A More Consistent Understanding of Consistency

Subhajit Sidhanta*, Ricardo J. Dias†‡, Rodrigo Rodrigues†
*IIT Bhilai, India
†INESC-ID / Instituto Superior Técnico, ULisboa
‡NOVA LINCS, Universidade NOVA de Lisboa & SUSE Linux GmbH

***Abstract*—Recent storage systems trade strong consistency for performance, availability, and scalability. However, this makes it hard to understand the semantics that the storage system provides, and also makes the design and implementation of the storage system itself more error-prone. This paper proposes a comprehensive solution to these problems. In particular, we propose a specification language named *ConSpec*, which enables the formalization of different consistency semantics that a storage system may provide, using a uniform syntax that is independent of the design and implementation of the target storage system. We use ConSpec to revisit several existing models in light of a common way to define and compare them. Furthermore, we generalize the CAP theorem, whose original formulation only considered linearizability, to precisely define the class of consistency definitions that can and cannot be implemented in a highly-available, partition-tolerant way. Finally, we present the design and implementation of a new consistency checker that takes a trace from a storage system (e.g., the output of a test suite) and validates whether it meets any consistency semantics defined using ConSpec. The evaluation of our consistency checker shows that it is able to verify the correctness of long traces in a reasonable time.**

I. Introduction

The development of Internet-scale applications and services led to the increasing adoption of scalable storage systems that trade performance for semantics. In particular, we have witnessed a profusion of proposals for systems that fall into the "NoSQL" category, which are characterized by compromising consistency semantics in order to scale to a large volume of data, stored in many nodes across different geographic locations.

This phenomenon increases the complexity of the design and implementation not only of the applications, since application developers now need to take into account weaker consistency semantics that are often hard to understand [1], but also of the storage systems themselves, which have to be carefully designed and implemented to provide the intended consistency.

In this paper, we present a comprehensive solution that addresses two important problems that arise in this setting: (1) gaining a better understanding of consistency semantics and its fundamental limitations, and (2) having practical tools to systematically test if a given consistency level is correctly implemented.

In particular, we begin by proposing a specification language called *ConSpec*: a generic definition for consistency models, which can be easily parameterized to obtain precise definitions of a variety of semantics. ConSpec builds on the observation made by several proposals that it is possible to define weak consistency semantics in terms of partial orders over the set of operations that were executed in the system, which comprise a visibility graph [2, 3, 4]. This allows us to reduce the configurable part of the consistency definition to a set of restrictions, written in Linear Temporal Logic (LTL) [5], on top of a generic partial order. Intuitively, these restrictions specify the allowed order in which the results of operations can become visible to a client application.

With this framework in place, we were able to express several existing consistency definitions in a common language, and compare them in a precise way. Further, we show the equivalence between our definitions and their original form.

Based on the same framework, we were also able to revisit the CAP theorem [6, 7] and restate it in more generic terms. The original proof for the theorem used linearizability as synonym for the strong consistency captured by the "C" property. In our new formulation of this theorem, we are able to define necessary and sufficient conditions for a given consistency model to be bound by the impossibility of being implemented in a highly available, partition-tolerant way.

Finally, we built a tool called ConSpecCheck, which is available online, to determine whether a client trace from a storage system satisfies a certain consistency model. This tool can be used with existing testing frameworks, which are used for generating test cases, to attempt to expose bugs that cause storage systems to deviate from the specified consistency semantics. For example, by coupling our system with a set of outputs of different thread interleavings that are generated by a concurrency testing framework like Chess [8], we are able to systematically attempt to uncover concurrency bugs that lead to subtle consistency violations. A noteworthy point is that the ConSpec checker highlights the advantage of our LTL-based specifications: even though we could specify consistency models without resorting to LTL, its use allowed us to seamlessly leverage existing model checking systems in our design.

We built the ConSpec checker and evaluated it using traces produced from workloads from the YCSB and TPC-C benchmarks. Our experimental evaluation shows that the tool is able to validate traces in a reasonable amount of time, and that the time to check a trace against ConSpec is comparable to the

time with prior definitions.

The remainder of this paper is organized as follows. Section II discusses related work. Section III presents the system model and terminology. Section IV presents a general format of specifications based on ConSpec. Section V expresses some example consistency models using ConSpec. Section VI presents our restatement of the CAP theorem, and proves its correctness. Section VII discusses how this extension of the CAP theorem can be used to categorize existing models. In Section VIII, we present the design of our tool for checking if a trace obeys a certain consistency model. In Section IX, we evaluate its performance.

II. Related Work

Consistency definitions restrict the set of valid traces for the execution of a given system. In broad terms, the gold standard of consistency definitions are "strong" consistency levels, which have the characteristic of approximating the behavior that is obtained when interacting with a system whose implementation has a centralized server that executes operations one at a time. There are several examples of such consistency models [9, 10, 11, 12, 13, 14].

Strong consistency is often forfeited by the algorithms that implement storage systems, in order to achieve better performance (e.g., when processors cache possibly stale data in a multiprocessor) and/or better availability (e.g., when multiple replicas of the data exist and operations proceed while contacting only a subset of them) [15, 16, 17, 18, 19, 20].

Often, the definitions of these consistency models are vague and/or underspecified. For instance, the strong consistency option of Cassandra, a widely used NoSQL storage system, is stated in terms of the size of the quorums that are used for read and write operations, leaving unspecified what happens as the system reconfigures and the set of replicas of a data item changes [21]. Even when the specification is more precise, it may suffer from being tied to implementation details that may not be widely applicable. For instance, some definitions assume the existence of a centralized server that keeps monotonically increasing version numbers associated with the data [22]; others explicitly define consistency in terms of the state maintained by different replicas of the data [2]; and other models, namely the well known set of session guarantees that strengthen the consistency offered by eventually consistent systems, are defined operationally in terms of the replicas that are accessed, the operations that these replica process, and the respective order in which they are processed [23].

In [24], the underlying system is represented as a collection of one or more state machines, where each state machine models operations performed in parallel or on different replicas. Consistency models impose constraints on the total order of operations performed by a given state machine. Chockler et al. [25] defined all session guarantees (and some other consistency levels) using first-order logic formulas that constrain a set of equivalent linear sequences comprising all operations that are part of a given execution. Similarly, Burckhardt et al. [26, 27, 28] present the definitions of a broad set of consistency models, by specifying consistency axioms in the form of constraints on *visibility* and *arbitration order* relations, which must be satisfied by a valid abstract execution for a given consistency model. Cerone et al. [29] developed a set of algebraic definitions of consistency for transactional systems, which are based on relations similar to those proposed by Burckhardt et al. to specify conflict resolution policies for ordering concurrent operations. Weber et al. [30] present EPTL, an extension of LTL, which they employ to specify correctness properties in weakly consistent systems. EPTL is used to capture details of the system internals, such as techniques applied in the resolution of conflict among concurrent operations. Wickerson et al. developed a framework, based on the Alloy language, for modeling consistency in shared memory systems and performing verification of a system against a given consistency model [31].

Compared to these prior approaches, ConSpec advances the state of the art by providing a generic way to describe consistency specifications, where each consistency level corresponds to a different parameterization of our generic definition. Using this generic framework we generalize the CAP theorem as proved by Gilbert and Lynch [7]. Another important distinguishing feature of our work is that we provide a software artifact to check whether traces meet a certain consistency level, leveraging our LTL definitions.

Our definitions are also related to recent proposals for models that are weakly consistent by default but distinguish a subset of the operations, and enforce visibility restrictions only among those [2, 3, 4]. In contrast to these proposals, our goal is not to propose a new consistency model, but to gain a deeper understanding of existing ones.

Other authors have explored the CAP theorem beyond its original formulation and first proof. Mahajan et al. [32] defined Real Time Causal Consistency, a stronger variant of causal consistency, and proved that it is the strongest consistency model that can be provided in a highly available and eventually consistent implementation. Attiya et al. [33] provided a formal specification of systems that implement causal consistency, and proved that Observable Causal Consistency, a stronger variant of causal consistency, is the strongest consistency model that can be provided in a highly available, partition tolerant manner. Their definitions are stated in terms of some implementation-level concepts, namely a set of replicas connected by a network. They assume a replicated datastore, and define a consistency model in terms of events (operations) observed at each individual replica. In contrast, the ConSpec definitions express consistency models in terms of constraints on the ordering of the operations observed from the viewpoint of the client, and the underlying storage system is a black box. This allows for a more implementation-agnostic generalization of the CAP theorem.

There have been several research papers on the verification of the consistency of protocol implementations [34, 35]. Consistency of a protocol is verified by checking all possible execution traces that can be generated from the execution of an implementation of that protocol [36]. In contrast, we are not

tied to a particular implementation nor a particular consistency definition, but instead we attempt to provide tools that are generic both in terms of the consistency semantics and the implementation that generates that traces.

III. Basic Definitions

We assume a set of client processes that interface the storage system by invoking operations. An operation is a pair comprising an invocation and the respective response. We denote the set of possible operations on the storage system as $\mathcal{O}$, and therefore $o \in \mathcal{O} = \langle \textit{invocation}, \textit{response} \rangle$. Invocations (resp. responses) belong to a generic set of possible invocations $\mathcal{I}$ (resp. responses $\mathcal{R}$).

A *session trace* st is a sequence of operations executed by the same client, ordered by the time when they were invoked. In this paper we assume that clients are well-formed, i.e., a client only invokes an operation after the preceding operation has returned its response value. As such, session traces can be modelled as sequences of elements of $\mathcal{O}$.

We define a session invocation trace s_{it} as the sequence of invocations that are obtained from transforming each element in a session trace using the projection operator to obtain only the invocations. We define a session invocation trace to be compatible with a session trace if the projection of the invocations in the session trace matches the session invocation trace (denoted $st \bowtie s_{it}$).

The *global session trace* $\mathcal{S}_t$ (resp. global session invocation trace $\mathcal{S}_{it}$) denotes the set of all session traces (resp. session invocation traces) in a given execution of the system.

A very large class of consistency definitions (namely those describing the behavior of loads and stores on computer hardware, and get and put operations on key-value stores) assume that these operations are partitioned into two classes, namely read operations that do not affect the result of subsequent operations and write operations that do. Given that many of the consistency levels we describe require this interface, we also assume that the interface consists of these two operations. Note, however, that this does not lose generality, since other models also distinguish between commands that change versus those that only read the state of the system. For example, databases make a similar distinction between queries and updates, and the state machine replication model distinguishes between read-only and read-write requests.

Similarly, many consistency definitions reason about an interface that exposes the existence of multiple objects (e.g., different memory addresses seen by a CPU or different keys in a key-value store). As such, we assume the interface allows the programmer to specify an object x associated with reads and writes. This does not lose generality since we can eliminate this by restricting the system to a single object.

We denote the set of all objects in a storage system as $\mathcal{X}$. In our notation, we denote an invocation of a write operation that writes a value v to an object x, as $w(x, v)$ (with an empty response). Conversely, a read operation on object x that outputs a value v' is denoted $r(x)v'$. In our formulas, variables denote operations that occur in a session trace. Instead of defining predicates over these variables that restrict the type of operation or the corresponding parameters (e.g., a predicate `isWrite(A)` to mean that variable A is a write operation), we use a shorthand notation W^x_{st} to denote a variable that refers to a write operation to object x that is part of session trace st. In turn, the notation R^x_{st} denotes a read operation to object x in session trace st. We also use the variable representation O^x_{st} to denote a read or write operation over object x that occurs in session trace st.

We use LTL to describe the necessary restrictions among operations. In particular, the LTL operator globally $\Box$ is used to express a condition that must hold across the entire trace; and the LTL operator eventually $\Diamond$ expresses that a condition must eventually hold at some point in a given session trace. For instance the LTL formula

$$\Box \left(R^x_{st} \rightarrow \Diamond W^{x'}_{st} \right)$$

is satisfied by any session trace st where, if a read operation on object x is issued, then it must be eventually followed by a write operation on object x'.

Finally, we assume the system has a sequential specification, corresponding to the output of the operations in a centralized system that executes operations in a sequence, one at a time. In the case of the sequential specification of a system whose interface is comprised of read and write operations, the read operation to object x must output the value associated with the most recent write operation to the same object x. For other types of interfaces (e.g., in state machine replication), that specification is specific to the service interface. This assumption implies that we cannot define specifications where concurrent operations lead to a result that would not be possible in a sequential execution. This decision has the advantage of simplifying our definitions.

While our consistency definitions are completely agnostic of implementation-level concepts, for our proofs regarding the characteristics of the implementation of a certain consistency level (namely in our restatement of the CAP theorem), we need to consider the protocols that implement the consistency level. As such, in that part of the paper only, we model each process as a deterministic state machine, whose transitions can be triggered either by an external input (i.e., an operation invocation) or by receiving a message from the network, and where the transition can trigger sending messages and/or issuing outputs (i.e., an operation response). Processes are connected by unreliable asynchronous communication channels. Note that the assumptions in this paragraph are only used in Section VII, and they are inevitable in this case since the generalized CAP proof concerns implementation characteristics. Therefore, this does not contradict our claim that ConSpec is implementation-agnostic.

IV. ConSpec

In this section we present our generic ConSpec definition, which can be parameterized to obtain specifications for commonly used consistency models.

ConSpec builds on recent proposals for consistency definitions that treat a subset of the operations in a given trace differently (e.g., operations labeled as being "strongly consistent"), since they only enforce visibility among those specific operations [2, 3, 4]. We can similarly see different consistency definitions as enforcing different visibility relationships only among a subset of the system operations, often depending on their types (e.g., reads versus writes).

As such, the generic definition of ConSpec requires the existence of a partial order that intuitively forms a "visibility graph", i.e., the output of each operation must reflect the effects of the operations that precede it according to that partial order. This allows us to see different consistency models as imposing different restrictive conditions on this precedence. Such a restrictive condition is then expressed as an LTL expression E^s.

Definition 1: Generalized form of ConSpec: Given a global session trace $\mathcal{S}_t$, we say that $\mathcal{S}_t$ satisfies a consistency model $\mathcal{C}$ if there exists a partial order $(\mathcal{O}_{St}, \preccurlyeq)$ over the set $\mathcal{O}_{St}$ comprising operations present in all session traces in $\mathcal{S}_t$, i.e., $\mathcal{O}_{St} = \bigcup_{st \in St}\{o \mid o \in st\}$, such that 1) for every operation o in $\mathcal{O}_{St}$, its output is equal to the one obtained by executing the sequential specification of an equivalent re-arrangement (i.e., permutation) of the operations preceding o in $\preccurlyeq$, and 2) $(\mathcal{O}_{St}, \preccurlyeq)$ obeys E^S_C, which is an LTL expression restricting $(\mathcal{O}_{St}, \preccurlyeq)$.

Condition 1, when applied to a system whose interface consists only of reads and writes, translates to a requirement that every read operation in $\mathcal{O}_{St}$ must return the value of the most recent write according to $\preccurlyeq$. Note that in the case of two or more concurrent preceding writes, since any equivalent re-arrangement of the operations is valid, the system may arbitrate any order for them. Condition 2, in turn, can be expressed as $E^S_C \models (\mathcal{O}_{St}, \preccurlyeq)$, where E^S_C is the ConSpec parameterization for each consistency model $\mathcal{C}$, and $\models$ is the satisfies operator. E^S_C can refer to any characteristic of the global session trace (e.g., impose restrictions based on the ordering of operations within sessions).

A noteworthy choice that was made when producing the above definition is that consistency models are expressed in terms of partial orders among operations, instead of using more general relations [28]. This choice entails a tradeoff between generality (by allowing for arbitrary relations) and simplicity (by producing more concise definitions using partial orders). We chose to have simpler definitions using partial orders for two main reasons. First, this is aligned with our goals of enabling a better understanding of consistency through simpler definitions, and second, the protocols that guarantee the extra properties required by partial orders (namely transitivity) are well studied and inexpensive in terms of their runtime overhead, namely since they do not require cross-replica synchronization [37, 38].

Another discussion point worth highlighting is the fact that, when concurrent branches of an execution merge, our definitions state that the output that is produced by any re-arrangement of the partial order of prior requests is admissible. In practice, this corresponds to saying that our merge policy is to serialize concurrent requests in an arbitrary order and re-execute them. Again, it would have been straightforward to add complexity to our definitions in order to allow for other merge strategies. For example, the "last writer wins" policy [39] is a special case of our more general definition, and it would be relatively easy to constrain the definition to always arbitrate an order that is compatible with LWW. However, since our goal is to simplify the definitions, we decided to leave this arbitration out.

V. Specifying existing models

In this section, we present the ConSpec specifications for a few example common consistency models. We have defined several other models using ConSpec, but, due to space limitations, we defer their presentation to a longer version of this document.

A. Causal Consistency

First, we present the following definition for Causal consistency, which is widely used in recent distributed storage systems.

$$E^S_C = \begin{array}{l} \forall x, y \in \mathcal{X}, st \in \mathcal{S}_t, O^x_{st}, O^y_{st} \in st : \\ (\square\,(O^x_{st} \rightarrow \Diamond O^y_{st}) \rightarrow O^x_{st} \preccurlyeq O^y_{st})\,, \end{array} \quad (1)$$

The above expression simply specifies that, for all session traces in a global session trace $\mathcal{S}_t$, there must exist a common valid partial order $\preccurlyeq$ that orders operations according to the precedence relation of those operations in the respective session trace. Note that this definition is simplified by the fact that the generic ConSpec construction already encodes an important aspect of Causal consistency, which is the fact that ConSpec requires the operations to form a partial order. Given the transitivity property of partial orders, this implies that if operation b sees the effects of a and operations c sees the effects of b, then c must also see the effects of a.

From this definition we can also derive the various session guarantees [40], namely Read Your Writes, Writes Follow Reads, Monotonic Writes and Monotonic Reads. These are essentially special cases of the above definition, which restrict the types of the pairs of operations bound by $\preccurlyeq$. Again, we defer a formal presentation to a longer version of this document.

Examples. To analyze a violation of causal consistency under the read-write interface, we consider the following global session trace comprising three session traces:

st_1: $w(x, 1)$,
st_2: $r(x)1, w'(x, 2)$,
and st_3: $r'(x)2, r''(x)1$.

The list of valid linear extensions of partial orders, comprising all operations in $\mathcal{S}_t$, that satisfy Condition 1 in Definition 1 are given as follows.

- $\preccurlyeq^1 = W^x_{st} \preccurlyeq R^x_{st} \preccurlyeq R''^x_{st} \preccurlyeq W'^x_{st} \preccurlyeq R'^x_{st}$
- $\preccurlyeq^2 = W^x_{st} \preccurlyeq R''^x_{st} \preccurlyeq R^x_{st} \preccurlyeq W'^x_{st} \preccurlyeq R'^x_{st}$
- $\preccurlyeq^3 = W'^x_{st} \preccurlyeq R'^x_{st} \preccurlyeq W^x_{st} \preccurlyeq R^x_{st} \preccurlyeq R''^x_{st}$

- $\preccurlyeq^4 = W'^x_{st} \preccurlyeq R'^x_{st} \preccurlyeq W^x_{st} \preccurlyeq R''^x_{st} \preccurlyeq R^x_{st}$

Since the session trace st_1 comprises a single write it vacuously satisfies the Causal consistency condition in Equation 1, given that this condition constrains pairs of operations in the trace. Next, we consider the session trace st_2 in $\mathcal{S}_t$, comprising operations $r(x)1$ and $w'(x,2)$. st_2 only matches the LTL condition $\Box O^x_{st} \rightarrow \Diamond O'^x_{st}$ for $O^x_{st} = r(x)1$ and $O'^x_{st} = w'(x,2)$. From the above list of valid partial orders, only $\preccurlyeq^1$ and $\preccurlyeq^2$ satisfy the condition $O^x_{st} \preccurlyeq O'^x_{st}$ where $O^x_{st} = r(x)1$ and $O'^x_{st} = w'(x,2)$. Thus, st_2 reduces the set of valid linear extensions of partial orders to the set $\{\preccurlyeq^1, \preccurlyeq^2\}$. Finally, let us consider the session trace st_3 in $\mathcal{S}_t$, comprising operations $w'(x,2)$ and $r''(x)1$. In this case, st_3 matches the LTL condition $\Box O'^x_{st} \rightarrow \Diamond O''^x_{st}$ for $O'^x_{st} = w'(x,2)$ and $O''^x_{st} = r''(x)1$. From the list above, the only valid orders that satisfy the condition $O'^x_{st} \preccurlyeq O''^x_{st}$ are $\preccurlyeq^3$ and $\preccurlyeq^4$. Putting together the previous requirements, we can see that only the orders $\preccurlyeq^1$ and $\preccurlyeq^2$ meet the condition in Equation 1) for st_2, whereas only $\preccurlyeq^3$ and $\preccurlyeq^4$ meet the same condition for st_3. Thus, there does not exist a single valid partial order that satisfies the causal consistency condition for all the session traces in $\mathcal{S}_t$.

As an example that obeys causal consistency – but fails to meet stronger conditions such as sequential consistency or linearizability – consider a global session trace comprising the following session traces:

st'_1: $w(x,1)$,
st'_2: $w'(x,2)$,
st'_3: $r(x)1, r'(x)2$,
and st'_4: $r''(x)2, r'''(x)1$

In this case, the following partial order $\preccurlyeq$ can be used in our ConSpec formula.

- $W^x_{st} \preccurlyeq R^x_{st}$
- $W'^x_{st} \preccurlyeq R'^x_{st}$
- $R^x_{st} \preccurlyeq R'^x_{st}$
- $W'^x_{st} \preccurlyeq R''^x_{st}$
- $W^x_{st} \preccurlyeq R'''^x_{st}$
- $R''^x_{st} \preccurlyeq R'''^x_{st}$

Thus the above global session trace satisfies Causal consistency. Note, however, that $\preccurlyeq$ is not a total order – and in fact there is no total order that can be used in this case, which, as we will see, explains why this trace does not meet strong consistency specifications.

B. Sequential Consistency

Our second example definition is Sequential consistency (SC), which is a strong consistency model that enforces a global ordering among operations executed from concurrent client processes. SC requires that a global execution comprising operations executed from one or more clients must be equivalent to the result of executing the operations in a sequential order, such that the order among operations from each client in that sequence matches the invocation order of the operations. Hence, there must exist a valid partial order set $(\mathcal{O}_{St}, \preccurlyeq)$ comprising all operations in a given global session trace $\mathcal{S}_t$ that satisfies the following condition. The precedence order among every pair of operations O^x_{st} and O'^y_{st} in $\preccurlyeq$ must follow the precedence order among O^x_{st} and O'^y_{st} in each session trace st comprised in $\mathcal{S}_t$. This condition can be given as $\Box\left(O^x_{st} \rightarrow \Diamond O'^y_{st}\right) \rightarrow O^x_{st} \preccurlyeq O'^y_{st}$. Additionally, the result of every pair of operations must be equivalent to that of performing a sequential execution of the operations, i.e., there must exist an equivalent total order among all the operations. Combining the above conditions, SC can be expressed as follows.

$$\begin{aligned} E^S_C = \forall x, y \in \mathcal{X}, st \in \mathcal{S}_t, O^x_{st}, O'^y_{st} \in st : \\ \left(\Box\left(O^x_{st} \rightarrow \Diamond O'^y_{st}\right) \rightarrow O^x_{st} \preccurlyeq O'^y_{st} \wedge \right. \\ \left. \forall O'', O''' \in \mathcal{S}_t : O'' \preccurlyeq O''' \vee O''' \preccurlyeq O''\right). \end{aligned} \tag{2}$$

Examples. Let us consider a global session trace comprising the following two sessions, each with three operations (two writes followed by a read).

st''_1: $w(x,1), w'(x,99), r(y)1$, and
st''_2: $w(y,1), w'(y,99), r(x)1$

This global session trace meets Causal consistency but is not sequentiallly consistent. In particular, for sequential consistency, the constraints in E^S_C require the partial order to be a total order, which is impossible to achieve while also obeying the session order and explaining the results that are observed according to the sequential specification of a read/write interface. This is because one would have to serialize both reads before the respective writes of value 99, but that would be impossible to achieve in a total order that respects the session orders. Therefore, we can conclude that there does not exist a valid partial order that satisfies Equation 2 for all write-read pairs in st_1, which means that the SC session guarantee is not upheld.

VI. Rewriting the CAP Theorem in terms of ConSpec

The CAP conjecture was initially stated informally as the impossibility of simultaneously achieving strong Consistency, Availability, and Partition tolerance in a replicated system [6]. When this was subsequently proven by Gilbert and Lynch [7], these three properties were stated precisely, and, in this context, strong consistency was defined as atomicity (or linearizability [10]).

The fact that the original proof of CAP is restricted to linearizability raises the question of whether CAP holds using other definitions of consistency models. In this section, we rewrite the CAP theorem in terms of ConSpec to precisely define the class of consistency models that can and cannot be implemented in a highly-available, partition-tolerant way.

To begin with, we need a helper definition to enumerate all admissible partial orders for a given restriction condition E^S_C and set of operations in a global session invocation trace $\mathcal{S}_{it}$.

Definition 2 (Partial order enumeration): Given a global session invocation trace $\mathcal{S}_{it}$ and a restrictive condition E^S_C for a ConSpec definition, we define the partial order enumeration of this session invocation trace and condition, $\Pi(\mathcal{S}_{it}, E^S_C)$, as the set of partial orders over the elements of any compatible global session trace $\mathcal{S}_t$ that are valid under E^S_C, i.e.:

$\Pi(\mathcal{S}_{it}, E_C^S) \equiv \{\preccurlyeq: \exists \mathcal{S}_t \left(E_C^S \vDash (\mathcal{S}_t, \preccurlyeq) \wedge \mathcal{S}_t \bowtie \mathcal{S}_{it}\right)\}$

This allows us to define the following necessary and sufficient condition for a consistency model to have an available and partition tolerant implementation.

Theorem 6.1 (Generalized CAP theorem): In an asynchronous system, it is possible to implement a consistency model E_C^S while simultaneously providing availability and partition tolerance if and only if, for any global session invocation trace $\mathcal{S}_{it}$ and all of its partial orderings that are allowed by E_C^S, when you consider the set of maxima of each partial order, it is always possible to make them depend only on the previous operation in the same session and still obtain a valid partial order, i.e., the following holds:

$\forall \mathcal{S}_{it}\ \forall \preccurlyeq \in \Pi(\mathcal{S}_{it}, E_C^S)\ \forall o \in max(\preccurlyeq)$
$(\textsc{RemoveAllExceptSession}(\preccurlyeq, o) \in \Pi(\mathcal{S}_{it}, E_C^S))$

where we define RemoveAllExceptSession as a partial order where the maximum o is only directly ordered after prior operations in the same session, i.e.:

$\textsc{RemoveAllExceptSession}(\preccurlyeq, o(x)) \equiv \preccurlyeq \setminus \{\langle o'(x), o(x)\rangle$, where $\langle o'(x), o(x)\rangle$ belongs to the transitive reduction of $\preccurlyeq$, and $o'(x)$ does not belong to the same session as $o(x)\}$.

Proof: We start by proving the implication in the direction ($\Rightarrow$). Following the proof style of Gilbert et al. [7], we prove this by contradiction as follows. Let us assume, by contradiction, that consistency model E_C^S is implemented by an algorithm that is highly available during partitions but does not meet the condition at the end of the Theorem. Let us consider initially that there are only two clients, c_1 and c_2, with sessions $s1$ and $s2$, respectively. The fact that E_C^S does not meet this condition means, given the definition of partial order enumeration and its use in the extended CAP theorem, that it is possible to produce an execution corresponding to a global session trace $\mathcal{S}_t$ with a valid partial order $\preccurlyeq$ that has a maximum element $o_{s1}(x)$ such that $o_{s2}(x) \preccurlyeq o_{s1}(x)$ (where $o_{s1}(x), o_{s2}(x)$ belong to $s1$ and $s2$), and where it is not admissible to have a partial order where $o_{s2}(x) \not\preccurlyeq o_{s1}(x)$.

Now let us construct the following execution. First, we run the system under the exact same conditions that produced $\mathcal{S}_t$ until client c_1 is about to execute $o_{s1}(x)$ and client c_2 is about to execute $o_{s2}(x)$. At this point, a partition occurs that separates c_1 and c_2, which persists until the end of the execution. By the availability and partition-tolerance properties, the operations $o_{s1}(x)$ and $o_{s2}(x)$ will eventually complete and, by our initial assumption in the previous paragraph, the former operation must see the effects of the latter, i.e., the partial order that supports that execution must be such that $o_{s2}(x) \preccurlyeq o_{s1}(x)$. Now run the exact same execution, but where the client c_2 crashes right before invoking $o_{s2}(x)$. This execution is indistinguishable from the previous one from the standpoint of c_1. Thus c_1 will follow the same sequence of states and produce the same outputs as in the previous execution. This would mean that the algorithm would not meet its ConSpec specification, since $o_{s1}(x)$ would reflect the execution of an operation that was not part of the global session trace $\mathcal{S}_t$, namely $o_{s2}(x)$. This contradicts the fact that the employed algorithm meets that specification and the CAP properties.

The assumption about there being only two clients does not lose generality because, with more clients, a pair of clients c_1, c_2 under the conditions above must also exist. Then the proof generalizes beyond two clients by partitioning the clients into two sets, one containing c_1 and another containing c_2, and crashing all the clients in the same side of the partition as c_2.

Next we focus on the implication in the direction $\Leftarrow$. Here, we need to prove that if a consistency model E_C^S meets the condition:

$\forall \mathcal{S}_{it}\ \forall \preccurlyeq \in \Pi(\mathcal{S}_{it}, E_C^S)\ \forall o(x) \in max(\mathcal{S}_{it}, \preccurlyeq)$
$(\textsc{RemoveAllExceptSession}(\preccurlyeq, o(x)) \in \Pi(\mathcal{S}_{it}, E_C^S))$

then it has an available and partition-tolerant implementation.

Given any global session invocation trace $\mathcal{S}_{it}$, we prove this by induction on the length of the execution that produced $\mathcal{S}_{it}$. The base case with an empty execution is vacuously true, since an empty trace meets any consistency condition (no safety properties are ever violated by an empty trace). For the induction step, we need to prove that, given an execution for which an available and partition-tolerant implementation produced a trace that conforms to E_C^S, it is possible for a client to invoke a new operation and produce an output that is also consistent. This is true because, even in the case that the client that invokes the new operation is partitioned from all or a subset of the remaining processes, the consistency model allows for the newly invoked operation to depend only on prior operations from the same session and all the operations that transitively precede them according to $\preccurlyeq$. (This is because, if that was not the case, then this would invalidate the hypothesis on the right hand side of the equivalence stated in Theorem, since a valid partial order would not meet the property that removing all but the edges in the same session would be a part of the partial order enumeration for E_C^S.) Furthermore, the valid output of this operation can be determined by using only information that is local to the session, by running the sequential specification of the system on the graph of preceding operations. □

VII. Analysis of Consistency Models With Respect to CAP

The previous section defined necessary and sufficient conditions for a consistency model $\mathcal{C}$ to have an available and partition-tolerant implementation. Now, we analyze the E_C^S-expression for the consistency models that we presented in Section V, to determine how they fare with respect to Theorem 6.1.

We observe that the definition of Causal consistency only forces constraints on the partial ordering across operations from the same session. This implies that these constraints are compatible with the conditions on the right hand side of the equivalence of Theorem 6.1. In particular, it is the case that it is always legal to remove orderings between operations across sessions, since these are never constrained by the implications in the E_C^S expression. Therefore, we conclude that Causal

$mtype = \{r, w, x, y\}$;
$typedef\, Op$ {
$mtype\ optype$;
$mtype\ var$;
$int\ val$;}
$typedef\ \ PO$ {
$Op\ st[max_{size}]$;
$mtype\ status$;}
$Op\ \ \ st[size]$;
$Op\ \ po[po_{size}]$;
$ltl\ \ cc\ \ \{\Box(\neg(po[i].st[j].optype = w$
$\Rightarrow\ \Diamond po[i].st[j].optype = r))\}$

Fig. 1: Example ConSpec specification in PROMELA

consistency is not affected by CAP, i.e., it can have a highly available and partition-tolerant implementation.

In contrast, SC requires that the visibility order $\preccurlyeq$ among operations from all the clients in the system forms a total order. This implies that if an operation is related by the transitive reduction of $\preccurlyeq$ to a previous operation from another session, it is not possible to remove this element of the partial order and still obtain a valid partial order, since it would violate the condition in the definition of SC that any two operations need to be ordered with respect to each other. Thus, this does not meet the necessary and sufficient condition for a partition-tolerant, highly available implementation.

VIII. Implementation

We provide ConsSpecCheck, an open source automated tool for verifying whether a session trace meets a given consistency model. This tool was built using Spin [41], a widely used open source software verification framework. The source code of the ConsSpecCheck tool along with instructions for running it are publicly available at https://github.com/ssidhanta/ConSpecTool. A global session trace is supplied to the tool as input, in the form of a text file containing a sequence of storage operations as a series of rows of comma separated strings. The ConSpec definition E_C^S of a consistency model $\mathcal{C}$ is expressed as a Spin LTL formula modelled using the PROMELA meta language. Internally, Spin translates the PROMELA source file into C code. The Spin driver then runs the built-in model checker to check for counter-examples for the above generated C code against the Spin formula for the definition E_C^S.

The snippet in Figure 1 shows an example PROMELA source file for the RYW consistency model. In the above snippet, cc is a Spin-style declaration of the E_C^S definition for this particular consistency model. Each operation in a trace is represented as a tuple Op comprising the following elements: the operation type $optype$, the variable name var, and the value val. Then, using the statement $Op\ \ st[size]$, a session trace of length $size$ is declared as an array st of elements belonging to the user-defined datatype Op. A valid partial order for a given session trace is declared using the typedef declaration $typedef\ \ PO$ ... The list of all possible valid partial orders for a session trace st is declared as an array po, which is created by the statement $PO\ \ po[po_{size}]$; this creates an array of elements belonging to the type PO with an array-size of po_{size}. Each element in the array po is a permutation of the elements in the array st. The i'th element in the list of all possible partial orders po is accessed as $po[i]$, and the j'th operation in $po[i]$ is accessed as $po[i].st[j]$. In the declaration of cc, $po[i].st[j].optype$ denotes the operation type of the j'th operation in the i'th partial order. $\Box$ is the LTL operator globally, $\Diamond$ is the LTL operator eventually, $\neg$ is the negation operator, and $\Rightarrow$ denotes the implies operator. Thus, the declaration cc in the above snippet denotes that if the operation type *optype* of an operation $st[j]$ in a given session trace st is write, then the next read operation, which follows $st[j]$ in the same session trace and reads an object that was written by $st[j]$, must return the value that was written by $st[j]$. The above specification is provided in the pml file, as a declarative statement, followed by the specification of the system behavior, namely the semantics of read and write operations. The pml file is then compiled using the Spinroot compiler with the above LTL specification as the spin invariant. This produces a C++ program, whose output flags if a given input session trace is a case of violation or satisfies the consistency model specified in the LTL invariant given in the above snippet.

IX. Evaluation

In this section, we evaluate how long the ConSpec tool takes to check the consistency of a session trace, how this validation time varies depending on the length of the trace, and that compares to checking traces expressed in other syntaxes. In our evaluation, we use two sets of traces, where the first one is generated by executing the Yahoo Cloud Serving Benchmark (YCSB) benchmark suite (YCSB v 0.1.4) [42] on top of a Cassandra cluster (Apache Cassandra v 2.1.2) [15], and the second one is obtained by executing the TPC-C benchmark on top of a MySQL database. The ConSpec Tool was run over the above traces on an Apple MacBook Pro, with 8 GB 1600 MHz DDR3 RAM, 2.9 GHz Intel Core i7 processor, running MacOS Sierra v10.12.4. The partial order generator component of the tool was run on Java 1.8.0_121, and the PROMELA component were compiled and run on Spin v6.4.6. Throughout the evaluation, by execution time we refer to the time duration beginning with the preprocessing of the input global session trace and ending with printing the result of checking the above trace against the given ConSpec specification.

A. Evaluation with YCSB

In this part of the evaluation, we use YCSB to evaluate the performance of our verification tool against cloud workloads. To understand how the execution time of the ConSpec tool varies with the length of the global session trace, we plot the total execution time that the tool incurs as a function of the size of the global session trace measured as the number of operations in that trace. To generate a series of global session traces of different lengths, we are able to vary two configuration parameters of YCSB, namely the thread count and the execution time. Using the thread count parameter,

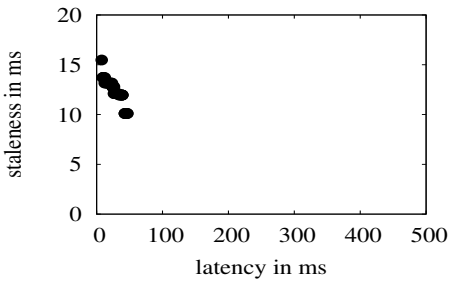

Fig. 2: Staleness in Gamma vs Latency with YCSB.

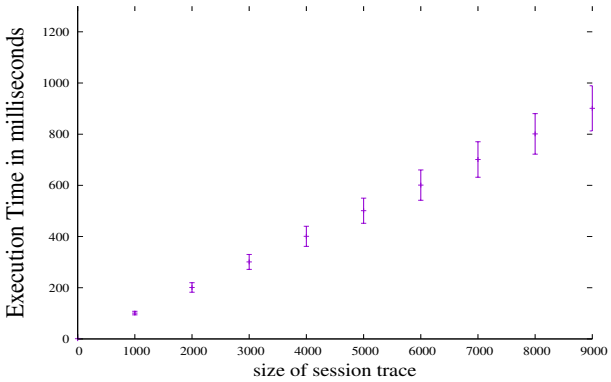

Fig. 3: Execution Time of ConSpec Tool Against YCSB Traces.

we simulated a number of concurrent YCSB client threads executing the given workload, where the number of clients corresponds to the value passed to this parameter. Thus, each execution of the YCSB client with a given value of the thread parameter generates a global session trace consisting of multiple session traces, where each session trace comprises the entire sequence of operations performed from a specific client thread. To record each trace, we modify the source code of the YCSB Java client to record the per-operation execution wall-clock time, the operation type (read or write), the keys that were accessed and the respective values returned. In this experiment, we verify the YCSB session trace against the ConSpec definition of the RYW consistency model, which is specified as a Spin LTL formula in the corresponding PROMELA source file. We execute the YCSB client with 50 concurrent client threads; thus, the global session trace comprises 50 concurrent sessions. Then, to obtain global session traces of an increasing length, we created several instances of an execution of the YCSB client, where, for each execution, we increase the length of the global session trace by increasing the duration of execution of the client while keeping the concurrency level fixed at 50 concurrent threads. Thus, for each execution, the size of the global session trace (i.e., the total number of operations in that trace) is given by $50 \times op$, where op is the number of operations executed by one client thread during that run. We preloaded the YCSB backend with $1,000,000$ records. The "request distribution" parameter is used to indicate to the YCSB client which specific random distribution to use for the keys, which, in turn, forms the basis for choosing the records on which a given operation is to be performed. We ran the client with the "latest" request distribution of keys, where the latest inserted records comprise the head of the distribution. The "target" parameter is used to throttle the target number of operations per second; we set the value of "target" to 100 operations per second. To better understand if the original trace was already close to being linearizable or not, we measured the level of deviation from strong consistency using a metric called Γ proposed by Golab et al. [43]. Intuitively this metric captures both the proportion and severity of stale results. The results of applying this metric to the data that is produced by our trace are depicted in Figure 2. This shows that the values of 95th percentile Γ score range from 0 to 14 ms, which implies that the severity of observed staleness in our results is small.

Figure 3 depicts the variation of the execution time of ConSpec on global session traces collected from YCSB, as we vary the length of the traces, i.e., the number of operations performed, which is plotted along the x axis. We repeated the verification of each global session trace 5 times, and in each case we report the average and, as error bars, the standard deviation of the execution times observed in the 5 runs. The execution time remains within 1 second for global session traces of size up to $100K$, i.e., comprising up to $100,000$ operations. Furthermore, the straight line appearance of the plot indicates that the performance of ConSpec scales approximately linearly with the size of the global session trace.

This linear scalability was somewhat surprising, since the preprocessing step in ConSpec for generating all possible legal serializations for a given session trace has a complexity $\mathcal{O}(l!)$, where l is the length of the session trace. To understand this performance trend, we note that the ConSpec tool performs the verification of a given session trace with the Spin model checker, which reduces the verification problem into a graph reachability problem; it internally constructs a graph of all possible states and subsequently determines reachable states in the graph [44]. Next, the Spin model checker employs the nested depth first search algorithm [45] to search for accepting states within the state space comprising all possible reachable states [46]. While performing the above task, the Spin model checker applies partial order reduction techniques [47], which result in reduction of the search space of reachable states explored during the nested depth first search. In [44], Holzmann et al. provide a complexity analysis of the Spin tool in performing verification of the well known leader election algorithm. Holzmann et al. observe a linear growth in the number of reachable states explored by the Spin model checking algorithm instead of an expected exponential growth. They also report a similar linear growth with other typical use cases of the model checker. As such, this is a plausible explaination for the results we obtained.

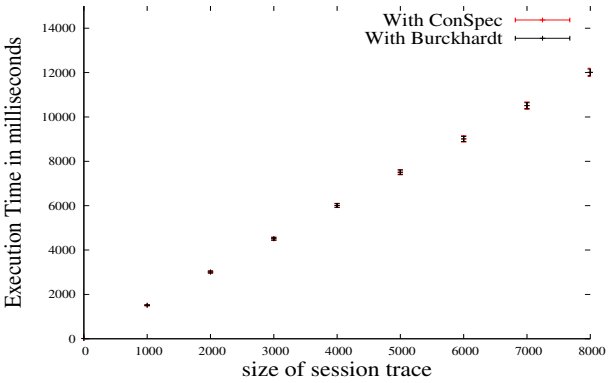

Fig. 4: Execution Time of ConSpec Tool Against TPCC Trace With ConSpec Definitions vs Burckhardt's Definitions.

B. Evaluation with TPC-C

Analogously to the previous set of experiments, we also ran the TPC-C benchmark for runs of increasing length, and collected global session traces from each run of the TPC-C client. We used an open source java implementation of the TPC-C benchmark [48], and modified the code to record the type of operation (insert and update queries were marked as write operations, and select queries were marked as read operations), database column, and updated values for each TPC-C operation. We ran the TPC-C client on a single machine with a single node MySQL server in the backend. We configured TPC-C with a connection pool of size 30, ramp up time of 30 seconds, and a single warehouse. Before running the TPC-C "run" commands, we loaded the TPC-C database tables by running the TPC-C "load" command for a period of 60 seconds. Subsequently, during each experiment, the TPC-C "run" command was executed repeatedly, with a duration of 60 seconds for each command, and the global session trace for each experiment was recorded into the TPC-C log files. The size of the log files was gradually increased by increasing the execution time of the TPC-C run command, which, in turn, increased size of the global session trace. We ran the ConSpec tool against the global session traces that were extracted from the TPC-C log files, where the number of operations performed and the execution time is plotted along the x and y axes, respectively. TPC-C disallows serialization anomalies and therefore we ran the ConSpec tool with strict serializability. The red points in Figure 4 depict the execution time averaged over the five runs, with the red error bars indicating the standard deviations. The results show that the performance variations of the tool with the size of the trace follows a pattern that is very similar to the one obtained using YCSB. This suggests that the performance is not very sensitive to the the application and the workloads it generates.

C. Comparison to other definitions

In the final part of the evaluation, we compare ConSpec against a baseline. Since ConSpec is a new language for defining consistency models, we compared it to an existing definition running exactly the same underlying verification system. The definition we chose was based on the work of Burckhardt [49], which we directly encoded in Promela so that we could verify global session traces against this definition using Spin. Then we compared the time it takes to verify the same traces against these prior definitions from Burckhardt and against the corresponding ConSpec definition. For both cases, we chose the RYW definition (referred to as read my writes in Burckhardt et al. [28]). For this comparison, we reused the global session traces collected from TPC-C benchmark experiments in Section IX-B.

Figure 4 shows the results of this comparison, where the results again correspond to the average of 5 runs, and the error bars represent one standard deviation. From Figure 4, we can observe the verification time is slightly slower with ConSpec than with the previous definition. Furthermore, the verification time shows comparable scalability with both definitions. We conclude that, with ConSpec definitions, the tool performs comparably with respect to other existing definition, and that the scalability seems close to linear in both cases.

X. Conclusions

In this paper, we presented a generic framework called ConSpec for defining consistency. ConSpec enables definitions that are precise, follow a generic structure, and are independent of implementation details. We used ConSpec to derive several concrete definitions of existing consistency levels. Furthermore, ConSpec also enabled a generic version of the CAP theorem, where the "C" property is no longer tied to a specific strong consistency definition. Instead, we define necessary and sufficient conditions for a consistency level to be within the scope of CAP, i.e., for the existence or not of a partition tolerant and available implementation. Finally, we developed and evaluated an automated tool for verifying whether a given session trace satisfies a consistency model.

ConSpec opens several interesting avenues for future work. First, we intend to apply ConSpec to a wider range of consistency models. Second, we intend to extend it to support isolation levels of transactional systems, where the visibility of individual operations within a transaction must be constrained. Finally, we intend to further develop our automatic verification tool and promote its adoption by the developer community.

Acknowledgements

This work was supported by the European Research Council ERC-2012-StG-307732 and by FCT/MCTES project UID/CEC/50021/2019.

References

[1] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Feral concurrency control: An empirical investigation of modern application integrity," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15.

[2] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues, "Making geo-replicated systems fast as possible, consistent when necessary," in *Proc. of the 10th USENIX conference on Operating Systems Design and Implementation*, ser. OSDI'12.

[3] A. Gotsman, H. Yang, C. Ferreira, M. Najafzadeh, and M. Shapiro, "'Cause I'm Strong Enough: Reasoning About Consistency Choices

in Distributed Systems," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '16.

[4] C. Li, J. a. Leitão, A. Clement, N. Preguiça, and R. Rodrigues, "Minimizing coordination in replicated systems," in *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data*, ser. PaPoC '15. ACM, 2015.

[5] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, 1977.

[6] E. A. Brewer, "Towards robust distributed systems (Invited Talk)," in *Proc. of the 19th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 2000.

[7] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, no. 2, pp. 51–59, Jun. 2002. [Online]. Available: http://doi.acm.org/10.1145/564585.564601

[8] M. Musuvathi, "Systematic concurrency testing using chess," in *Proceedings of the 6th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, ser. PADTAD '08.

[9] H. Attiya and J. L. Welch, "Sequential consistency versus linearizability," *ACM Trans. Comput. Syst.*, vol. 12, no. 2, May 1994.

[10] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, Jul. 1990.

[11] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Comput.*, vol. 28, no. 9, pp. 690–691, Sep. 1979.

[12] R. Lipton and J. S. Sandberg, "PRAM: a scalable shared memory," Princeton University (NJ US), Tech. Rep. CS-TR-180-88, 1988. [Online]. Available: http://opac.inria.fr/record=b1024856

[13] M. Mizuno, M. Raynal, and J. Z. Zhou, "Sequential consistency in distributed systems," in *Theory and Practice in Distributed Systems*, K. P. Birman, F. Mattern, and A. Schiper, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 224–241.

[14] M. Raynal and A. Schiper, "From causal consistency to sequential consistency in shared memory systems," in *Proceedings of the 15th Conference on Foundations of Software Technology and Theoretical Computer Science*, 1995.

[15] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010.

[16] R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman, and S. Shah, "Serving large-scale batch computed data with project Voldemort," in *Proc. of the 10th USENIX Conference on File and Storage Technologies (FAST)*, 2012.

[17] C. Meiklejohn, "Riak PG: Distributed process groups on dynamo-style distributed storage," in *Proc. of the Twelfth ACM SIGPLAN Workshop on Erlang*, ser. Erlang '13, pp. 27–32.

[18] E. Plugge, T. Hawkins, and P. Membrey, *The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing*, 1st ed. Berkeley, CA, USA: Apress, 2010.

[19] T. Schütt, F. Schintke, and A. Reinefeld, "Scalaris: Reliable transactional P2P key/value store," in *Proceedings of the 7th ACM SIGPLAN Workshop on ERLANG*, ser. ERLANG '08.

[20] 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," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 205–220, Oct. 2007.

[21] D. Inc., "Configuring data consistency," http://docs.datastax.com/en/archived/cassandra/2.0/cassandra/dml/, 2017.

[22] A. Adya, B. Liskov, and P. E. O'Neil, "Generalized isolation level definitions," in *Proceedings of the 16th International Conference on Data Engineering*, ser. ICDE '00, 2000, pp. 67–78.

[23] 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," in *Proc. ACM Symposium on Operating Systems Principles (SOSP)*, ser. SOSP '95, 1995, pp. 172–182.

[24] A. Szekeres and I. Zhang, "Making consistency more consistent: A unified model for coherence, consistency and isolation," in *Proceedings of the 5th Workshop on the Principles and Practice of Consistency for Distributed Data*, ser. PaPoC '18, 2018.

[25] G. Chockler, R. Friedman, and R. Vitenberg, "Consistency conditions for a CORBA caching service," in *Proceedings of the 14th International Conference on Distributed Computing*, ser. DISC '00, 2000.

[26] S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski, "Replicated data types: Specification, verification, optimality," in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '14.

[27] S. Burckhardt, A. Gotsman, and H. Yang, "Understanding eventual consistency," Tech. Rep., March 2013.

[28] S. Burckhardt, "Principles of eventual consistency," *Found. Trends Program. Lang.*, vol. 1, no. 1-2, pp. 1–150, Oct. 2014.

[29] A. Cerone, A. Gotsman, and H. Yang, "Algebraic laws for weak consistency," in *28th International Conference on Concurrency Theory, CONCUR 2017*.

[30] M. Weber, A. Bieniusa, and A. Poetzsch-Heffter, "EPTL - A temporal logic for weakly consistent systems (short paper)," in *Formal Techniques for Distributed Objects, Components, and Systems - 37th IFIP WG 6.1 International Conference, FORTE 2017*.

[31] J. Wickerson, M. Batty, T. Sorensen, and G. A. Constantinides, "Automatically comparing memory consistency models," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL 2017.

[32] P. Mahajan, L. Alvisi, and M. Dahlin, "Consistency, availability, convergence," Computer Science Department, University of Texas at Austin, Tech. Rep. TR-11-22, May 2011.

[33] H. Attiya, F. Ellen, and A. Morrison, "Limitations of highly-available eventually-consistent data stores," in *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, ser. PODC '15.

[34] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon, "Implementing a cache consistency protocol," *SIGARCH Comput. Archit. News*, vol. 13, no. 3, pp. 276–283, Jun. 1985.

[35] F. Pong and M. Dubois, "The verification of cache coherence protocols," in *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA '93.

[36] A. E. Condon and A. J. Hu, "Automatable verification of sequential consistency," *Theory of Computing Systems*, vol. 36, no. 5, pp. 431–460, Oct 2003.

[37] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Bolt-on causal consistency," in *Proc. of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13.

[38] 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*, ser. SOSP '11.

[39] R. H. Thomas, "A majority consensus approach to concurrency control for multiple copy databases," *ACM Trans. Database Syst.*, vol. 4, no. 2, pp. 180–209, Jun. 1979.

[40] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. W. Welch, "Session guarantees for weakly consistent replicated data," in *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, ser. PDIS '94.

[41] G. Holzmann, *Spin Model Checker, the: Primer and Reference Manual*, 1st ed. Addison-Wesley Professional, 2003.

[42] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10.

[43] W. Golab, M. R. Rahman, A. Auyoung, K. Keeton, and I. Gupta, "Client-centric benchmarking of eventual consistency for cloud storage systems," in *Proceedings of the 2014 IEEE 34th International Conference on Distributed Computing Systems*, ser. ICDCS '14.

[44] G. J. Holzmann, "The model checker spin," *IEEE Trans. Softw. Eng.*, vol. 23, no. 5, pp. 279–295, May 1997.

[45] G. J. Holzmann, D. A. Peled, and M. Yannakakis, "On nested depth first search," in *The Spin Verification System, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, August, 1996*, 1996, pp. 23–32.

[46] S. Schwoon and J. Esparza, "A note on on-the-fly verification algorithms," in *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005*.

[47] D. Peled, "Combining partial order reductions with on-the-fly model-checking," *Formal Methods in System Design*, vol. 8, no. 1, Jan 1996.

[48] C. Corporation, "Java TPC-C," https://github.com/AgilData/tpcc, 2014.

[49] S. Burckhardt, "Principles of eventual consistency," *Foundations and Trends in Programming Languages*, vol. 1, no. 1-2, pp. 1–150, 2014.