitrary state that satisfies I, then it is non-empty and contains, without loss of generality, only even integers. If t_1 and t_2 are arbitrary transactions such that $I(t_1(s))$ and $I(t_2(s))$, then $t_1(s)$ and $t_1(s)$ are also non-empty and contain only even integers. Thus, $t_1(s) \cup t_2(s)$ is clearly non-empty and contains only even integers, so $I(t_1(s) \sqcup t_2(s))$.

Invariant closure is not necessary for invariant confluence because it fails to incorporate any notion of reachability. Criteria (1) – (4) are also unnecessary, but they can be used to prove that some non-invariant closed objects are invariant confluent because the criteria *do* incorporate notions of reachability. In particular, criterion (3) is a slight variant of invariant closure; it also states that invariant satisfying states should be closed under merge. The fundamental difference is that criterion (3) restricts its attention to the merge of two states that are *reachable* from a common ancestor state.

In Example 3, we saw this fundamental difference rear its head. O is not I-closed because the union of an odd-only set with an even-only set is a set with both odd and even integers. However, if we begin in an invariant satisfying state, we cannot reach both an odd-only and even-only set. Criterion (3) is able to recognize this fact and conclude that O is invariant confluent despite it not being invariant closed.

6. 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. In Section 7, we see that replicating a non-invariant confluent object with segmented invariant confluence can achieve between 10x and 100x more throughput than linearizable replication for certain workloads.

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.

6.1 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 (not a set) 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. 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.

Example 4. Consider again the object $O = (\mathbb{Z} \times \mathbb{Z}, \sqcup)$, transactions $T = \{t_{x+1}, t_{y-1}\}$, and invariant $I = \{(x, y) \mid xy \leq 0\}$ 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

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

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.

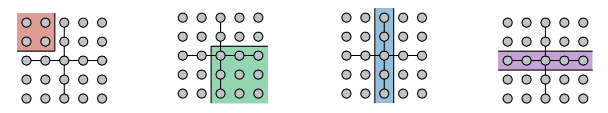

(a) (I_1, T_1). (b) (I_2, T_2). (c) (I_3, T_3). (d) (I_4, T_4).

Figure 4: An illustration of Example 4

6.2 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 a bit more involved. Consider a server s_i with active segment (I_i, T_i). A client can request that p_i execute a transaction $t \in T$. We consider what happens when $t \in T_i$ and $t \notin T_i$ separately.

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, 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. Transaction execution is summarized in Algorithm 2.

Algorithm 2 Transaction execution in the segmented invariant confluence system model

```
if $t \notin T_i$ then
    Execute $t$ with global coordination
else
    if $I_i(t(s_i))$ then Commit $t$
    else if $\neg I(t(s_i))$ then Abort $t$
    else Execute $t$ with global coordination
```

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.

6.3 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 T and 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 5. *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 3 Interactive invariant confluence decision procedure for arbitrary start state $s \in I$

```
// Return if $O$ is $(s, T, I)$-confluent for every $s \in I$.
function IsInvConfluent($O$, $T$, $I$)
    return Helper($O$, $T$, $I$, $\emptyset$, $\emptyset$)

// $R$ 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, $s_1$, $s_2 \leftarrow$ IsIclosed($O$, $I$, $NR$)
    if closed then return true
    Augment $R, NR$ with random search and user input
    if $(s_1, s_2) \in R$ then return false
    return Helper($O$, $T$, $I$, $R$, $NR$)
```

Using Theorem 5, we develop Algorithm 3: a natural generalization of Algorithm 1. Algorithm 1 iteratively refines the set of *reachable* states whereas Algorithm 3 iteratively refines the set of *coreachable* states, but otherwise, the core of the two algorithms is the same.[2] Now, a segmented invariant confluence decision procedure, can simply invoke Algorithm 3 once on each segment.

Example 5. 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 [42]), 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 3, 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 (all modifications of (n_1, n_2, n_3) require global coordination, so it is impossible for s_1 and s_2 to differ on these values). 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 Algorithm 3 return true.

6.4 Discussion and Limitations

There are a few things worth noting about segmented invariant confluence, its system model, and its decision procedure. First, invariant confluence is a very coarse-grained property. If an object is invariant confluent, then we can replicate it with no coordination. If it is not invariant confluent, then we have no guarantees. There's no in-between.

[2]Another small difference is that IsIclosed behaves differently in Algorithm 1 and Algorithm 3. In Algorithm 3, IsIclosed returns a triple (closed, s_1, s_2). If closed is false, then $s_1, s_2 \in I$ are two states not in NR such that $I(s_1)$ and $I(s_2)$ but $\neg I(s_1 \sqcup s_2)$. If no such states exist, then closed is true, and s_1 and s_2 are null.

Segmented invariant confluence, on the other hand, is a much more fine-grained property that can be applied to applications with varying degrees of complexity. Segmented invariant confluence provides guarantees to complex applications that require a large number of segments and to simple applications with a smaller number of segments, whereas invariant confluence only provides guarantees to applications that can be segmented into a single segment.

Second, while our segmented invariant confluence decision procedure can help decide whether or not an object is segmented invariant confluent, it cannot currently help construct a segmentation. It is the responsibility of the programmer to think of a segmentation that is amenable to segmented invariant confluence. This can be an onerous process. In the future, we plan to explore ways by which we can automatically suggest segmentations to the application designer to ease this process.

Third, segmented invariant confluence naturally subsumes a distributed locking approach to replicating non-invariant confluent objects. This approach first recognizes which transactions cannot be safely executed concurrently and then requires them to acquire a distributed lock before executing [9, 21]. For example, in a banking application with the invariant that all balances remain non-negative, concurrent deposits are permitted, but concurrent withdrawals can lead to invariant violations. Thus, we require that withdrawals acquire a distributed lock before executing. This example is exactly the same as Example 5 which we handled by simply removing withdrawal transactions from our segmentation's set of transactions.

Fourth, we can integrate a couple of optimizations into our system model to further reduce the amount of coordination it requires. To begin, if a server with state s_i and active segment (I_i, T_i) receives a transaction $t \in I_i$ to execute, and $t(s_i)$ violates the active invariant but not the global invariant, instead of initiating a round of global coordination, p_i can simply buffer t for re-execution at a later time. While this increases the latency required to execute t, it's possible that after other transactions are executed, re-executing t may lead to a state that either satisfies the active invariant or violates the global invariant. In either case, a round of global coordination is avoided. Similarly, servers can buffer transactions that require global coordination, executing an entire batch of these transactions during a single round of global coordination.

Fifth, a segmented invariant confluence decision procedure can also leverage Theorem 4 in addition to Algorithm 3. If an object O meets criteria (1) - (3), then it is (s, T, I)-confluent for every state $s \in I$.

7. EVALUATION

In this section, we describe and evaluate Lucy: a prototype implementation of our decision procedures and system models.

7.1 Implementation

Lucy includes an implementation of the interactive decision procedure described in Algorithm 1, an implementation of a decision procedure that checks criteria (1) - (4) from Theorem 4, and an implementation of the decision procedure described in Algorithm 3. The decision procedures are implemented in roughly 2,500 lines of Python. Programmers specify objects, transactions, and invariants in a small Python DSL and interact with the interactive decision procedures using an interactive Python console. Note that a programmer only has to run the decision procedures offline a single time to check the invariant confluence of their distributed object. The decision procedures do not have to be run online when transactions are being processed.

We use Z3 [17] to implement our invariant closure decision procedure, compiling an object and invariant into a formula that is satisfiable if and only if the object is *not* invariant closed. If the object is invariant closed, then Z3 concludes that the formula is unsatisfiable. Otherwise, if the object is not invariant closed, then Z3 produces a counterexample witnessing the satisfiability of the formula.

Lucy also includes an implementation of the invariant confluence and segmented invariant confluence system models in roughly 3,500 lines of C++. Users specify objects, transactions, invariants, and segmentations in C++. Lucy then replicates the objects using segmented invariant confluence. Clients send every transaction request to a randomly selected server. When a server receives a transaction request, it executes Algorithm 2 to attempt to execute the transaction locally. If the transaction requires global coordination, then the server forwards the transaction request to a predetermined leader. When the leader receives a transaction request, it broadcasts a coordination request to the other servers. When a server receives a coordination request from the leader, it stops processing transactions and sends the leader its state. When the leader receives the states of all other servers, it executes the transaction, and then sends its state to the other servers. When a server receives a new state, it adopts the state, computes its new active segment, and resumes normal processing. After every 100 transactions processed, a server sends a merge request to a randomly selected server.

Lucy can also replicate an object with eventual consistency and with linearizability. With eventual consistency, clients send every transaction request to a randomly selected server. The server executes the transaction locally and returns immediately to the client, sending merge requests after every 100 transactions. With linearizability, clients send every transaction request to a predetermined leader. The leader relays the transaction request to all other servers, and when the leader receives replies from them, it executes the transaction and replies to the client. This communication pattern mimics the "normal operation" of state machine replication protocols [28, 32].

Because fault-tolerance is largely an orthogonal concern to invariant confluence, Lucy is implemented without fault-tolerance. It would be straightforward to add fault-tolerance to Lucy, but it would not affect our discussions or evaluation, so we leave it for future work.

7.2 Decision Procedures

We now evaluate the practicality and efficiency of our decision procedure prototypes. We begin by demonstrating the decision procedure on a handful of simple, yet practical examples. We then discuss how our tool can be used to analyze the TPC-C benchmark. All decision procedures were run on a MacBook Pro laptop with a 3.5 GHz Intel Core i7 processor and 16 GB of RAM. A summary of these results is given in Table 1.

Table 1: Example 6 to Example 10 Summary

Example	Run time (s)	Lines of code
6	0.09	7
7 (all transactions)	0.06	8
7 (limited transactions)	0.09	10
8	0.04	21
9	0.09	49
10 (Invariant 1)	0.46	66
10 (Invariant 2)	0.44	33

Example 6 ($\mathbb{Z}^2$)**.** We begin with a minimal working example. Consider again our recurring example of $\mathbb{Z}^2$ from Example 2. The Python code used to describe the object, transactions, and invariant is given in Figure 5. When we call `checker.check()`, the interactive decision procedure produces a counterexample $s_1 = (0, 1), s_2 = (1, 0)$ in less than a tenth of a second and automatically recognizes that s_2 is reachable. After we label s_1 as unreachable and refine the invariant with $y \leq 0$, the interactive decision procedure determines that the object is invariant confluent, again, in less than a tenth of a second. Note that the object is invariant confluent but *not* invariant closed, so prior work [30, 29, 10, 21] that relies on invariant closure—or another equivalent sufficient condition—to determine invariant confluence would not be able to identify this example as invariant confluent.

```
checker = InteractiveInvariantConfluenceChecker()
x = checker.int_max('x', 0) # An int, x, merged by max.
y = checker.int_max('y', 0) # An int, y, merged by max.
checker.add_transaction('increment_x', [x.assign(x + 1)])
checker.add_transaction('decrement_y', [y.assign(y - 1)])
checker.add_invariant(x * y <= 0)
checker.check()
```

Figure 5: Example 6 specification

Example 7 (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 [42]. 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, previous tools depending on invariant closure as a sufficient condition would be 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 [8], but our tool was able to confirm them automatically in a negligible amount of time.

Example 8 (Auction)**.** We now consider a simple auction system introduced in [21]. Our object consists of a set B of integer-valued bids and a optional winning bid w. Initially, $B = \emptyset$ and $w = \perp$ (indicating that there is no winning bid yet) and we merge states by taking the union of B and the maximum of w (where $\perp < n$ for all integers n). One transaction t_b places a bid b by inserting it into B. Another transaction t_{close} closes the auction and sets w equal to the largest bid in B. Our invariant is that if the auction is closed (i.e. $w \neq \perp$), then $w = \max(B)$. We ran our decision procedure on this example and in a third of a second, it produced a reachable counterexample witnessing the fact that the object is not invariant confluent. If we concurrently close the auction and place a large bid, then we can end up in a state in which the auction is closed, but there is a bid in B larger than w.

We then segmented our object as follows. The first segment $(\{(B, w) \mid w = \perp\}, \{t_b \mid b \in \mathbb{Z}\})$ allows bidding so long as the bid is open. The second segment $(\{B, w \mid w \neq \perp\} \cap I, \emptyset)$ includes all auctions that have already been closed and forbids all transactions. This segmentation captures the intuition that bids should be permitted only when the auction is open. We ran our segmented invariant confluence decision procedure on this example, and it was able to deduce without any human interaction that the example was segmented invariant confluent in less than a tenth of a second.

Example 9 (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 [37]. 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 5 replicated on three servers with transactions to increment and decrement the PN-Counter. In Example 5, 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. They can decrement freely, so long as they don't exceed their escrow budget of k. The second segment is the one presented in Example 5 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.

Example 10 (TPC-C)**.** TPC-C is a ubiquitous OLTP benchmark with a workload that models a simple warehousing application [19]. The TPC-C specification outlines twelve "consistency requirements" (read invariants) that govern the warehousing application. In [8], Bailis et al. categorize the invariants into one of three types:

Three of the twelve invariants involve **foreign key constraints**. As discussed in Example 7, our decision procedures can automatically verify conditions under which foreign key constraints are invariant confluent.

Seven of the twelve invariants involve **maintaining arithmetic relationships between relations**. Our decision procedures can correctly identify these as invariant confluent. Consider, for example, invariant 1 which dictates that a warehouse's year to date balance `W_YTD` is equal to the sum of the district year to date balances `D_YTD` of the twenty districts that are associated with the warehouse. The Payment transaction randomly selects a district and increments `W_YTD` and `D_YTD` by a randomly generated amount. We model this workload with a PN-Counter for `W_YTD` and twenty PN-Counters for the twenty instances of `D_YTD`. We applied Lucy to this workload, and it determined that the workload was invariant confluent in less than a second.

Two of the twelve invariants involve generating **sequential and unique identifiers**. Consider, for example, invariant 2 which dictates that a district's next order ID `D_NEXT_O_ID` is equal to the maximum order id `O_ID` of orders within the district. The New Order transaction places an order with `O_ID` equal to the current value of `D_NEXT_O_ID` and then increments `D_NEXT_O_ID`. We model this workload with an integer for `D_NEXT_O_ID` and a map for `O_ID` that maps order IDs to order. We applied Lucy to this workload and in less than a second, it produced a counterexample that—when labelled as reachable—confirms Bailis et al.'s finding that the workload is *not* invariant confluent [8]. Thus, the TPC-C benchmark requires some form of coordination to ensure unique and sequential IDs. Alternatively, as Bailis et al. describe in [8], the workload can be run coordination free if we drop the requirement that IDs are assigned sequentially.

7.3 Segmented Invariant Confluence

Now, we evaluate the performance of replicating an object with segmented invariant confluence as compared to the performance of replicating it with eventual consistency or linearizability. There are two hypotheses about the performance of segmented invariant confluent replication that we aim to confirm. First, segmented invariant confluent replication provides higher throughput and better scalability than linearizable replication for workloads that require little coordination (i.e. low-coordination workloads). Second, the throughput and scalability of segmented invariant confluent replication decreases as we increase the fraction of transactions that require coordination.

These hypotheses state that segmented invariant confluent replication is more performant than linearizable replication for low-coordination workloads. But by how much? We also aim to measure the absolute performance and scalability benefits of segmented invariant confluent replication and how they vary as we vary the coordination required by a workload. We perform two controlled microbenchmarks to confirm our hypotheses and discover the absolute performance benefits. The workloads themselves are trivial but are not the focus of our experiments. Our objective is to obtain a controlled measure of throughput and scalability as we vary workload contention.

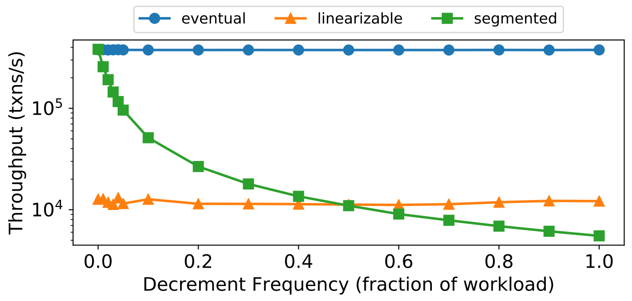


Figure 6: Segmented invariant confluent replication throughput versus coordination induced by executing disallowed decrement transactions.

Benchmark 1. Consider again the PN-Counter from Example 5 and the corresponding transactions, invariants, and single-segment segmentation that forbids concurrent decrements. We replicate this object on 16 servers deployed on 16 m5.xlarge EC2 instances within the same availability zone. Each server has three colocated clients that issue increment and decrement transactions. We replicate the object with eventual consistency, segmented invariant confluence, and linearizability and measure the system's total throughput as we vary the fraction of client requests that are decrements. The results are shown in Figure 6.

Both eventually consistent replication and linearizable replication are unaffected by the workload, achieving roughly 375,000 and 12,000 transactions per second respectively. Segmented invariant confluent replication performs well for low-decrement (i.e. low-coordination) workloads and performs increasingly poorly as we increase the fraction of decrement transactions, eventually performing worse than linearizable replication. For example, with 5% decrement transactions, segmented invariant confluent replication performs over an order of magnitude better than linearizable replication; with 50% decrements, it performs as well; and with 100% decrements, it performs two times worse.

These results offer two insights. First, the relationship between segmented invariant confluent and linearizable replication is analogous to the relationship between optimistic and pessimistic concurrency control protocols. Linearizable replication pessimistically assumes that concurrently executing *any* pair of transactions will lead to an invariant violation. Thus, clients send transactions directly to a leader to be linearized. Conversely, segmented invariant confluent replication optimistically attempts to perform every transaction locally and without coordination. A server only initiates a round of coordination if it is found to be necessary. As a consequence, segmented invariant confluent replication can offer substantial performance benefits over linearizable replication for low-coordination workloads. However, it is inferior for medium to high contention workloads because the majority of transactions that are sent to a server are eventually aborted and relayed to the leader. This additional latency is avoided by linearizable replication which sends transactions directly to the leader.

Second, throughput does not decrease linearly with the amount of coordination. Even infrequent coordination can drastically decrease throughput. Increasing the fraction of

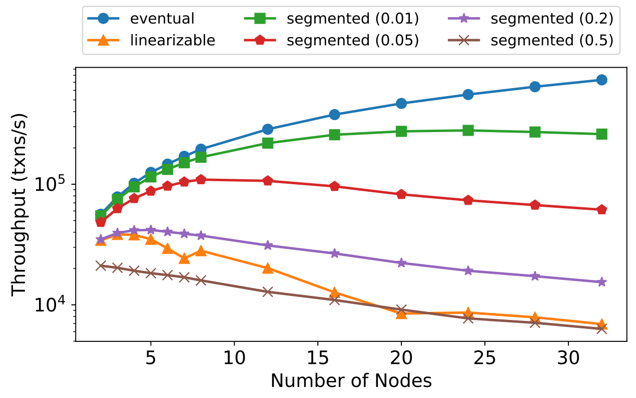


Figure 7: Throughput of eventually consistent, segmented invariant confluent, and linearizable replication measured against the number of nodes for workloads with varying fractions of decrement transactions. For example, the "segmented (0.2)" line measures the performance of segmented invariant confluent replication with 20% decrement transactions. Eventually consistent replication and linearizable replication are not affected by workload.

decrements from 0% to 1% decreases throughput by a factor of 2. Increasing again to 3%, the throughput decreases by another factor of 2. With 90% coordination-free transactions (i.e. 10% decrements), we achieve only 10% of the throughput of eventually consistent replication.

Benchmark 2. In this benchmark, we measure the scale-out of segmented invariant confluent replication. We repeat Benchmark 1 while we vary the number of servers that we use to replicate our object. When we replicate with n servers, we use $3n$ clients (the 3 colocated clients on each server) as part of the workload. The results are shown in Figure 7.

Eventually consistent replication scales perfectly with the number of nodes, confirming the results in [8]. Linearizable replication, on the other hand, scales up to about 3 servers before performance begins to decrease. Segmented invariant confluent replication scales well for low-coordination workloads and poorly for high-coordination workloads. For 1%, 5%, 20%, and 50% decrement transactions, segmented invariant confluent replication scales up to 24, 12, 4, and 1 server respectively.

These results echo the results of Benchmark 1. For low-coordination workloads, segmented invariant confluent replication can offer almost an order of magnitude better scalability compared to linearizable replication, but coordination decreases scalability superlinearly. Even infrequent coordination can drastically reduce the scalability of segmented invariant confluent replication with segmented invariant confluent replication ultimately scaling worse than linearizable replication for high-coordination workloads.

8. RELATED WORK

RedBlue Consistency and SIEVE. RedBlue consistency is a consistency model that sits between causal consistency and linearizability [30]. With RedBlue consistency, every operation is manually labelled as either red or blue. All operations are executed with causal consistency, but with the added restrictions that red operations are executed in a single total order embedded within the causal ordering. In [30], 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 [29], 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. In contrast with these techniques, our invariant confluence decision procedures can determine the invariant confluence of objects that are *not* invariant safe.

The Demarcation and Homeostasis Protocols. The homeostasis protocol [40], a generalization of the demarcation protocol [11], uses program analysis to avoid unnecessary coordination between servers in a *sharded* database (whereas invariant confluence targets *replicated* databases). The protocol guarantees that transactions are executed with observational equivalence with respect to some serial execution of the transactions. This means that intermediate states may be inconsistent, but externally observable side effects and the final database state are consistent. The observational equivalence guaranteed by the homeostasis protocol is stronger than the guarantees of invariant confluence. As a result, there are invariants and workloads that the homeostasis protocol would execute with more coordination than a segmented invariant confluent execution. Moreover, the homeostasis and demarcation protocols' mechanism of establishing global invariants and operating without coordination so long as the invariants are maintained is very similar to our design of segmented invariant confluence.

Explicit Consistency. Explicit consistency [10] is a consistency model that combines invariant confluence and causal consistency, similar to RedBlue consistency with invariant safety. To determine if a workload is amenable to explicitly consistent replication, Balegas et al. determine if all pairs of transactions can be concurrently executed on the same start state without violating the invariant [10]. Balegas et al. argue that this is a sufficient condition for explicit consistency. It is similar to criterion (3) in Theorem 4. In our work, we take a step further and explore sufficient *and necessary* conditions for invariant confluence. Balegas et al. also describe a variety of techniques—like conflict resolution, locking, and escrow transactions [37]—that can be used to replicate workloads that do not meet their sufficient conditions. Segmented invariant confluence is a general-purpose formalism that can be used to model simple forms of these techniques.

Token Based Invariant Confluence. In [21], 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. An application designer defines a set of tokens and specifies which pairs of tokens conflict, and transactions acquire some subset of the tokens when they execute. This allows the application designer to specify which transactions conflict with one another. Gotsman et al. develop sufficient conditions to determine whether a given token scheme is sufficient to guarantee that a global invariant is never broken. The token based approach allows users to specify certain conflicts that are not possible with segmented invariant confluence because a segmentation only allows transactions within a segment to acquire a single self-conflicting lock.

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.

Serializable Distributed Databases. In Section 7, we saw that segmented invariant confluent replication vastly outperforms linearizable replication for low coordination workloads, and it performs comparably or worse for medium and high coordination workloads. Distributed databases like Calvin [45], Janus [36], and TAPIR [48] employ algorithmic optimizations to implement serializable transactions with high throughput and low latency. While segmented invariant confluent replication will likely always outperform serializable replication for low coordination workloads, these databases make serializable replication the most performant option for executing workloads that require a modest amount of coordination.

Branch and Merge. Bayou [44], Dynamo [18], and TARDiS [16] all take a branch and merge approach to maintaining distributed invariants without coordination. With this approach, servers execute transactions without any coordination but keep track of the causal dependencies between transactions. Periodically, two servers merge states and invoke a user defined merge function to reconcile the divergent states. This approach does *not* provide any formal guarantees that invariants are maintained. Its correctness depends on the correctness of the potentially complex user defined merge functions.

CRDTs. CRDTs [43, 42] are distributed semilattices with inflationary update methods. Due to their algebraic properties, CRDTs can be replicated with strong eventual consistency without the need for any coordination. Our definition of distributed objects and our invariant confluence system model are inspired directly by the corresponding definitions and system models in [43]. CRDTs are eventually consistent but may not preserve invariants. Conversely, invariant confluent objects preserve invariants but may not be eventually consistent. Thus, it is natural (though not necessary) to use CRDTs as distributed objects. If a CRDT is determined to be invariant confluent with respect to a particular invariant and set of transactions, then it achieves a combination of strong eventual consistency and invariant preservation. Any CRDT (e.g., counters, sets, graphs, sequences) can be used for this purpose. Finally, our criteria in Theorem 4 also borrow ideas from CRDTs, exploiting the algebraic properties of semilattices.

CALM Theorem. Bloom [4, 5, 15] and its formalism, Dedalus [6, 3], are declarative Datalog-based programming languages that are designed to program distributed systems. The accompanying CALM theorem [24, 7] states that if and only if a program can be written in the monotone fragment of these languages, then there exists a consistent, coordination-free implementation of the program. The CALM theorem provides guarantees about the consistency of program outputs. It does not directly capture our notions of transactions or invariant maintenance during program execution. Moreover, Bloom and Dedalus are general-purpose programming languages that can be used to implement a variety of distributed systems that are outside of the scope of invariant confluence.

Program Analysis in Database Systems. Using program analysis to improve the performance of database systems is not new. For example, it has been used to improve the performance of database-backed web applications [14, 47, 39] and used to improve the performance of optimistic concurrency control on multi-core machines [47]. Our work on invariant confluence continues the theme of using program analysis to reap the performance benefits gained from understanding program semantics.

9. CONCLUSION

This paper revolved around two major contributions. First, we developed a deeper understanding of invariant closure and invariant confluence by looking at the two criteria with reachability in mind. 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 new consistency model and 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. We found that segmented invariant confluence naturally subsumes existing techniques for maintaining invariants of replicated objects (e.g. locking and escrow transactions), and we developed an interactive decision procedure for segmented invariant confluence.

Through our evaluation, we found that our decision procedures could analyze a number of realistic workloads, each in less than a second. We also showed that segmented invariant confluence can significantly outperform linearizable replication for low-coordination workloads.

Acknowledgments. The authors would like to thank Alan Fekete, Peter Alvaro, Alvin Cheung, Alexandra Meliou, Anthony Tan, and Cristina Teodoropol for fruitful discussion and feedback. 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. Alvaro, T. J. Ameloot, J. M. Hellerstein, W. Marczak, and J. Van den Bussche. A declarative semantics for dedalus. Technical Report UCB/EECS-2011-120, EECS Department, University of California, Berkeley, Nov 2011.

[4] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. Sears. Boom analytics: exploring data-centric, declarative programming for the cloud. In *Proceedings of the 5th European conference on Computer systems*, pages 223–236. ACM, 2010.

[5] P. Alvaro, N. Conway, J. M. Hellerstein, and W. R. Marczak. Consistency analysis in bloom: a calm and collected approach. In *CIDR*, pages 249–260, 2011.

[6] P. Alvaro, W. R. Marczak, N. Conway, J. M. Hellerstein, D. Maier, and R. Sears. Dedalus: Datalog in time and space. In *Datalog Reloaded*, pages 262–281. Springer, 2011.

[7] T. J. Ameloot, F. Neven, and J. Van den Bussche. Relational transducers for declarative networking. *Journal of the ACM (JACM)*, 60(2):15, 2013.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[13] E. Brewer. Cap twelve years later: How the" rules" have changed. *Computer*, 45(2):23–29, 2012.

[14] A. Cheung, S. Madden, A. Solar-Lezama, O. Arden, and A. C. Myers. Using program analysis to improve database applications. *IEEE Data Eng. Bull.*, 37(1):48–59, 2014.

[15] N. Conway, W. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier. Logic and lattices for distributed programming. Technical Report UCB/EECS-2012-167, EECS Department, University of California, Berkeley, Jun 2012.

[16] N. Crooks, Y. Pu, N. Estrada, T. Gupta, L. Alvisi, and A. Clement. Tardis: A branch-and-merge approach to weak consistency. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1615–1628. ACM, 2016.

[17] 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.

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

[19] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *PVLDB*, 7(4):277–288, 2013.

[20] 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.

[21] 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.

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

[23] A. Gupta and J. Widom. Local verification of global integrity constraints in distributed databases. *ACM SIGMOD Record*, 22(2):49–58, 1993.

[24] J. M. Hellerstein. The declarative imperative: experiences and conjectures in distributed logic. *ACM SIGMOD Record*, 39(1):5–19, 2010.

[25] 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.

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

[27] D. Kröning, P. Rümmer, and G. Weissenbacher. A proposal for a theory of finite sets, lists, and maps for the smt-lib standard. In *Informal proceedings, 7th International Workshop on Satisfiability Modulo Theories at CADE*, volume 22, 2009.

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

[29] 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.

[30] 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.

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

[32] B. Liskov and J. Cowling. Viewstamped replication revisited. Technical Report MIT-CSAIL-TR-2012-021, CSAIL, Massachusetts Institute of Technology, July 2012.

[33] 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.

[34] 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.

[35] C. Mohan, B. Lindsay, and R. Obermarck. Transaction management in the r* distributed database management system. *ACM Transactions on Database Systems (TODS)*, 11(4):378–396, 1986.

[36] S. Mu, L. Nelson, W. Lloyd, and J. Li. Consolidating concurrency control and consensus for commits under conflicts. In *OSDI*, pages 517–532, 2016.

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

[38] D. Ongaro and J. K. Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*, pages 305–319, 2014.

[39] K. Ramachandra, R. Guravannavar, and S. Sudarshan. Program analysis and transformation for holistic optimization of database applications. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, pages 39–44. ACM, 2012.

[40] 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.

[41] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[42] 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.

[43] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*, pages 386–400. Springer, 2011.

[44] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. *Managing update conflicts in Bayou, a weakly connected replicated storage system*, volume 29. ACM, 1995.

[45] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 1–12. ACM, 2012.

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

[47] Y. Wu, C.-Y. Chan, and K.-L. Tan. Transaction healing: Scaling optimistic concurrency control on multicores. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1689–1704. ACM, 2016.

[48] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. Ports. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 263–278. ACM, 2015.