Automating the choice of consistency levels in replicated systems

Cheng Li[‡], João Leitão[†], Allen Clement[‡], Nuno Preguiça[†], Rodrigo Rodrigues[†], Viktor Vafeiadis[‡]
[‡] *MPI-SWS* [†] *NOVA Univ. of Lisbon / CITI / NOVA-LINCS*

Abstract

Online services often use replication for improving the performance of user-facing services. However, using replication for performance comes at a price of weakening the consistency levels of the replicated service. To address this tension, recent proposals from academia and industry allow operations to run at different consistency levels. In these systems, the programmer has to decide which level to use for each operation. We present SIEVE, a tool that relieves Java programmers from this error-prone decision process, allowing applications to automatically extract good performance when possible, while resorting to strong consistency whenever required by the target semantics. Taking as input a set of application-specific invariants and a few annotations about merge semantics, SIEVE performs a combination of static and dynamic analysis, offline and at runtime, to determine when it is necessary to use strong consistency to preserve these invariants and when it is safe to use causally consistent commutative replicated data types (CRDTs). We evaluate SIEVE on two web applications and show that the automatic classification overhead is low.

1 Introduction

To make web services more interactive, the providers of planetary-scale services—such as Google, Amazon, or Facebook—replicate the state and the application logic behind these services either within a data center or across multiple data centers, and direct users to a single (and preferably the closest or least loaded) replica [11, 28, 14]. Gaining performance through replication, however, comes at a price. To avoid the high cost of coordinating among replicas, the infrastructures that provide replicated services resort to weak consistency levels such as causal consistency [23, 4], eventual consistency [11], or timeline consistency [10]. Under these weak consistency models, good performance is extracted by the fact that only a small number of replicas needs to be contacted for the execution of each operation before producing a reply to the user. However, this adaption also modifies the semantics provided by the replicated service, when compared to strong consistency models like serializability [36] or linearizability [13], where a replicated system behaves like a single server that serializes all operations. Using weak consistency models requires special care, because their semantics may violate user expectations, for example by allowing an auction service to declare two different users to be the winners of the same auction.

Recognizing this tension between performance and meeting user expectations, many research [18, 30, 20, 33] and commercial [31, 12, 25] systems offer the choice between executing an operation under a strong or a weak consistency model. All of these proposals require the application programmer to declare which operations should run under which consistency level. In most cases this is done explicitly by extending the interface for operation execution with an indication of the desired consistency level, while in a recent proposal this is done implicitly by associating a consistency SLA to each operation, ranking and assigning utility values to the various consistency levels [33]. The problem with these strategies is that they impose on the application programmer the non-trivial burden of understanding the semantics of each operation and how the assignment of different consistency levels to different operations influences overall semantics that are perceived by the users.

In this paper, we address this problem by automating the process that assigns consistency levels to the various operations, focusing on an important and widely deployed class of applications, namely Java-based applications with a database backend. To achieve this goal, we build on prior work [20] that defines sufficient properties for safely using a weak consistency model (namely causal consistency), and changes the replication model to separate the generation of the side effects of an operation from their application to the state of the replicas. Adapting existing applications to this model requires manual work that can be challenging and error-prone. First, one must transform every application operation into a generator and a commutative shadow operation. Second, one must correctly identify which shadow operations may break some application invariant, and label them appropriately so that they execute under strong consistency.

In order to ease the burden on the programmer, we have designed SIEVE, a tool that automates this adaptation. Using SIEVE, we require the programmer to only specify the application invariants that must be preserved and to annotate a small amount of semantic information about how to merge concurrent updates. SIEVE achieves this automation by addressing the two identified challenges using the following approach:

First, to ensure convergence under weak consistency, SIEVE automatically transforms the side effects of every application operation into their commutative form. To this end, we build on previous work on commutative replicated data types (CRDTs) [29, 26], i.e., data types whose concurrent operations commute, and apply this concept to relational databases. This allows programmers to only specify which particular CRDT semantics they intend by adding a small annotation in the database schema, and SIEVE automatically generates the shadow operation code implementing the chosen semantics.

Second, SIEVE uses program analysis to identify commutative shadow operations that might violate application-specific invariants when executed under weak consistency semantics, and runs them under strong consistency [20]. To make the analysis accurate and lightweight, we divide it into a potentially expensive static part and an efficient check at runtime. The static analysis generates a set of abstract forms (*templates*) that represent the space of possible shadow operations produced at runtime, and identifies for each template a logical condition (*weakest precondition*) under which invariants are guaranteed to be preserved. This information is then stored in a dictionary, which is looked up and evaluated at runtime, to determine whether each shadow operation can run under weak consistency.

We evaluate SIEVE using TPCW and RUBiS. Our results show that it is possible to achieve the performance benefits of weakly consistent replication when it does not lead to breaking application invariants without imposing the burden of choosing the appropriate consistency level on the programmer, and with a low runtime overhead.

2 Background

Before presenting the various aspects of SIEVE, we first introduce the system model it builds upon, and the operation classification methodology it relies on.

In previous work [20], we defined RedBlue consistency, where operations can be labeled red (strongly consistent) or blue (weakly consistent). Red operations are totally ordered with respect to each other, meaning that they execute in the same relative order at all replicas, and therefore no two red operations execute concurrently. (This corresponds to the requirements of serializability.) In contrast, blue operations can be reordered with respect to other operations, provided they preserve causality (corresponding to causal consistency).

A pre-requisite to being able to label operations as blue is that operations should commute, so that executing them in a different order at various replicas does not lead to a divergent replica state. To increase the space of commutative operations, we proposed a change in the state machine replication model such that operations are split between a generator operation running only on the replica that first receives the operation and producing no side effects, and a shadow operation sent to all replicas, which effectively applies the side effects in a commutative way. More formally, in the original state machine replication model, an operation u deterministically modifies the state of a replica from S to S' (denoted as $S+u=S'$). In the proposed model, the application programmer decomposes every operation u into generator and shadow operations g_u and $h_u(S)$, respectively, where S is the replica state against which g_u was executed. The pair of generator and shadow operations must satisfy the following correctness requirement: for any state S, $S+g_u=S$ and $S+h_u(S)=S+u$.

Given this system model, we defined sufficient conditions for labeling operations in a way that ensures that application invariants are not violated. In particular, a shadow operation can be labeled blue if it commutes with all other shadow operations, and it is *invariant safe*, meaning that if states S and S' preserve the invariants, then the state $S'+h_u(S)$ does so as well.

3 Overview

Using RedBlue consistency requires the programmer to generate commutative shadow operations and identify which can be blue and which must be red. Our goal is to automate these two tasks, to the extent possible.

For the first task, we leverage the rich commutative replicated data type (CRDT) literature [29, 26], which defines a list of data types whose operations commute. CRDTs can be employed to produce commutative shadow operations that converge to identical final states, independent of the order in which they are applied. Shadow operations are thus constructed as a sequence of updates to CRDT data types that commute by construction.

The challenge in developing shadow operations based on CRDTs is that the programmer must explicitly transform the applications to replace all the application state mutations by calls to the appropriate CRDT object. This involves not only identifying the parts of the programs that encode these actions, but also understanding the catalogue of CRDT structures and choosing the appropriate one. To minimize this programmer intervention, we focus on two-tier architectures that store all of the state that must persist across operations in a database. This gives us two main advantages: (1) We can automatically identify the actions that mutate the state, namely the operations that access the database. (2) We can reduce the user intervention to small annotations referring to the database data organization.

The second challenge SIEVE addresses is automatically labeling commutative shadow operations. To this end, for each shadow operation that is generated, we need to decide whether it is invariant safe, according to the definition in Section 2. (Commutativity does not need to be checked since the previous step ensures that shadow operations commute by design.) To automate the classification process, two design alternatives that represent two ends of a spectrum: (1) a dynamic solution, which determines at runtime, when the shadow operation is produced, whether that shadow operation meets the invariant safety property, and (2) a fully static solution that determines which combinations of initial operation types, parameters, and initial states they are applied against lead to generating a shadow operation that is invariant safe. The problem with the former solution is that it introduces runtime overheads, and the problem with the latter solution, as we will detail in Section 5, is that the static analysis could be expensive and end up conservatively flagging too many operations as strongly consistent.

To strike a balance between the two approaches, we split the labeling into a potentially expensive static part and a lightweight dynamic part. Statically, we generate a set of templates corresponding to different possible combinations of CRDT operations that comprise shadow operations, along with weakest preconditions for each template to be invariant safe. Then, at runtime, we perform a simple dictionary lookup to determine which template the shadow operation falls into, so that we can retrieve the corresponding weakest precondition and determine whether it is met.

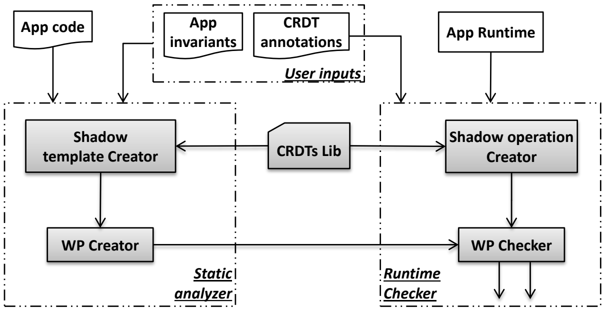


Figure 1: Overview of SIEVE. Shaded boxes are system components comprising SIEVE. (WP stands for weakest precondition.)

These two main solutions lead to the high level system architecture depicted in Figure 1. The application programmer writes the *application code* as a series of transactions written in Java, which access a database for storing persistent state. Beyond the application code, the only additional inputs that the programmer needs to provide are *CRDT annotations* specifying the semantics for merging concurrent updates and a set of application-specific invariants. The static analyzer then creates *shadow operation templates* from the code of each transaction, where these templates represent different sequences of invocations of functions in a *CRDT library*. The analyzer also computes the *weakest preconditions* required for each template to be invariant safe.

At runtime, application servers run both the Java logic and the *runtime checker*, and interact with a database server (not shown in the figure) and the replication tier (not shown in the figure). While executing a transaction, the application server runs the generator operation inside a *shadow operation creator*, which, instead of directly committing side effects to the database, generates a shadow operation consisting of a sequence of invocations from the CRDT library. This shadow operation is then fed to the *weakest precondition checker* to decide which static template it falls into, and what is the precondition required for the operation to be invariant safe, which allows the runtime to determine how to label the operation. The labeled shadow operation is then fed to the replication system implementing multi-level consistency. In the following sections we further discuss the design and implementation of the main components of this architecture.

4 Generating shadow operations

This section covers how we automate the conversion of application code into commutative *shadow operations*.

4.1 Leveraging CRDTs

We leverage several observations and technologies to achieve a sweet spot between the need to capture the semantics of the original operation when encoding its side effects and the desire to minimize the amount of programmer intervention. First, we observe that many applications are built under a two-tier model, where all the persistent state of the service is stored in a relational database accessed through SQL commands. Second, we leverage CRDTs [26], which construct operations that commute by design by encapsulating all side effects into a library of commutative operations.

SQL type	CRDT	Description
FIELD*	LWW	Use last-writer-wins to solve concurrent updates
	NUMDELTA	Add a delta to the numeric value
TABLE	AOSET, UOSET, AUSET, ARSET	Sets with restricted operations (add, update, and/or remove). Conflicting ops. are logically executed by timestamp order.

Table 1: Commutative replicated data types (CRDTs) supported by our type system. * FIELD covers primitive types such as integer, float, double, datetime and string.

These two concepts allow us to achieve commutativity while overcoming the disadvantage of CRDTs, namely the need to adapt applications. This is because the state of two-tier applications is accessed through the narrow SQL interface, and therefore we can focus exclusively on adapting the implementation of SQL commands to access a CRDT. For example, database tables can be seen as a set of tuples, and therefore all the calls in the original operation to add or remove tuples in a table can be replaced in the shadow operation with a CRDT set add or remove, which, in turn, is implemented on top of the database. The programmer only has to select the appropriate merging strategy (i.e., the adequate CRDT type) to encode these operations, without being required to program these CRDT transformations or to change the code of each operation.

However, it is impossible to completely remove the programmer from the loop, due to the choice of which CRDT to use for encoding appropriate merging semantics. For instance, when an integer field of a tuple is written to in a SQL update command, the programmer could have two different intentions in terms of what the update means and how concurrent updates should be handled: (1) the update can represent a delta to be added or subtracted from the current value (e.g., when updating the stock of a certain item), in which case all concurrent updates should be applied possibly in a different order at all replicas to ensure that no stock changes are lost, or (2) it can be overwriting an old value with a new value (e.g., when updating the year of birth in a user profile), in which case an order for these updates should be arbitrated, and the last written value should prevail. Even though both strategies ensure convergence, their semantics differ significantly. For example, the second strategy leads to a final state that does not reflect the effects of all update operations.

Since the appropriate merging strategy is application-specific, the programmer has to convey this decision. To minimize this input, we only require the programmer to declare such semantics on a per-table and per-attribute basis. In more detail, we provide programmers a number of CRDT types (shown in Table 1). These types form two categories: field, which is the smallest component of a record and defines its commuting update operation in the presence of concurrency, and set, which is a collection of such records plus the support for commutative appending or removing. Programmers only need to annotate the data schema with the desired CRDT type using the following annotation syntax: @[*CRDTName*][*TableName*|*DataFieldName*]

```
@AUSET CREATE TABLE exampleTable (
    objId INT(11) NOT NULL,
    @NUMDELTA objCount INT(11) default 0,
    @LWW objName char(60) default NULL,
    PRIMARY KEY (id)
) ENGINE=InnoDB
```

Figure 2: Annotated table definition schema.

Figure 2 presents a sample annotated SQL table creation statement. We assign `exampleTable` the type `AUSET` (Append-Update Set), a CRDT set that only allows append and update operations, thus precluding the concurrent insertion and deletion of the same item (less restrictive CRDT sets also exist). The field `objCount` associated with `NUMDELTA` always expects a delta value to be added or subtracted to its current value. By default, if no annotations are provided, we conservatively mark the corresponding table or field to be read-only.

4.2 Runtime creation of shadow operations

With these schema annotations in place, it is easy to generate commutative shadow operations at runtime. The idea is to invoke the original operation upon the arrival of a new user request (as would happen in a system that does not make use of shadow operations) but with the difference that all the calls to execute commands in the database are intercepted by a modified JDBC driver that builds the sequence of CRDT operations that comprise the shadow operation as the original operation progresses. Furthermore, using the schema annotations, SIEVE maps each database update to an appropriate merge semantics and replaces the operations on a certain table with the appropriate operations over the corresponding CRDT type.

For instance, to create a shadow operation for a transaction that updates `objCount` in Figure 2, when an update is invoked, we first query the old value s, and then, given the new value s', we compute a `delta` by subtracting s from s'. Finally, we use `delta` and the primary key pk of the corresponding object to parameterize a CRDT operation that reads the tuple identified by pk and then adds `delta` to it.

Finally, when the original operation issues a commit to the database, the tool outputs a shadow operation containing the accumulated sequence of CRDT operations.

```
Begin transaction;
for(int i = 0; i < x.length; i++){
  if(x[i] < 100)
    x[i]++;
  else
    x[i] = -100            }
End transaction;
```

(a) Original code

```
func txnShadow(int[] obsX, int[] deltaA){
    for (i = 0; i < obsX.length; i++){
        if(obsX[i] < 100)
            CRDT_x[i].applyDelta(deltaA[i]);
        else:
            CRDT_x[i].applyDelta(deltaA[i]); }
}
```

(b) Possible corresponding shadow template

Figure 3: Code snippet of a transaction and a possible template for the corresponding shadow operation.

5 Classification of shadow operations

In this section we explain how we automatically label shadow operations as strongly or weakly consistent.

5.1 Overview

As mentioned in Section 3, a possible solution would be to statically compute the combinations of operation types, parameters, and initial states that generate invariant safe shadow operations. This can be done by performing a weakest precondition computation—a common technique from Programming Languages and Verification research for which some tool support already exists—which enables us to statically compute, given the code of each operation and the application-specific invariants (which are inserted as postconditions), a precondition over the initial state and operation parameters that ensures the invariant safety property. However, this raises the following two important problems.

First, there is a scalability problem, which is exemplified by the following hypothetical code for the generator operation, assuming an invariant that the state variable x should be non-negative. (For simplicity, we write conventional Java code accessing variable x instead of SQL.)

```
void generator(string s) {
  if (SHA-1(s)==SOME_CONSTANT) {
     if (x>=10)
        x -= 10;
  } else
     x +=10;
}
```

The problem with this code is that a weakest precondition analysis to determine which values of s lead to a negative (non-invariant safe) delta over x is computationally infeasible, since it amounts to inverting a hash function. As such, we would end up conservatively labeling the shadow operations generated by this code as red (i.e., the weakest precondition would be FALSE). Even though this is an extreme example, it highlights the difficulty in handling complex conditions over the input, even when the side effects are simple. In particular, that there are only three patterns of side effects produced by this generator, regardless of the inputs provided to the generator operation. Based on this observation, to simplify the weakest precondition computation and to minimize the space of strongly consistent shadow operations, our static analysis is conducted over the set of possible sequences of CRDT operations that can be generated, which is the same as saying that we analyze all possible shadow operations. We call each possible sequence of shadow operations that can be generated by a given generator operation a template. In the above example, there are only three sequences of shadow operations that can be generated: the empty sequence, adding a delta of 10, and adding a delta of -10. From these three possible sequences, only a delta of -10 leads to a weakest precondition of FALSE, i.e., is always non-invariant safe. The remaining ones have a weakest precondition of TRUE.

The second challenge that needs to be overcome is related to handling loops. The generator code in Figure 3(a) illustrates that the number of iterations in the loop can be unbounded, which in turn leads to an unbounded number of CRDT operations in the shadow operation. To abstract this, we could produce a template that preserves the loop structure, such as the one in Figure 3(b). However, when computing a weakest precondition over this piece of code, verification tools face a scalability problem, which is overcome by requiring the programmer to specify loop invariants that guide the computation of this weakest precondition [17]. Again, this would represent an undesirable programmer intervention.

To address this challenge, we note that in many cases (including all applications that we analyzed), loop iterations are independent, in the sense that the parts of the state modified in each iteration are disjoint. Again, this is illustrated by the example in Figure 3, where the loop is used to iterate over a set of items, and each iteration only modifies the state of the item being iterated.

This iteration independence property enables us to significantly simplify the handling of loops. In particular, when generating the weakest precondition associated with a loop, we only have to consider the CRDT operations invoked in two sets of control flow paths, one where the code within the loop is never executed, and another with all possible control flow paths when the loop is executed and iteration repetitions are eliminated. (We will explain in detail how to handle loops using an example

Sequential path	Description
$2 \cdot 3 \cdot 4 \cdot 2$	only if
$2 \cdot 3 \cdot 6 \cdot 2$	only else
$2 \cdot 3 \cdot 4 \cdot 2 \cdot 3 \cdot 6 \cdot 2$	else follows if
$2 \cdot 3 \cdot 6 \cdot 2 \cdot 3 \cdot 4 \cdot 2$	if follows else

Table 2: Distinct sequential paths obtained for the transaction in Figure 3(a).

in the following subsection.) This condition can then be validated against each individual iteration of the loop at runtime and, given the independence property, this validation will be valid for the entire loop execution.

In our current framework, the iteration independence property is validated manually. In all our case-study applications, it was straightforward to see that this property was met at all times. We leave the automation of this step as future work.

5.2 Generating templates

Instead of reasoning about the generator code, our analysis is simplified by reasoning about the side effects of each code path taken by the generator operation. Furthermore, we can cut the number of possible code paths by eliminating code sections that are repeated due to loops.

To perform this analysis, we require an algorithm for extracting the set of sequential paths of a transaction and eliminating loop repetition. The high level idea of this algorithm is to split branch statements and replace loops with all non-repeating combinations of branches that can be taken within a loop. The algorithm works as follows. First, for every transaction, we create its path abstraction, which is a regular expression encoding all control flow information within that transaction. In the example shown in Figure 3(a), its path abstraction is $2 \cdot (3 \cdot (4|6) \cdot 2)*$, where numbers represent the statement identifiers shown in the figure, $\cdot$ concatenates two sequential statements, $|$ is a binary operator that indicates that the statements at its two sides are in alternative branches, and $*$ represents repetition within a loop. Second, we recursively apply the following two steps to simplify a path abstraction until it is sequential (i.e., no $*$ and $|$). For a path abstraction containing $*$, we create two duplicated abstractions, where one excludes the entire loop, and the other simplifies the loop into its body. For a path abstraction containing the operator $|$, we create two duplicated path abstractions, where one excludes the right operand and the other excludes the left operand. Additionally, if such $|$ is affected by a $*$, then we have to create another path abstraction combining both alternatives, i.e., where the if and the else sides are executed sequentially.

In the previous example, the set of sequential paths that is produced is shown in Table 2. By ignoring the read-only path where the loop is not executed, we only consider four cases, namely only the if or the else path, and the two sequences including both if and else. Because of the loop independence property, these cases are able to capture all relevant sequences of shadow operations. Note that we would only require considering one of the two orderings for the if and the else code within the loop, since their side effects commute, but taking both orderings into account simplifies the runtime matching of an execution to its corresponding path.

Given a set of sequential paths for a transaction, creating shadow operation templates become straightforward. For each path, we collect a sequence of statements specified by the identifiers in the abstraction from the corresponding control flow graph. Then, we translate every database function call into either a CRDT operation by following the instructions stated in Section 4, or a no-op operation (for read-only queries). Finally, all these CRDT operations are packed into a function, which denotes the shadow operation template. These CRDT operations are parameterized by their respective arguments, and the static analysis computes a weakest precondition over these arguments for the template to be invariant safe.

The final output from the static analysis is a dictionary consisting of a set of $< key, value >$ pairs, one for each previously generated shadow operation template, where *key* is the unique identifier of the template, and *value* is the weakest precondition for the template. The unique identifier of the template encodes the set of possible paths using signatures of CRDT operations in a restricted form of regular expression.

5.3 Runtime evaluation

Template/shadow operation matching. At runtime, it is necessary to evaluate the weakest precondition to classify operations as red or blue. To this end, we must lookup in the dictionary created during the static analysis the template corresponding to each shadow operation as it is produced.

The challenge with performing this lookup is that it requires determining the identifier of the shadow operation corresponding to the path taken, and this must be done by taking into account *only* the operations that are controlled by the runtime, i.e., the CRDT operations. This explains why the dictionary keys consist only of CRDT operations. With the shadow operation identifier, matching the path taken at runtime with the keys present in the dictionary is done efficiently by using a search tree.

Weakest precondition check. Finally, once the weakest precondition for the template that corresponds to a particular shadow operation is retrieved, we evaluate that precondition against the CRDT parameters of the shadow operation. This is achieved by simply replacing the variables in the precondition with their instantiated values and evaluating the final expression to either true or

App	Invariants
TCPW	$\forall item \in item\_table.\ \ item.stock \geq 0$
RUBiS	$\forall item \in item\_table.\ \ item.stock \geq 0$
	$\forall u, v \in user\_table.$ $u.uname = v.uname \implies u = v$

Table 3: Application-specific invariants

false. If the weakest precondition is evaluated to true the shadow operation is labeled blue, otherwise the shadow operation in labeled red.

After this step, the shadow operation is delivered to the replication layer, which replicates it using different strategies according to its classification.

6 Evaluation

In this section, we report our experience with implementing SIEVE, adapting existing web applications to run with SIEVE, and evaluating these systems.

6.1 Implementation

We implemented most of our tool using Java (15k lines of code), and changed parts of the Jahob code to obtain weakest preconditions in OCaml (553 lines of code). The backend storage system we used was a MySQL database. We used an existing Java parser [1] to parse java files. Finally, we connected our tool to the Gemini replication and coordination system [20] to enable both consistency classification and operation replication.

6.2 Use cases

To adapt an application to use SIEVE, one has to annotate the corresponding SQL schema with the proper CRDT semantics, specify all invariants, and finally the original JDBC driver must be replaced by the driver provided by SIEVE, to enable SIEVE to intercept interactions between the application and the database.

We applied SIEVE to two web application benchmarks, namely TPCW [9] and RUBiS [7]. Both of them simulate an online store and the interactions between users and the web application. There are two main motivations for selecting these use cases: (1) both have been widely used by the community to evaluate system performance; and (2) both have application-specific invariants that can be violated under weak consistency. (In our prior work [20] a social application is evaluated, but it made no sense to include this application because it did not contain any invariants that could be violated under weak consistency.)

For TPCW, we use `AOSET`, `AUSET`, `UOSET` and `ARSET` to annotate the database tables, no annotations for unmodified attributes, `NUMDELTA` for `stock`, and `LWW` for the remaining attributes. For RUBiS, we annotate its tables with `AUSET` and `AOSET`. We use `NUMDELTA` as annotations for both `quantity` and `numOfBids`, and no annotations or `LWW` for the remaining attributes. Identified invariants in these two applications are summarized in Table 3. For additional details, we refer the interested reader to the code available in [2].

In terms of the time required to do this adaptation, we do not report results for TPCW as we relied on this use case during the design and development phase of SIEVE. However for the RUBiS use case, the entire process was concluded in only a few hours. An interesting point to highlight is that SIEVE is able to detect inconsistencies between these annotations, enabling programmers to correct mistakes such as type omissions in the SQL schema that are inconsistent with the CRDT annotations.

In both our prior work [20] and the current work, the effort we made analyzing application code to determine invariants and merge semantics is unavoidable. In our prior work, however, we additionally spent a significant amount of time manually implementing merge semantics, and classifying shadow operations by taking into account their properties, for every application. SIEVE eliminates all this manual work, and limits human error.

6.3 Experimental setup

All reported experiments were obtained by deploying applications on a local cluster, where each machine has 2*6 i7 cores and 48GB RAM, and runs Linux 3.2.48.1 (64bit), MySQL 5.5.18, Tomcat 6.0.35, and Java 1.7.0.

6.4 Experimental results

Our experimental work aims at evaluating both the static analysis component of SIEVE and also the runtime component, which includes a performance comparison between each application using our tool, its unmodified version, and its version under RedBlue consistency where the entire classification is done manually and offline.

Concerning the static analysis component we focus on the following main questions: (*i*) How long does the static analysis process take to complete? (*ii*) What is the scalability of the static analysis component in relation to the size of the code base?

For the runtime component of SIEVE we focus on the following main questions: (*i*) Is the runtime classification of shadow operations accurate? (*ii*) What is the (runtime) overhead for adapted applications compared to their stand-alone unmodified counterparts? (*iii*) What are the performance gains obtained through weakly consistent replication using SIEVE?

6.4.1 Static analysis

As mentioned before, taking the application source code and CRDT annotations as input, SIEVE first maps each transaction into a set of distinct paths, and automatically transforms each path into a shadow operation template.

Transaction name	#paths	#templates	Transaction name	#paths	#templates	Transaction name	#paths	#templates
read-only txns (13)	1	0	createNewCustomer	2	2	doBuyConfirm-A	32	32
createEmptyCart	1	1	adminUpdate	4	4	doCart	36	36
refreshSession	1	1	doBuyConfirm-B	16	16			

Transaction name	#paths	#templates	Transaction name	#paths	#templates	Transaction name	#paths	#templates
ViewUserInfo	6	0	PutComment	10	0	PutBid	14	0
BrowseRegions	5	0	StoreComment	11	3	StoreBid	17	5
BuyNow	7	0	ViewBidHistory	11	0	AboutMe	37	0
SearchItemsByRegion	20	0	StoreBuyNow	13	6	RegisterItem	59	24
SearchItemsByCategory	20	0	BrowseCategories	13	0			
ViewItem	10	0	RegisterUser	14	3			

Table 4: Number of reduced paths and templates generated for each transaction in TPCW (top) and RUBiS (bottom).

Table 4 summarizes the number of paths (excluding loops) and the corresponding number of shadow operation templates that were produced by SIEVE for both TPCW and RUBiS. For TPCW, 15 out of the total 20 transactions only exhibit a single path, as the code of these transactions is sequential. The two most complex transactions in this use case are `doBuyConfirm` and `doCart`, which are associated with the user actions of shopping and purchasing. In contrast, most transactions in RUBiS have a more complex control flow, which generated a larger number of possible execution paths.

Note that the majority of transactions in both use cases do not lead SIEVE to produce any template. This happens when the transactions are read-only, and therefore do not have side effects. Additionally, in TPCW every path in an update transaction generates a shadow operation template, since system state is always modified. However, this is not true in RUBiS, because its code verifies several conditions, some of which lead to a read-only transaction.

As depicted in Table 5, the execution of SIEVE generated a total of 92 and 41 shadow operation templates for TPCW and RUBiS, respectively. In addition to these templates, our tool also generates automatically a set of Java classes that represent database data structures, which are necessary for computing weakest preconditions.

Table 6 depicts a full list of the different weakest preconditions generated by SIEVE for both use cases. These weakest preconditions alongside their respective shadow operation template identifiers are used by the runtime logic to classify shadow operations as either blue or red. A weakest precondition denoted by `True` implies that any shadow operation associated with that template is always invariant safe and therefore labeled blue. In contrast, a weakest precondition denoted by `False` implies that shadow operations associated to that template must always be classified as red. The remaining non-trivial conditions must be evaluated at runtime by replacing their arguments with concrete values. For instance, when a `doBuyConfirm` transaction produces a negative delta, then the condition will be evaluated to `False` and the corresponding shadow operation will be classified as red, otherwise the condition will be evaluated to `True` and the shadow operation will be classified as blue.

App	#code	templates		#db code	#specs
		num	#code		
TPCW	8.3k	92	1554	879	730
RUBiS	9.8k	41	251	477	371

Table 5: Overview of the output produced by the static analysis. "db code" refers to the Java classes representing database structures required for computing weakest preconditions.

	WP	Comments
TPCW	`True`	Not influencing invariants
	`delta` ≥ 0	Non-negative stock
RUBiS	`True`	Not influencing invariants
	`False`	Nickname must be unique
	`delta` ≥ 0	Non-negative quantity
	`quantity` ≥ 0	Non-negative quantity (new item)

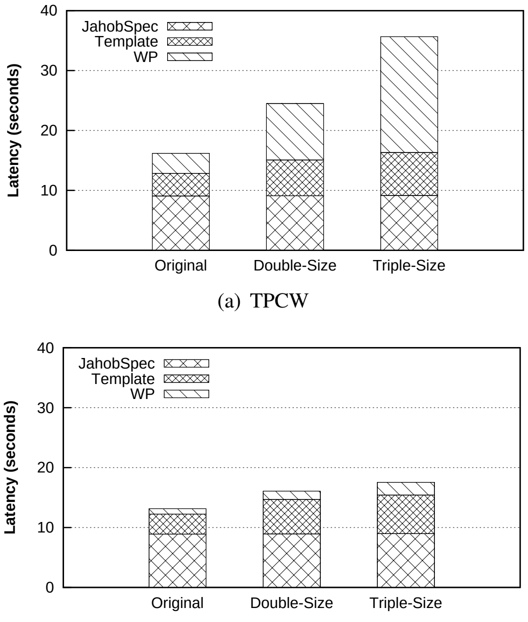
Table 6: Weakest preconditions (WP)

App	JahobSpec	Template	WP	Total
TPCW	9.1 ± 0.1	3.8 ± 0.1	3.3 ± 0.1	16.2 ± 0.3
RUBiS	8.9 ± 0.0	3.3 ± 0.3	0.9 ± 0.1	13.2 ± 0.3

Table 7: Average and standard deviation of latency in seconds for static analysis tasks (5 runs).

Cost of static analysis. A relevant aspect of the static analysis component in SIEVE is the time required to execute it. To study this we have measured the time taken by the static analysis and present the obtained results in Table 7. We not only measured the end-to-end completion time, but also the time spent for each step, namely, creating database data structures required by Jahob (JahobSpec), template creation (Template), and weakest precondition computation (WP). Overall, we can see that the execution time of the static component of SIEVE is acceptable, as less than 20 seconds are required to analyze both TPCW and RUBiS. The code generation phase including both JahobSpec and Template dominates the overall static analysis. Compared to TPCW, the time spent computing weakest preconditions is shorter in RUBiS, due to the smaller number of templates in Table 5.

Scalability. The code base size of TPCW and RUBiS is somewhat small when compared to deployed applica-



(a) TPCW

(b) RUBiS

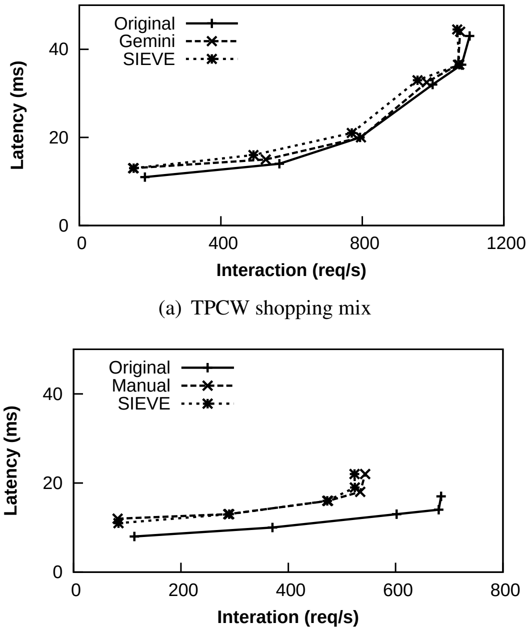
Figure 4: Static analysis time vs. code base size.

tions. This raises a question concerning the scalability of the static analysis component of SIEVE with respect to the size of the code base. In order to analyze this aspect of SIEVE we have artificially doubled and tripled the size of each application code base and measured the time spent analyzing these larger code bases when compared with the original. The results are shown in Figure 4. The time spent generating the data structures required by Jahob is constant, since we did not change the database schema. However, the time spent computing the weakest preconditions for templates in TPCW grows exponentially, and the time taken for the remaining steps presents a sub-linear increase. These results lead us to conclude that the static analysis of SIEVE may scale to reasonable code sizes, especially taking into account that this process is executed a single time when adapting an application through the use of SIEVE.

6.4.2 Runtime logic

We evaluated the runtime performance of our example applications using SIEVE on top of Gemini, which is a coordination and replication layer supporting generator and shadow operation execution [20].

Configurations. We populated the dataset for TPCW using the following parameters: 50 EBS and 10,000 items. For RUBiS we populated the dataset with 33,000 items for sale, 1 million users, and 500,000 old items. We exercised all TPCW workloads, namely browsing mix, shopping mix, and ordering mix, where the purchase activity varies from 5% to 50%. For RUBiS, we ran the bidding mix workload, in which 15% of all user activities generate updates to the application state.



(a) TPCW shopping mix

(b) RUBiS bidding mix

Figure 5: Throughput-latency graph without replication

Correctness validation. To verify that SIEVE labels operations correctly for both case studies, we inspected the log files generated by running SIEVE with TPCW and RUBiS, and we found that SIEVE conducts the same classification that was achieved manually in our previous work [20].

SIEVE runtime overhead. Next we compared the performance (throughput vs. latency) of the two applications across three single-site deployments: 1) SIEVE, 2) Original—the original unreplicated service without any overheads from creating and applying shadow operations, and 3) Manual—the RedBlue scheme with all labeling performed offline by the programmer. The expected sources of overhead for SIEVE are: *i*) the dynamic creation of shadow operations; and *ii*) the runtime classification of each shadow operation. The results in Figure 5 show that the performance achieved by SIEVE is similar to the one obtained with a manual classification scheme, and therefore the overheads of runtime classification are low. The comparison with the original scheme in a single site shows some runtime overhead due to creating and applying shadow operations (which is required for a replicated deployment so that all operations commute).

To better understand the sources of overhead imposed by SIEVE we measured the latency contribution of each runtime step executed by SIEVE and compared it with the latency contribution of these steps when relying on a manual adaptation. In particular, we focused on the following tasks: generator execution (producing a shadow operation), classification (determining shadow operation colors), and shadow execution (applying shadow opera-

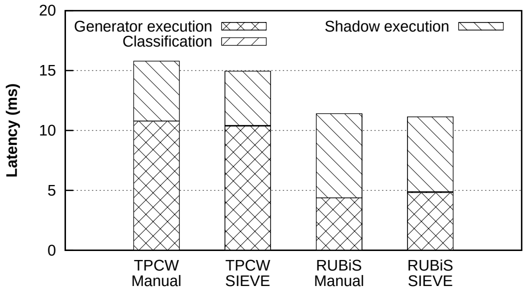


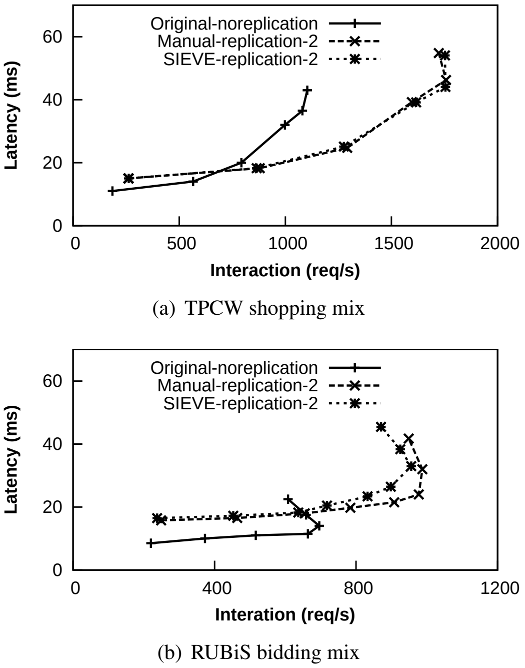
Figure 6: Breakdown of latency.

tions).

Figure 6 shows the average contribution to request latency of each of these steps (Only update requests are considered since read-only queries do not generate side effects.) For the manual adaptation, there is no latency associated with classifying shadow operations, since the classification of all shadow operations is pre-defined. In contrast, SIEVE performs a runtime classification, but the results show that the time consumed in this task is negligible. In particular, SIEVE takes 0.064 ± 0.002 *ms* and 0.072 ± 0.001 *ms* for looking up the dictionary and evaluating the condition for TPCW and RUBiS, respectively. Regarding the generator execution and shadow execution, both the manual adaptation and SIEVE present the same latency overheads.

Replication benefits. The results previously discussed in this section have shown that the use of SIEVE imposes a small overhead when compared to a standalone execution of the unmodified use cases, mostly due to runtime classification. However, SIEVE was designed to allow replication to bring performance gains through the use of weak consistency in replicated deployments. To evaluate these benefits, we conducted an experiment where we deployed the two applications (1) without replication, (2) using manual classification in Gemini, and (3) using SIEVE, with two replicas in the same site for the last two options. (The use of single site replication instead of geo-replication makes our results conservative, since the overheads of runtime classification become diluted when factoring in cross-site latency.)

The results in Figure 7 show that weakly consistent replication for a large fraction of the operations brings performance gains. In particular, one observes that the peak throughput with 2 replicated Gemini instances running TPCW is improved by 59.0%, and the peak throughput for RUBiS in this setting is improved by 37.4%. The additional latency introduced in this case is originated by the necessity of coordination among replicas to totally order red shadow operations. The results also confirm that the overhead of runtime classification when compared to the manual, offline classification are low. Note that there is a point where the throughput goes down while there is still an increase in latency in Figure 7(b). This happens because the database becomes saturated at this point.



(a) TPCW shopping mix

(b) RUBiS bidding mix

Figure 7: Throughput-latency graph with two replicas.

7 Related work

We summarize and compare previous work with SIEVE according to the following categories:

Eventual consistency and commutativity. A large number of replicated systems have relied on eventual consistency for supporting low latency for operations by returning as soon as an operation executes in a single replica. These systems must handle conflicts that may arise from concurrent operations. In some systems, such as Bayou [34], Depot [24], and Dynamo [11], applications must provide code for merging concurrent versions. Other systems, such as Cassandra [19], COPS [22], Eiger [23] and ChainReaction [4], use a simple last-writer-wins strategy for merging concurrent versions. This simple strategy may, however, lead to lost updates.

Some systems have explored using operation commutativity to guarantee that all replicas converge to the same state, regardless of operation execution order. For example, Walter [32] includes a single pre-defined data type with commutative operations, *cset*. This system could be extended for supporting other data types with commutative operations proposed in the literature [26, 29]. Lazy replication [18] and RedBlue [20] support unordered execution of commutative operations defined by programmers. Furthermore, RedBlue [20] extends the space of commutative operations by decoupling operation generation and application, requiring only that operation application code is commutative.

Unlike these systems, SIEVE automatically adapts applications so that commutativity is obtained without modifying existing application code or adopting a new programming model – a commutative operation that encodes the operation side effects is automatically generated from the application code.

Multi-level consistency. As some application operations cannot execute correctly under eventual consistency, a few multi-level consistency models that combine eventual and strong consistency have been proposed [32, 20, 18, 33]. The properties of these models overlap with each other, and differ mainly in the composition of the different consistency levels. For instance, some work [32, 20] has found that it is sufficient to categorize operations into strong and weak consistency. Some other work [33] presents a more fine-grained division for read-only operations, which includes consistent prefix read, monotonic reads, and so on. We build on these models, and, in order to keep our design and our presentation simple, we follow the two-level consistency model proposed by RedBlue consistency [20].

Classification for multi-level consistency. In order to help developers adopt different proposals for multi-level consistency models, their creators introduced a few instructions to guide how to use their work. Relying on a probabilistic model, consistency rationing [16] associates different consistency levels with different states, instead of operations, and allows states to switch from one level to another at runtime. Unlike this approach, we partition operations into strong and eventual consistency groups. Both RedBlue consistency [20] and I-confluence [6] define conditions that operations must meet in order to run under weak consistency, i.e., without coordination. We build on this line of work and extend it so that an automatic tool, and not the programmer, is responsible for determining whether the operations meet these conditions.

To free programmers from the classification process, some researchers have attempted to apply program analysis techniques to reason about the consistency requirements of real applications. Alvaro et al. [5] identify code locations that need to inject coordination to ensure consistency, while Zhang et al. [36] inspect read/write conflicts across all operations. However, they focus on commutativity, and ignore application invariants, which are very important and taken into account by our solution. Very recently, Roy et al. [27] devised a way to summarize transaction semantics to allow sites to execute transactions independently, and without leading to inconsistencies. Their approach differs in the way that invariants are maintained, which resembles the concept of warranties [21] in that some operations cannot proceed so that others execute locally. Furthermore, the paper focuses extensively on the analysis of transaction code to determine their abstract semantics, which is complementary to the goal of our work since we rely on Jahob to determine only certain properties of shadow operations.

Commutativity and classification beyond eventual consistency. Commutativity has been explored in other settings to improve performance and scalability – e.g. in databases [35] and in OS design for multi-core systems [8]. Program analysis techniques have also been used to identify commuting code blocks. Aleen et al. [3] proposed a new approach to find commutative functions automatically at compile time for allowing legacy software to extract performance from many-core architectures. Kim et al. [15] used the Jahob verification system to determine commuting conditions under which two operations can execute in different orders. Unlike these two prior solutions that focus on identifying commutative code blocks, our tool automatically transforms operations by decoupling operation generation and application, which makes more operations commute [20], and we also focus on determining invariant safety.

8 Conclusion

In this paper we presented SIEVE, the first system to automate the choice of consistency levels in a replicated system. Our system relieves the programmer from having to reason about the behaviors that weak consistency introduces, only requiring the programmer to write the system invariants that must be preserved and provide annotations regarding merge semantics. Our evaluation shows that SIEVE labels operations accurately, incurring a modest runtime overhead when compared to labeling operations manually and offline.

Acknowledgments

The authors wish to express their gratitude to the anonymous reviewers, Rose Hoberman, and our shepherd Jinyang Li, whose comments improved the quality of the paper, as well as Thomas Wies for his assistance with Jahob. The research of R. Rodrigues has received funding from the European Research Council under an ERC starting grant. Authors from NOVA were supported by FCT/MCT project PEst-OE/EEI/UI0527/2014. Computing resources for this work were supported by an AWS in Education Grant.

References

[1] The web page of javaparser. `http://code.google.com/p/javaparser`. Accessed May-2014.

[2] SIEVE example files. `http://www.mpi-sws.org/~chengli/atc2014files/`, 2014.

[3] ALEEN, F., AND CLARK, N. Commutativity analysis for software parallelization: Letting program transformations see the big picture. In *Proc. of the 14th ASPLOS* (2009).

[4] ALMEIDA, S., LEITÃO, J., AND RODRIGUES, L. Chainreaction: A causal+ consistent datastore based on chain replication. In *Proc. of the 8th ACM EuroSys* (2013).

[5] ALVARO, P., CONWAY, N., HELLERSTEIN, J., AND MARCZAK, W. R. Consistency analysis in bloom: a calm and collected approach. In *Proc. of the CIDR* (2011).

[6] BAILIS, P., FEKETE, A., FRANKLIN, M. J., GHODSI, A., HELLERSTEIN, J. M., AND STOICA, I. Coordination-avoiding database systems. *CoRR abs/1402.2237* (2014).

[7] CECCHET, E., AND MARGUERITE, J. Rubis: Rice university bidding system. `http://rubis.ow2.org/`, 2009.

[8] CLEMENTS, A. T., KAASHOEK, M. F., ZELDOVICH, N., MORRIS, R. T., AND KOHLER, E. The scalable commutativity rule: Designing scalable software for multicore processors. In *Proc. of the 24th ACM SOSP* (2013).

[9] CONSORTIUM, T. Tpc benchmark-w specification v. 1.8. `http://www.tpc.org/tpcw/spec/tpcw_v1.8.pdf`, 2002.

[10] COOPER, B. F., RAMAKRISHNAN, R., SRIVASTAVA, U., SILBERSTEIN, A., BOHANNON, P., JACOBSEN, H.-A., PUZ, N., WEAVER, D., AND YERNENI, R. Pnuts: Yahoo!'s hosted data serving platform. *In Proc. of the VLDB Endow. 1*, 2 (Aug. 2008).

[11] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's highly available key-value store. In *Proc. of the 21st ACM SOSP* (2007).

[12] GOOGLE. Welcome to google app engine. `https://appengine.google.com/`. Accessed May-2014.

[13] HERLIHY, M. P., AND WING, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12*, 3 (July 1990).

[14] HOFF, T. Latency is everywhere and it costs you sales - how to crush it. `http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it`, 2009.

[15] KIM, D., AND RINARD, M. C. Verification of semantic commutativity conditions and inverse operations on linked data structures. In *Proc. of the 32nd ACM PLDI* (2011).

[16] KRASKA, T., HENTSCHEL, M., ALONSO, G., AND KOSSMANN, D. Consistency rationing in the cloud: Pay only when it matters. *In Proc. of the VLDB Endow. 2*, 1 (Aug. 2009).

[17] KUNCAK, V. *Modular Data Structure Verification*. PhD thesis, EECS Department, MIT, 2007.

[18] LADIN, R., LISKOV, B., SHRIRA, L., AND GHEMAWAT, S. Providing high availability using lazy replication. *ACM Trans. Comput. Syst. 10*, 4 (Nov. 1992).

[19] LAKSHMAN, A., AND MALIK, P. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev. 44*, 2 (Apr. 2010).

[20] LI, C., PORTO, D., CLEMENT, A., GEHRKE, J., PREGUIÇA, N., AND RODRIGUES, R. Making geo-replicated systems fast as possible, consistent when necessary. In *Proc. of the 10th USENIX OSDI* (2012).

[21] LIU, J., MAGRINO, T., ARDEN, O., GEORGE, M. D., AND MYERS, A. C. Warranties for faster strong consistency. In *Proc. of the 11th USENIX NSDI* (2014).

[22] LLOYD, W., FREEDMAN, M. J., KAMINSKY, M., AND ANDERSEN, D. G. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proc. of the 23rd ACM SOSP* (2011).

[23] LLOYD, W., FREEDMAN, M. J., KAMINSKY, M., AND ANDERSEN, D. G. Stronger semantics for low-latency geo-replicated storage. In *Proc. of the 10th USENIX NSDI* (2013).

[24] MAHAJAN, P., SETTY, S., LEE, S., CLEMENT, A., ALVISI, L., DAHLIN, M., AND WALFISH, M. Depot: Cloud storage with minimal trust. In *Proc. of the 9th USENIX OSDI* (2010).

[25] ORACLE. Oracle NoSQL database. `http://www.oracle.com/us/products/database/nosql/overview/index.html`. Accessed May-2014.

[26] PREGUICA, N., MARQUES, J. M., SHAPIRO, M., AND LETIA, M. A commutative replicated data type for cooperative editing. In *Proc. of the 29th IEEE ICDCS* (2009).

[27] ROY, S., KOT, L., FOSTER, N., GEHRKE, J., HOJJAT, H., AND KOCH, C. Writes that fall in the forest and make no sound: Semantics-based adaptive data consistency. *CoRR abs/1403.2307* (2014).

[28] SCHURMAN, E., AND BRUTLAG, J. Performance related changes and their user impact. Presented at velocity web performance and operations conference, 2009.

[29] SHAPIRO, M., PREGUIÇA, N., BAQUERO, C., AND ZAWIRSKI, M. Conflict-free replicated data types. In *Proc. of the 13th SSS* (2011).

[30] SINGH, A., FONSECA, P., KUZNETSOV, P., RODRIGUES, R., AND MANIATIS, P. Zeno: Eventually consistent byzantine-fault tolerance. In *Proc. of the 6th USENIX NSDI* (2009).

[31] SIVASUBRAMANIAN, S. Amazon dynamoDB: A seamlessly scalable non-relational database service. In *Proc. of the ACM SIGMOD* (2012).

[32] SOVRAN, Y., POWER, R., AGUILERA, M. K., AND LI, J. Transactional storage for geo-replicated systems. In *Proc. of the 23rd ACM SOSP* (2011).

[33] TERRY, D. B., PRABHAKARAN, V., KOTLA, R., BALAKRISHNAN, M., AGUILERA, M. K., AND ABU-LIBDEH, H. Consistency-based service level agreements for cloud storage. In *Proc. of the 24th ACM SOSP* (2013).

[34] TERRY, D. B., THEIMER, M. M., PETERSEN, K., DEMERS, A. J., SPREITZER, M. J., AND HAUSER, C. H. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proc. of the 15th ACM SOSP* (1995).

[35] WEIHL, W. E. Commutativity-based concurrency control for abstract data types. *IEEE Trans. Comput.* (1988).

[36] ZHANG, Y., POWER, R., ZHOU, S., SOVRAN, Y., AGUILERA, M. K., AND LI, J. Transaction chains: Achieving serializability with low latency in geo-distributed storage systems. In *Proc. of the 24th ACM SOSP* (2013).