Transaction Chopping for Parallel Snapshot Isolation

Andrea Cerone[1], Alexey Gotsman[1], and Hongseok Yang[2]

[1] IMDEA Software Institute
[2] University of Oxford

Abstract. Modern Internet services often achieve scalability and availability by relying on large-scale distributed databases that provide consistency models for transactions weaker than serialisability. We investigate the classical problem of transaction chopping for a promising consistency model in this class—parallel snapshot isolation (PSI), which weakens the classical snapshot isolation to allow more efficient large-scale implementations. Namely, we propose a criterion for checking when a set of transactions executing on PSI can be chopped into smaller pieces without introducing new behaviours, thus improving efficiency. We find that our criterion is more permissive than the existing one for chopping serialisable transactions. To establish our criterion, we propose a novel declarative specification of PSI that does not refer to implementation-level concepts and, thus, allows reasoning about the behaviour of PSI databases more easily. Our results contribute to building a theory of consistency models for modern large-scale databases.

1 Introduction

Modern Internet services often achieve scalability and availability by relying on databases that replicate data across a large number of nodes and/or a wide geographical span [18, 22, 25]. The database clients can execute transactions on the data at any of the replicas, which communicate changes to each other using message passing. Ideally, we want this distributed system to provide strong guarantees about transaction processing, such as serialisability [9]. Unfortunately, achieving this requires excessive synchronisation among replicas, which increases latency and limits scalability [1, 15]. For this reason, modern large-scale databases often provide weaker consistency models that allow non-serialisable behaviours, called *anomalies*. Recent years have seen a plethora of consistency model proposals that make different trade-offs between consistency and performance [6, 7, 20, 22]. Unfortunately, whereas transactional consistency models have been well-studied in the settings of smaller-scale databases [2, 13, 21] and transactional memory [5, 12, 14, 16], models for large-scale distributed databases are poorly understood. In particular, we currently lack a rich theory that would guide programmers in using such models correctly and efficiently.

In this paper we make a step towards building such a theory by investigating the classical problem of transaction chopping [21] for a promising consistency model of *parallel snapshot isolation (PSI)* [22]. PSI weakens the classical *snapshot isolation (SI)* [8] in a way that allows more efficient large-scale implementations. Like in SI, a transaction in PSI reads values of objects in the database from a snapshot taken at its start. Like SI, PSI precludes *write conflicts*: when two concurrent transactions write

(a) Original transactions.

```
txn lookup(acct) {
 return acct.balance; }

txn transfer(acct1,acct2,amnt) {
 acct1.balance -= amnt;
 acct2.balance += amnt; }
```

(b) A chopping of `transfer` (`lookup` is left as is).

```
txn withdraw(acct,amnt) {
 acct.balance -= amnt; }

txn deposit(acct,amnt) {
 acct.balance += amnt; }

chain transfer(acct1,acct2,amnt)
{ withdraw(acct1,amnt);
  deposit(acct2,amnt); }
```

(c) An additional transaction making the chopping incorrect.

```
txn lookup2(acct1,acct2) {
 return acct1.balance+acct2.balance }
```

Fig. 1. Example of chopping transactions.

to the same object, one of them must abort. A PSI transaction initially commits at a single replica, after which its effects are propagated asynchronously to other replicas. Unlike SI, PSI does not enforce a global ordering on committed transactions: these are propagated between replicas in *causal* order. This ensures that, if Alice posts a message that is seen by Bob, and Bob posts a response, no user can see Bob's response without also seeing Alice's original post. However, causal propagation allows two clients to see concurrent events as occurring in different orders: if Alice and Bob concurrently post messages, then Carol may initially see Alice's message, but not Bob's, and Dave may see Bob's message, but not Alice's.

A common guideline for programmers using relational databases and transactional memory is to keep transactions as short as possible to maximise performance; long transactions should be *chopped* into smaller pieces [3, 21, 24]. This advice is also applicable to PSI databases: the longer a transaction, the higher the chances that it will abort due to a write conflict. Unfortunately, the subtle semantics of PSI makes it non-trivial to see when a transaction can be chopped without introducing undesirable behaviours. In this paper, we determine conditions that ensure this. In more detail, we assume that the code of all transactions operating on the database is known. As a toy example, consider the transactions in Figure 1(a), which allow looking up the balance of an account `acct` and transferring an amount `amnt` from an account `acct1` to an account `acct2` (with a possibility of an overdraft). To improve the efficiency of `transfer`, we may chop this transaction into a *chain* [25] of smaller transactions in Figure 1(b), which the database will execute in the order given: a `withdraw` transaction on the account `acct1` and a `deposit` transaction on the account `acct2`. This chopping is *correct* in that any client-observable behaviour of the resulting chains could be produced by the original unchopped transactions. Intuitively, even though the chopping in Figure 1(b) allows a database state where `amnt` is missing from both accounts, a client cannot notice this, because it can only query the balance of a single account. If we added the transaction `lookup2` in Figure 1(c), which returns the sum of the accounts `acct1` and `acct2`, then the chopping of `transfer` would become incorrect: by executing `lookup2` a client could observe the state with `amnt` missing from both accounts.

We propose a criterion that ensures the correctness of a given chopping of transactions executing on PSI (§5). Our criterion is weaker than the existing criterion for chopping serialisable transactions by Shasha et al. [21]: weakening consistency allows more flexibility in optimising transactions. Recent work has shown that transactions arising in web applications can be chopped in a way that drastically improves their performance when executed in serialisable databases [19, 25]. Our result enables bringing these benefits to databases providing PSI.

A challenge we have to deal with in proposing a criterion for transaction chopping is that the specification of PSI [22] is given in a low-level *operational* way, by an idealised algorithm formulated in terms of implementation-level concepts (§2). This complicates reasoning about the behaviour of an application using a PSI database and, in particular, the correctness of a transaction chopping. To deal with this problem, we propose an alternative *axiomatic* specification of PSI that defines the consistency model declaratively by a set of axioms constraining client-visible events (§3). We prove that our axiomatic specification of PSI is equivalent to the existing operational one (§4). The axiomatic specification is instrumental in formulating and proving our transaction chopping criterion.

2 Operational Specification of PSI

We first present an operational specification of PSI, which is a simplification of the one originally proposed in [22]. It is given as an idealised algorithm that is formulated in terms of implementation-level concepts, such as replicas and messages, but nevertheless abstracts from many of the features that a realistic PSI implementation would have.

We consider a database storing ***objects*** $\mathsf{Obj} = \{x, y, \ldots\}$, which for simplicity we assume to be integer-valued. Clients interact with the database by issuing `read` and `write` operations on the objects, grouped into ***transactions***. We identify transactions by elements of $\mathsf{TId} = \{t_0, t_1, \ldots\}$. The database system consists of a set of ***replicas***, identified by $\mathsf{RId} = \{r_0, r_1, \ldots\}$, each maintaining a copy of all objects. Replicas may fail by crashing.

All client operations within a given transaction are initially executed at a single replica (though operations in *different* transactions can be executed at different replicas). When a client terminates the transaction, the replica decides whether to commit or abort it. To simplify the formal development, we assume that every transaction eventually terminates. If the replica decides to commit a transaction, it sends a message to all other replicas containing the ***transaction log***, which describes the updates done by the transaction. The replicas incorporate the updates into their state upon receiving the message. A transaction log has the form $(t, \mathtt{start})\,(t, \mathtt{write}(x_1, n_1)) \ldots\ (t, \mathtt{write}(x_k, n_k))$, which gives the sequence of values $n_i \in \mathbb{N}$ written to objects $x_i \in \mathsf{Obj}$ by the transaction $t \in \mathsf{TId}$; the record $(t, \mathtt{start})$ is added for convenience of future definitions. Transaction logs are ranged over by l, and we denote their set by Log.

We assume that every replica executes transactions locally without interleaving (this is a simplification in comparison to the original PSI specification [22] that makes the

```
operation start:
    requires $\mathsf{Current}[r] = \varepsilon$
    $t :=$ (unique identifier from $\mathsf{TId}$)
    $\mathsf{Current}[r] := (t, \mathtt{start})$

operation receive($l$):
    requires $\mathsf{Current}[r] = \varepsilon$
    requires $l = (t, \mathtt{start}) \cdot \_$
    $\mathsf{Committed}[r] = \mathsf{Committed}[r] \cdot l$

operation abort:
    requires
        $\mathsf{Current}[r] = (t, \mathtt{start}) \cdot \_$
    $\mathsf{Current}[r] := \varepsilon$

operation write($x, n$):
    requires $\mathsf{Current}[r] = (t, \mathtt{start}) \cdot \_$
    $\mathsf{Current}[r] := \mathsf{Current}[r] \cdot (t, \mathtt{write}(x, n))$

operation read($x, n$):
    requires $\mathsf{Current}[r] = (t, \mathtt{start}) \cdot \_$
    requires $\mathtt{write}(x, n)$ is the last write to $x$
        in $\mathsf{Committed}[r] \cdot \mathsf{Current}[r]$ or
        there is no such write and $n = 0$

operation commit:
    requires $\mathsf{Current}[r] = (t, \mathtt{start}) \cdot \_$
    requires $\neg\exists x, r', t'. \, ((t, \mathtt{write}(x, \_)) \in \mathsf{Current}[r]) \wedge$
        $(r \neq r') \wedge ((t', \mathtt{write}(x, \_)) \in \mathsf{Committed}[r']) \wedge$
        $((t', \mathtt{start}) \notin \mathsf{Committed}[r])$
    send $\mathsf{Current}[r]$ to all other replicas
    $\mathsf{Committed}[r] := \mathsf{Committed}[r] \cdot \mathsf{Current}[r]$
    $\mathsf{Current}[r] := \varepsilon$
```

Fig. 2. Pseudocode of the idealised PSI algorithm at replica r.

algorithm cleaner). This assumption allows us to maintain the state of a replica r in the algorithm by:

– $\mathsf{Current}[r] \in \mathsf{Log} \cup \{\varepsilon\}$—the log of the (single) transaction currently executing at r or an empty sequence ε, signifying that no transaction is currently executing; and
– $\mathsf{Committed}[r] \in \mathsf{Log}^*$—the sequence of logs of transactions that committed at r.

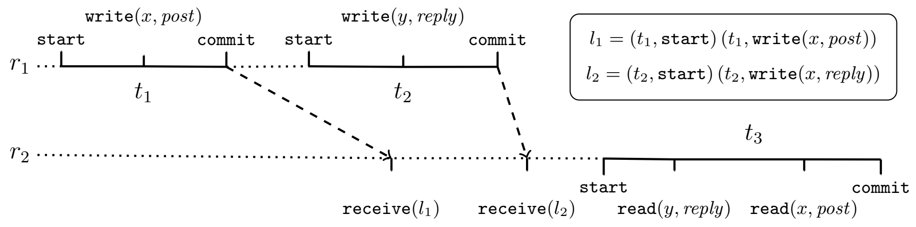
Initially $\mathsf{Current}[r] = \mathsf{Committed}[r] = \varepsilon$.

We give the pseudocode of the algorithm executing at a replica r in Figure 2. This describes the effects of ***operations*** executed at the replica, which come from the set

$$\mathsf{Op} = \{\mathtt{start}, \mathtt{receive}(l), \mathtt{write}(x, n), \mathtt{read}(x, n), \mathtt{abort}, \mathtt{commit} \mid l \in \mathsf{Log}, x \in \mathsf{Obj}, n \in \mathbb{N}\}$$

and are ranged over by o. The execution of the operations is atomic and is triggered by client requests or internal database events, such as messages arriving to the replica. The **requires** clauses give conditions under which an operation can be executed. For convenience of future definitions, operations do not return values. Instead, the value fetched by a `read` is recorded as its parameter; as we explain below, the **requires** clause for $\mathtt{read}(x, n)$ ensures that the operation may only be executed when the value it fetches is indeed n. We use $\cdot$ for sequence concatenation, $\in$ to express that a given record belongs to a given sequence, and _ for irrelevant expressions.

When a client starts a transaction at the replica r (operation `start`), the database assigns it a unique identifier t and initialises $\mathsf{Current}[r]$ to signify that t is in progress. Since we assume that the replica processes transactions serially, in the idealised algorithm the transaction can start only if r is not already executing a transaction, as expressed by the **requires** clause. The operation $\mathtt{receive}(l)$ executes when the replica receives a message l with the log of some transaction t, at which point it appends l to its log of committed transactions. A replica can receive a message only when it is not executing a transaction. When a client issues a write of n to an object x inside a transaction t, the corresponding record $(t, \mathtt{write}(x, n))$ is appended to the log of the



$\mathtt{write}(x, post)$
start
commit
$\mathtt{write}(y, reply)$
start
commit
r_1
t_1
t_2
$l_1 = (t_1, \mathtt{start})\,(t_1, \mathtt{write}(x, post))$
$l_2 = (t_2, \mathtt{start})\,(t_2, \mathtt{write}(x, reply))$
t_3
r_2
start
commit
$\mathtt{receive}(l_1)$
$\mathtt{receive}(l_2)$
$\mathtt{read}(y, reply)$
$\mathtt{read}(x, post)$

Fig. 3. An example execution of the operational PSI specification.

current transaction (operation $\mathtt{write}(x, n)$). The **requires** clause ensures that a `write` operation can only be called inside a transaction. A client can read n as the value of an object x (operation $\mathtt{read}(x, n)$) if it is the most recent value written to x at the replica according to the log of committed transactions concatenated with the log of the current one; if there is no such value, the client reads 0 (to simplify examples, in the following we sometimes assume different initial values).

If the current transaction aborts (operation `abort`), the $\mathsf{Current}[r]$ log is reset to be empty. Finally, if the current transaction commits (operation `commit`), its log is sent to all other replicas, as well as added to the log of committed transactions of the replica r. Crucially, as expressed by the second **requires** clause of `commit`, the database may commit a transaction t only if it passes the ***write-conflict detection*** check: there is no object x written by t that is also written by a concurrent transaction t', i.e., a transaction that has been committed at another replica r', but whose updates have not yet been received by r. If this check fails, the only option left for the database is to abort t using the operation `abort`.

In the algorithm we make certain assumptions about message delivery between replicas. First, every message is delivered to every replica at most once. Second, message delivery is ***causal***: if a replica sends a message l_2 after it sends or receives a message l_1, then every other replica will receive l_2 only after it receives or sends l_1; in this case we say that the transaction generating l_2 causally depends on the one generating l_1. This is illustrated by the execution of the algorithm depicted in Figure 3: due to causal delivery, the transaction t_3 that reads *reply* from y is also guaranteed to read *post* from x.

The operational specification of PSI is given by all sets of client-database interactions that can arise when executing the implementations of the operations in Figure 2 at each replica in the system. Due to the asynchronous propagation of updates between replicas, the specification of PSI allows non-serialisable behaviours, called ***anomalies***. We introduce structures to describe client-database interactions allowed by PSI and discuss its anomalies while presenting our declarative PSI specification, which is the subject of the next section.

3 Axiomatic Specification of PSI

Reasoning about PSI database behaviour using the operational specification may get unwieldy. It requires us to keep track of low-level information about the system state, such as the logs at all replicas and the set of messages in transit. We then need to reason about how the system state is affected by a large number of possible interleavings of operations at different replicas. We now present a specification of PSI that is more declarative than the operational one and, in particular, does not refer to implementation-level details, such as message exchanges between replicas. It thus makes it easier to establish results about PSI, such as criteria for transaction chopping.

Our PSI specification is given by a set of *histories*, which describe all client-database interactions that this consistency model allows. To simplify presentation, our specification does not constrain the behaviour of aborted or ongoing transactions, so that histories only record operations inside committed transactions. Our specification also assumes that the database interface allows a client to group a finite number of transactions into a ***chain*** [25], which establishes an ordering on the transactions, similarly to a *session* [23]. Chains are needed for transaction chopping (§1) and can be implemented, e.g., by executing all transactions from a chain at the same replica.

To define histories and similar structures, we need to introduce some set-theoretic concepts. We assume a countably infinite set of ***events*** $\mathsf{Event} = \{e, f, g, \ldots\}$. A relation $R \subseteq E \times E$ on a set E is a ***strict partial order*** if it is transitive and irreflexive; it is an ***equivalence relation*** if it is reflexive, transitive and symmetric. For an equivalence relation $R \subseteq E \times E$ and $e \in E$, we let $[e]_R = \{f \mid (f, e) \in R\}$ be the ***equivalence class*** of e. A ***total order*** is a strict partial order such that for every two distinct elements e and f, the order relates e to f or f to e. We write $(e, f) \in R$ and $e \xrightarrow{R} f$ interchangeably.

Definition 1. *A **history** is a tuple $\mathcal{H} = (E, \mathsf{op}, \mathsf{co}, \sim)$, where:*

- $E \subseteq \mathsf{Event}$ *is a finite set of events, denoting reads and writes performed inside committed transactions.*
- $\mathsf{op} : E \to \{\mathtt{write}(x, n), \mathtt{read}(x, n) \mid x \in \mathsf{obj}, n \in \mathbb{N}\}$ *defines the operation each event denotes.*
- $\mathsf{co} \subseteq E \times E$ *is the **chain order**, arranging events in the same chain into the order in which a client submitted them to the database. We require that* co *be a union of total orders defined on disjoint subsets of* E*, which correspond to events in different chains.*
- $\sim \subseteq E \times E$ *is an equivalence relation grouping events in the same transaction. Since every transaction is performed by a single chain, we require that* co *totally order events within each transaction, i.e., those from* $[e]_\sim$ *for each* $e \in E$*. We also require that a transaction be contiguous in* co*:*

$$\forall e, f, g.\, e \xrightarrow{\mathsf{co}} f \xrightarrow{\mathsf{co}} g \wedge e \sim g \implies e \sim f \sim g.$$

Let Hist be the set of all histories. We denote components of a history $\mathcal{H}$ as in $E_{\mathcal{H}}$, and use the same notation for similar structures introduced in this paper. Our specification of PSI is given as a particular set of histories allowed by this consistency model. To define this set, we enrich histories with a *happens-before* relation, capturing causal relationships between events. In terms of the operational PSI specification, an event e happens before an event f if the information about e has been delivered to the replica performing f, and hence, can affect f's behaviour. The resulting notion of an *abstract execution* is similar to those used to specify weak shared-memory models [4].

$\mathsf{op}(e) = \mathtt{read}(x,n) \implies \big(\exists f.\, \mathsf{op}(f) = \mathtt{write}(x,n) \land f \xrightarrow{\mathsf{hb}} e \land \neg\exists g.\, f \xrightarrow{\mathsf{hb}} g \xrightarrow{\mathsf{hb}} e \land \mathsf{op}(g) = \mathtt{write}(x,\_)\big) \lor \big(n = 0 \land \neg\exists f.\, f \xrightarrow{\mathsf{hb}} e \land \mathsf{op}(f) = \mathtt{write}(x,\_)\big)$ (Reads)	
$\mathsf{co} \subseteq \mathsf{hb}$ (Chains)	$\{(e',f') \mid e \xrightarrow{\mathsf{hb}} f \land e \not\sim f \land e' \sim e \land f \sim f'\} \subseteq \mathsf{hb}$ (Atomic)
$(e \neq f \land \{\mathsf{op}(e), \mathsf{op}(f)\} \subseteq \{\mathtt{write}(x,n) \mid n \in \mathbb{N}\}) \implies (e \xrightarrow{\mathsf{hb}} f \lor f \xrightarrow{\mathsf{hb}} e)$ (Wconflict)	

Fig. 4. Consistency axioms of PSI, stated for an execution $\mathcal{A} = ((E, \mathsf{op}, \mathsf{co}, \sim), \mathsf{hb})$. All free variables are universally quantified.

Definition 2. *An **abstract execution** is a pair $\mathcal{A} = (\mathcal{H}, \mathsf{hb})$ of a history $\mathcal{H}$ and the **happens-before** relation $\mathsf{hb} \subseteq E \times E$, which is a strict partial order.*

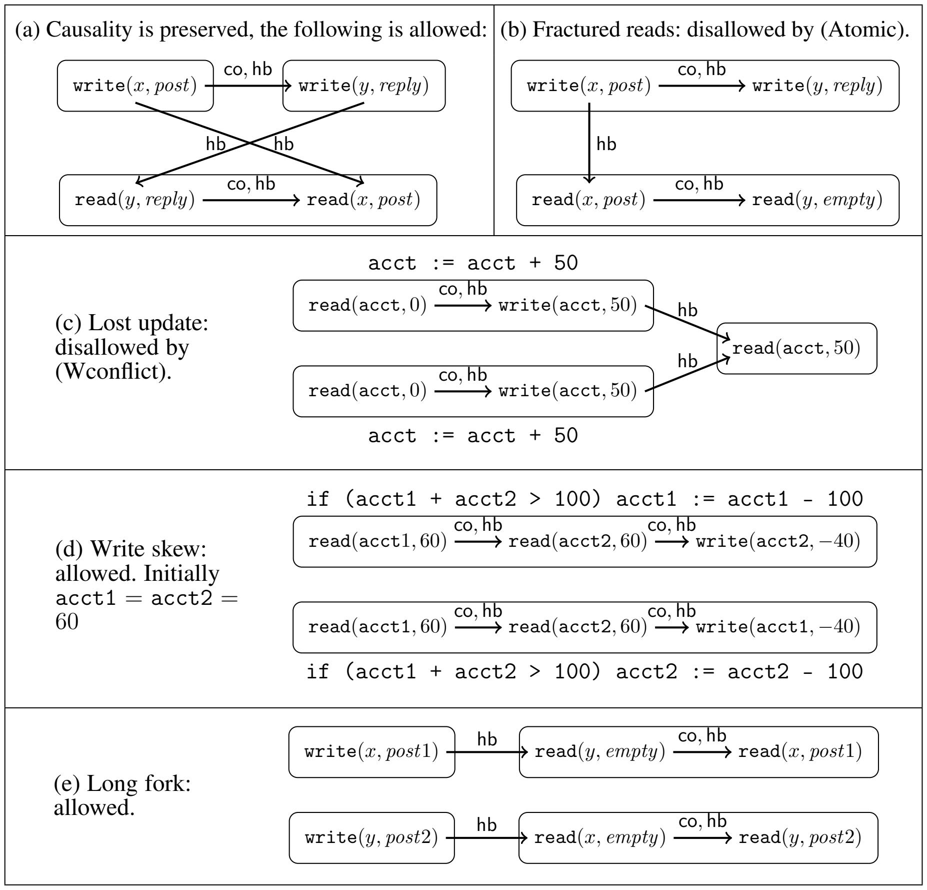
For example, Figure 5(a) shows an abstract execution, which corresponds to the execution of the operational specification in Figure 3 (as we formalise in §4). Our PSI specification is defined by *consistency axioms* (Figure 4), which constrain happens-before and other execution components and thereby describe the guarantees that a PSI database provides about transaction processing. We thus call this specification ***axiomatic***.

Definition 3. *An abstract execution $\mathcal{A}$ is **valid** if it satisfies the **consistency axioms** in Figure 4. We denote the set of all valid executions by* AbsPSI *and let the set of PSI histories be* $\mathsf{HistPSI} = \{\mathcal{H} \in \mathsf{Hist} \mid \exists \mathsf{hb}.\, (\mathcal{H}, \mathsf{hb}) \in \mathsf{AbsPSI}\}$.

The axiom (Reads) constrains the values fetched by a read using the happens-before relation: a read e from an object x has to return the value written by a hb-preceding write f on x that is most recent according to hb, i.e., not shadowed by another write g to x. If there is no hb-preceding write to x, then the read fetches the default value 0 (we sometimes use other values in examples). The axiom (Chains) establishes a causal dependency between events in the same chain (thus subsuming *session guarantees* [23]), and the transitivity of happens-before required in Definition 2 ensures that the database respects causality. For example, in the abstract execution in Figure 5(a), the chain order between the two writes induces an hb edge according to (Chains). Then, since hb is transitive, we must have an hb edge between the two operations on x and, hence, by (Reads), the read from x has to fetch *post*. There is no valid execution with a history where the read from y fetches *reply*, but the read from x fetches the default value. The operational specification ensures this because of causal message delivery.

The axiom (Atomic) ensures the ***atomic visibility*** of transactions: all writes by a transaction become visible to other transactions together. It requires that, if an event e happens before an event f in a different transaction, then all events e' in the transaction of e happen before all the events f' in the transaction of f. For example, (Atomic) disallows the execution in Figure 5(b), which is a variant of Figure 5(a) where the two writes are done in a single transaction and the order of the reads is reversed.

The axiom (Wconflict) states that the happens-before relation is total over write operations on a given object. Hence, the same object cannot be written by concurrent transactions, whose events are not related by happens-before. This disallows the ***lost update*** anomaly, illustrated by the execution in Figure 5(c). This execution could arise



```
acct := acct + 50
```

```
if (acct1 + acct2 > 100) acct1 := acct1 - 100
if (acct1 + acct2 > 100) acct2 := acct2 - 100
```

Fig. 5. Abstract executions illustrating PSI guarantees and anomalies. The boxes group events into transactions. We omit the transitive consequences of the co and hb edges shown.

from the code, also shown in the figure, that uses transactions to make deposits into an account; in this case, one deposit is lost. The execution violates (Wconflict): one of the transactions would have to hb-precede the other and, hence, read 50 instead of 0 from x. In the operational specification this anomaly is disallowed by the write-conflict detection, which would allow only one of the two concurrent transactions to commit.

Despite PSI disallowing many anomalies, it is weaker than serialisability. In particular, PSI allows the ***write skew*** anomaly, also allowed by the classical snapshot isolation [8]. We illustrate how our consistency axioms capture this by the valid execution in Figure 5(d), which could arise from the code also shown in the figure. Here each transaction checks that the combined balance of two accounts exceeds 100 and, if so, withdraws 100 from one of them. Both transactions pass the checks and make the withdrawals from different accounts, resulting in the combined balance going negative. The operational specification allows this anomaly because the two transactions can be executed at different replicas and allowed to commit by the write-conflict detection check.

PSI also allows so-called ***long fork*** anomaly in Figure 5(e) [22], which we in fact already mentioned in §1. We have two concurrent transactions writing to x and y, respectively. A third transaction sees the write to x, but not y, and a fourth one sees the write to y, but not x. Thus, from the perspective of the latter two transactions, the two writes happen in different orders. It is easy to check that this outcome is not serialisable; in fact, it is also disallowed by the classical snapshot isolation. In the operational specification this anomaly can happen when each transaction executes at a separate replica, and the messages about the writes to x and y are delivered to the replicas executing the reading transactions in different orders.

4 Equivalence of the Specifications

We now show that the operational (§2) and axiomatic specifications (§3) are equivalent, i.e., the sets of histories they allow coincide. We start by introducing a notion of *concrete executions* of the operational PSI specification and using it to define the set of histories the specification allows. Concrete executions are similar to abstract ones of Definition 2, but describe *all* operations occurring at replicas as per Figure 2, including both client-visible and database-internal ones. We use the set-theoretic notions introduced before Definition 1.

Definition 4. *A* ***concrete execution*** *is a tuple* $\mathcal{C} = (E, \mathsf{op}, \mathsf{repl}, \mathsf{trans}, \prec)$*, where:*

- $E \subseteq \mathsf{Event}$ *is a finite set of events, denoting executions of operations in Figure 2.*
- $\mathsf{op} : E \to \mathsf{Op}$ *defines which of the operations in Figure 2 a given event denotes.*
- $\mathsf{repl} : E \to \mathsf{RId}$ *defines the replica on which the event occurs.*
- $\mathsf{trans} : E \to \mathsf{TId}$ *defines the transaction to which the event pertains.*
- $\prec \subseteq E \times E$ *is a total order, called* **execution order***, in which events take place in the system.*

The set $\mathsf{ConcPSI}$ of concrete executions that can be produced by the algorithm in Figure 2 is defined as expected. Due to space constraints, we defer its formal definition to [11, §C]. Informally, the definition considers the execution of any sequence of operations in Figure 2 at arbitrary replicas, subject to the **requires** clauses and the constraints on message delivery mentioned in §2; the values of repl and trans are determined by the variables r and t in the code of operations in Figure 2. For example, Figure 3 can be viewed as a graphical depiction of a concrete execution from $\mathsf{ConcPSI}$, with the execution order given by the horizontal placement of events. For a $\mathcal{C} \in \mathsf{ConcPSI}$ and $e \in E_{\mathcal{C}}$, we write $e \triangleright_{\mathcal{C}} t : o \mathbin{@} r$ if $\mathsf{trans}_{\mathcal{C}}(e) = t$, $\mathsf{op}_{\mathcal{C}}(e) = o$ and $\mathsf{repl}_{\mathcal{C}}(e) = r$.

Definition 5. *The history of a concrete execution* $\mathcal{C}$ *is*

$$\begin{aligned}
\mathsf{history}(\mathcal{C}) &= (E_{\mathcal{H}}, \mathsf{op}_{\mathcal{H}}, \mathsf{co}_{\mathcal{H}}, \sim_{\mathcal{H}}), \textit{ where}\\
E_{\mathcal{H}} &= \{e \in E_{\mathcal{C}} \mid \exists f \in E_{\mathcal{C}}, t \in \mathsf{TId}.\, (f \triangleright_{\mathcal{C}} t : \mathtt{commit} \mathbin{@} \_) \land {}\\
&\qquad ((e \triangleright_{\mathcal{C}} t : \mathtt{write}(\_,\_) \mathbin{@} \_) \lor (e \triangleright_{\mathcal{C}} t : \mathtt{read}(\_,\_) \mathbin{@} \_))\};\\
\mathsf{op}_{\mathcal{H}} &= (\textit{the restriction of } \mathsf{op}_{\mathcal{C}} \textit{ to } E_{\mathcal{H}});\\
\mathsf{co}_{\mathcal{H}} &= \{(e, f) \in E_{\mathcal{H}} \times E_{\mathcal{H}} \mid \mathsf{repl}_{\mathcal{C}}(e) = \mathsf{repl}_{\mathcal{C}}(f) \land e \prec_{\mathcal{C}} f\}.\\
\sim_{\mathcal{H}} &= \{(e, f) \in E_{\mathcal{H}} \times E_{\mathcal{H}} \mid \mathsf{trans}_{\mathcal{C}}(e) = \mathsf{trans}_{\mathcal{C}}(f)\};
\end{aligned}$$

For example, the concrete execution in Figure 3 has the history shown in Figure 5(a). The history $\mathsf{history}(\mathcal{C})$ contains only the events describing reads and writes by the committed transactions in $\mathcal{C}$. To establish a correspondence between the operational and axiomatic specifications, we assume that chains are implemented by executing every one of them at a dedicated replica. Thus, we define the chain order $\mathsf{co}_{\mathcal{H}}$ as the order of events on each replica according to $\prec_{\mathcal{C}}$. This is, of course, an idealisation acceptable only in a specification. In a realistic implementation, multiple chains would be multiplexed over a single replica, or different transactions in a chain would be allowed to access different replicas [23]. We define the set of histories allowed by the operational PSI specification as $\mathsf{history}(\mathsf{ConcPSI})$, where we use the expected lifting of $\mathsf{history}$ to sets of executions. The following theorem (proved in [11, §D]) shows that this set coincides with the one defined by the axiomatic specification (Definition 3).

Theorem 1. $\mathsf{history}(\mathsf{ConcPSI}) = \mathsf{HistPSI}$.

5 Chopping PSI Transactions

In this section, we exploit the axiomatic specification of §3 to establish a criterion for checking the correctness of a ***chopping*** [21] of transactions executing on PSI. Namely, we assume that we are given a set of ***chain programs*** $\mathcal{P} = \{P_1, P_2, \ldots\}$, each defining the code of chains resulting from chopping the code of a single transaction. We leave the precise syntax of the programs unspecified, but assume that each P_i consists of k_i ***program pieces***, defining the code of the transactions in the chain. For example, for given `acct1`, `acct2` and `amnt`, Figure 1(b) defines a chain program resulting from chopping `transfer` in Figure 1(a). For a given `acct`, we can also create a chain program consisting of a single piece `lookup(acct)` in Figure 1(a). Let $\mathcal{P}^1$ consist of the programs for `lookup(acct1)`, `lookup(acct2)` and `transfer(acct1,acct2,amnt)`, and $\mathcal{P}^2$ of those for `transfer(acct1,acct2,amnt)` and `lookup2(acct1,acct2)`.

Following Shasha et al. [21], we make certain assumptions about the way clients execute chain programs. We assume that, if the transaction initiated by a program piece aborts, it will be resubmitted repeatedly until it commits, and, if a piece is aborted due to system failure, it will be restarted. We also assume that the client does not abort transactions explicitly.

In general, executing the chains $\mathcal{P}$ may produce more client-observable behaviours than if we executed every chain as a single PSI transaction. We propose a condition for checking that no new behaviours can be produced. To this end, we check that every valid abstract execution consisting of *fine-grained* transactions produced by the chains $\mathcal{P}$ can be *spliced* into another valid execution that has the same operations as the original one, but where all operations from each chain are executed inside a single *coarse-grained* transaction.

Definition 6. *Consider a valid abstract execution* $\mathcal{A} = ((E, \mathsf{op}, \mathsf{co}, \sim), \mathsf{hb}) \in \mathsf{AbsPSI}$ *and let* $\approx_{\mathcal{A}} = \mathsf{co} \cup \mathsf{co}^{-1} \cup \{(e,e) \mid e \in E\}$*. The execution* $\mathcal{A}$ *is **spliceable** if there exists* hb' *such that* $((E, \mathsf{op}, \mathsf{co}, \approx_{\mathcal{A}}), \mathsf{hb}') \in \mathsf{AbsPSI}$.

The definition groups fine-grained transactions in $\mathcal{A}$, identified by $\sim_{\mathcal{A}}$, into coarse-grained transactions, identified by $\approx_{\mathcal{A}}$, which consist of events in the same chain.

We now establish the core technical result of this section—a criterion for checking that an execution $\mathcal{A}$ is spliceable. From this *dynamic* criterion on executions we then obtain a *static* criterion for the correctness of chopping transaction code, by checking that all executions produced by the chain programs $\mathcal{P}$ are spliceable. We first need to define some auxiliary relations, derived from the happens-before relation in an abstract execution [2, 4].

Definition 7. *Given $\mathcal{A} \in \mathsf{AbsPSI}$, we define the **reads-from** $\mathsf{rf}_{\mathcal{A}}$, **version-order** $\mathsf{vo}_{\mathcal{A}}$ and **anti-dependency** $\mathsf{ad}_{\mathcal{A}}$ relations on $E_{\mathcal{A}}$ as follows:*

$$\begin{aligned}
e \xrightarrow{\mathsf{rf}_{\mathcal{A}}} f \iff{} & \exists x, n.\, e \xrightarrow{\mathsf{hb}_{\mathcal{A}}} f \wedge \mathsf{op}_{\mathcal{A}}(e) = \mathtt{write}(x, n) \wedge \mathsf{op}_{\mathcal{A}}(f) = \mathtt{read}(x, n) \wedge \\
& \neg\exists g.\, e \xrightarrow{\mathsf{hb}_{\mathcal{A}}} g \xrightarrow{\mathsf{hb}_{\mathcal{A}}} f \wedge \mathsf{op}_{\mathcal{A}}(g) = \mathtt{write}(x, \_);\\
e \xrightarrow{\mathsf{vo}_{\mathcal{A}}} f \iff{} & \exists x.\, e \xrightarrow{\mathsf{hb}_{\mathcal{A}}} f \wedge \mathsf{op}_{\mathcal{A}}(e) = \mathtt{write}(x, \_) \wedge \mathsf{op}_{\mathcal{A}}(f) = \mathtt{write}(x, \_);\\
e \xrightarrow{\mathsf{ad}_{\mathcal{A}}} f \iff{} & \exists x.\, \mathsf{op}_{\mathcal{A}}(e) = \mathtt{read}(x, \_) \wedge \mathsf{op}_{\mathcal{A}}(f) = \mathtt{write}(x, \_) \wedge \\
& ((\exists g.\, g \xrightarrow{\mathsf{rf}_{\mathcal{A}}} e \wedge g \xrightarrow{\mathsf{vo}_{\mathcal{A}}} f) \vee (\neg\exists g.\, g \xrightarrow{\mathsf{rf}_{\mathcal{A}}} e)).
\end{aligned}$$

The reads-from relation determines the write e that a read f fetches its value from (uniquely, due to the axiom (Wconflict)). The version order totally orders all writes to a given object and corresponds to the order in which replicas find out about them in the operational specification. The anti-dependency relation [2] is more complicated. We have $e \xrightarrow{\mathsf{ad}_{\mathcal{A}}} f$ if the read e fetches a value that is overwritten by the write f according to $\mathsf{vo}_{\mathcal{A}}$ (the initial value of an object is overwritten by any write to this object).

Our criterion for checking that $\mathcal{A}$ is spliceable requires the absence of certain cycles in a graph with nodes given by the fine-grained transactions in $\mathcal{A}$ and edges generated using the above relations. The transactions are defined as equivalence classes $[e]_{\sim}$ of events $e \in E_{\mathcal{A}}$ (§3).

Definition 8. *Given $\mathcal{A} \in \mathsf{AbsPSI}$, its **dynamic chopping graph** $\mathsf{DCG}(\mathcal{A})$ is a directed graph whose set of nodes is $\{[e]_{\sim_{\mathcal{A}}} \mid e \in E_{\mathcal{A}}\}$, and we have an edge $([e]_{\sim_{\mathcal{A}}}, [f]_{\sim_{\mathcal{A}}})$ if and only if $[e]_{\sim_{\mathcal{A}}} \neq [f]_{\sim_{\mathcal{A}}}$ and one of the following holds: $e \xrightarrow{\mathsf{co}_{\mathcal{A}}} f$ (a **successor** edge); $f \xrightarrow{\mathsf{co}_{\mathcal{A}}} e$ (a **predecessor** edge); $e \xrightarrow{\mathsf{ad}_{\mathcal{A}} \setminus \approx_{\mathcal{A}}} f$ (an **anti-dependency** edge); or $e \xrightarrow{(\mathsf{rf}_{\mathcal{A}} \cup \mathsf{vo}_{\mathcal{A}}) \setminus \approx_{\mathcal{A}}} f$ (a **dependency** edge).*

*A **conflict** edge is one that is either a dependency or an anti-dependency. A directed cycle in the dynamic chopping graph is **critical** if it does not contain two occurrences of the same vertex, contains at most one anti-dependency edge, and contains a fragment of three consecutive edges of the form "conflict, predecessor, conflict".*

Theorem 2 (Dynamic Chopping Criterion). *An execution $\mathcal{A} \in \mathsf{AbsPSI}$ is spliceable if its dynamic chopping graph $\mathsf{DCG}(\mathcal{A})$ does not have critical cycles.*

We give a (non-trivial) proof of the theorem in [11, §E]. For example, the execution in Figure 6 satisfies the criterion in Theorem 2 and, indeed, we obtain a valid execution by grouping `withdraw` and `deposit` into a single transaction and adding the dotted happens-before edges.

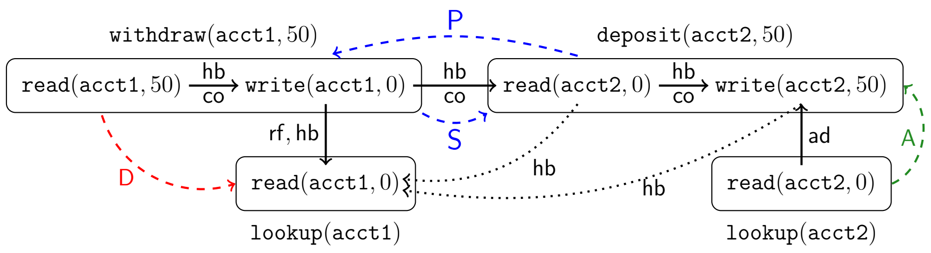


Fig. 6. An execution produced by the programs $\mathcal{P}^1$ and its derived relations. Initially $\mathtt{acct1} = 50$ and $\mathtt{acct2} = 0$. We omit the transitive consequences of the hb edges shown. The dashed edges show the dynamic chopping graph, with S, P, A, D denoting edge types. The dotted edges show additional happens-before edges that define a splicing of the execution (Definition 6).

We now use Theorem 2 to derive a static criterion for checking the correctness of code chopping given by $\mathcal{P}$. As is standard [13, 21], we formulate the criterion in terms of the sets of objects read or written by program pieces. Namely, for each chain program $P_i \in \mathcal{P}$ we assume a sequence

$$(R^i_1, W^i_1)\,(R^i_2, W^i_2)\,\ldots\,(R^i_{k_i}, W^i_{k_i}), \qquad (1)$$

of ***read and write sets*** $R^i_j, W^i_j \subseteq \mathsf{Obj}$, i.e., the sets of all objects that can be, respectively, read and written by the j-th piece of P_i. For example, the `transfer(acct1,acct2,amnt)` chain in Figure 1(b) is associated with the sequence $(\{\mathtt{acct1}\}, \{\mathtt{acct1}\})\,(\{\mathtt{acct2}\}, \{\mathtt{acct2}\})$.

We consider a chopping defined by the programs $\mathcal{P}$ correct if all executions that they produce are spliceable. To formalise this, we first define when an execution can be produced by programs with read and write sets given by (1). Due to space constraints, we give the definition only informally.

Definition 9. *An abstract execution $\mathcal{A}$* ***conforms*** *to a set of programs $\mathcal{P}$, if there is a one-to-one correspondence between every chain of transactions in $\mathcal{A}$ and a chain program $P_i \in \mathcal{P}$ whose read and write sets (1) cover the sets of objects read or written by the corresponding transactions in the chain.*

For example, the execution in Figures 6 conforms to the programs $\mathcal{P}^1$. Due to the assumptions about the way clients execute $\mathcal{P}$ that we made at the beginning of this section, the definition requires that every chain in an execution $\mathcal{A}$ conforming to $\mathcal{P}$ executes completely, and that all transactions in it commit. Also, for simplicity (and following [21]), we assume that every chain in $\mathcal{A}$ results from a distinct program in $\mathcal{P}$.

Definition 10. *Chain programs $\mathcal{P}$ are* ***chopped correctly*** *if every valid execution conforming to $\mathcal{P}$ is spliceable.*

We check the correctness of $\mathcal{P}$ by defining an analogue of the dynamic chopping graph from Definition 8 whose nodes are pieces of $\mathcal{P}$, rather than transactions in a given execution. Each piece is identified by a pair (i, j) of the number of a chain P_i and the piece's position in the chain.

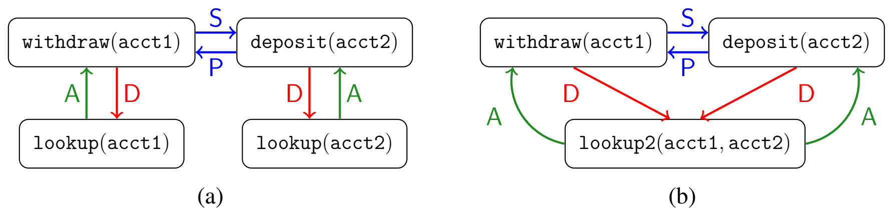


Fig. 7. Static chopping graphs for the programs (a) $\mathcal{P}^1$ and (b) $\mathcal{P}^2$.

Definition 11. *Given chain programs $\mathcal{P} = \{P_1, P_2, \dots\}$ with read and write sets (1), the **static chopping graph** $\mathsf{SCG}(\mathcal{P})$ is a directed graph whose set of nodes is $\{(i, j) \mid i = 1..|\mathcal{P}|,\ j = 1..k_i\}$, and we have an edge $((i_1, j_1), (i_2, j_2))$ if and only if one of the following holds: $i_1 = i_2$ and $j_1 < j_2$ (a **successor** edge); $i_1 = i_2$ and $j_1 > j_2$ (a **predecessor** edge); $i_1 \neq i_2$, and $R^{i_1}_{j_1} \cap W^{i_2}_{j_2} \neq \emptyset$ (an **anti-dependency** edge); or $i_1 \neq i_2$, and $W^{i_1}_{j_1} \cap (R^{i_2}_{j_2} \cup W^{i_2}_{j_2}) \neq \emptyset$ (a **dependency** edge).*

For example, Figures 7(a) and 7(b) show the static chopping graph for the programs $\mathcal{P}^1$ and $\mathcal{P}^2$ respectively. There is a straightforward correspondence between $\mathsf{SCG}(\mathcal{P})$ and $\mathsf{DCG}(\mathcal{A})$ for an execution $\mathcal{A}$ conforming to $\mathcal{P}$: we have an (anti-)dependency edge between two pieces in $\mathsf{SCG}(\mathcal{P})$ if there *may* exist a corresponding edge in $\mathsf{DCG}(\mathcal{A})$ between two transactions resulting from executing the pieces, as determined by the read and write sets. Using this correspondence, from Theorem 2 we easily get a criterion for checking chopping correctness statically.

Corollary 1 (Static Chopping Criterion). *$\mathcal{P}$ is chopped correctly if $\mathsf{SCG}(\mathcal{P})$ does not contain any critical cycles.*

The graph in Figure 7(a) satisfies the condition of the corollary, whereas the one in Figure 7(b) does not. Hence, the corresponding chopping of `transfer` is correct, but becomes incorrect if we add `lookup2` (we provide an example execution illustrating the latter case in [11, §A]).

The criterion in Corollary 1 is more permissive than the one for chopping serialisable transactions previously proposed by Shasha et al. [21]. The latter does not distinguish between dependency and anti-dependency edges (representing them by a single type of a conflict edge) and between predecessor and successor edges (representing them by *sibling* edges). The criterion then requires the absence of any cycles containing both a conflict and a sibling edge. We illustrate the difference in Figure 8. The static chopping graph for the programs shown in the figure does not have critical cycles, but has a cycle with both a conflict and a sibling edge, and thus does not satisfy Shasha's criterion. We also show an execution produced by the programs: splicing the chains in it into single transactions (denoted by the dashed boxes) yields the execution in Figure 5(e) with a long fork anomaly. We provide a similar example for write skew (Figure 5(d)) in [11, §A]. Thus, the chopping criterion for PSI can be more permissive than the one for serialisability because of the anomalies allowed by the former consistency model.

```
txn write1 { x := post1; }          txn write2 { y := post2; }
chain read1 { txn { a := y }; txn { b := x }; return (a, b); }
chain read2 { txn { a := x }; txn { b := y }; return (a, b); }
```

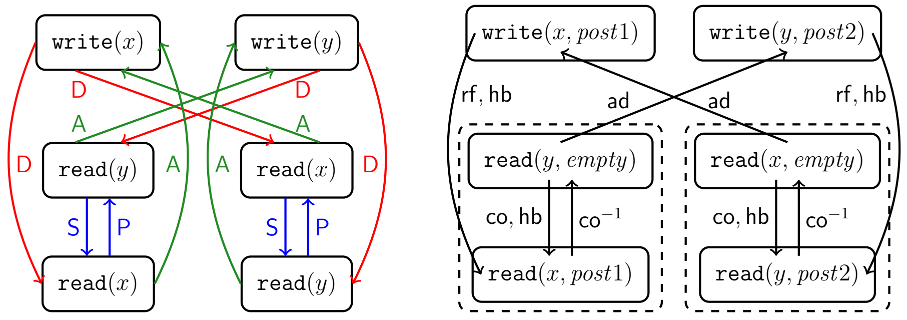


Fig. 8. An illustration of the difference between the chopping criteria for PSI and serialisability: programs, their static chopping graph and an example execution. The variables a and b are local.

Finally, we note that Theorem 2 and Corollary 1 do not make any assumptions about the structure of transactions, such as their commutativity properties, which may result in an excessive number of conflict edges in chopping graphs. These results can be strengthened to eliminate conflict edges between transactions whose effects commute, as done in [21, 25].

6 Related Work

Our criterion for the correctness of chopping PSI transactions was inspired by the criterion of Shasha et al. [21] for serialisable transactions. However, establishing a criterion for PSI is much more difficult than for serialisability. Due to the weakly consistent nature of PSI, reasoning about chopping correctness cannot be reduced to reasoning about a total serialisation order of events and requires considering intricate relationships between them, as Theorem 2 illustrates.

Our declarative specification of PSI uses a representation of executions more complex than the one in notions of strong consistency, such as serialisability [9] or linearizability [17]. This is motivated by the need to capture PSI anomalies. In proposing our specification, we built on the axiomatic approach to specifying consistency models, previously applied to eventual consistency [10] and weak shared-memory models [4]. In comparison to prior work, we handle a more sophisticated consistency model, including transactions with write-conflict detection. Our specification is also similar in spirit to the specifications of weak consistency models of relational databases of Adya's [2], which are based on the relations in Definition 7. While PSI could be specified in Adya's framework, we found that the specification based on the happens-before relation (Definition 2) results in simpler axioms and greatly eases proving the correspondence to the operational specification (Theorem 1) and the chopping criterion (Theorem 2).

Acknowledgements. We thank Hagit Attiya and Giovanni Bernardi for helpful discussions. This work was supported by EPSRC and an EU FET project ADVENT.

References

1. D. Abadi. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *IEEE Computer*, 45(2), 2012.
2. A. Adya. Weak consistency: A generalized theory and optimistic implementations for distributed transactions. PhD thesis, MIT, 1999.
3. Y. Afek, H. Avni, and N. Shavit. Towards consistency oblivious programming. In *OPODIS*, 2011.
4. J. Alglave. A formal hierarchy of weak memory models. *Formal Methods in System Design*, 41(2), 2012.
5. H. Attiya, A. Gotsman, S. Hans, and N. Rinetzky. A programming language perspective on transactional memory consistency. In *PODC*, 2013.
6. P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Highly Available Transactions: virtues and limitations. In *VLDB*, 2014.
7. P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Scalable atomic visibility with RAMP transactions. In *SIGMOD*, 2014.
8. H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD*, 1995.
9. P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
10. S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski. Replicated data types: specification, verification, optimality. In *POPL*, 2014.
11. A. Cerone, A. Gotsman, and H. Yang. Transaction chopping for parallel snapshot isolation (extended version). Available from http://software.imdea.org/∼gotsman/.
12. S. Doherty, L. Groves, V. Luchangco, and M. Moir. Towards formally specifying and verifying transactional memory. *Formal Aspects of Computing*, 25(5), 2013.
13. A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2), 2005.
14. P. Felber, V. Gramoli, and R. Guerraoui. Elastic transactions. In *DISC*, 2009.
15. S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2), 2002.
16. R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPoPP*, 2008.
17. M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3), 1990.
18. A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2), 2010.
19. S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li. Extracting more concurrency from distributed transactions. In *OSDI*, 2014.
20. M. Saeida Ardekani, P. Sutra, and M. Shapiro. Non-monotonic snapshot isolation: Scalable and strong consistency for geo-replicated transactional systems. In *SRDS*, 2013.
21. D. Shasha, F. Llirbat, E. Simon, and P. Valduriez. Transaction chopping: Algorithms and performance studies. *ACM Trans. Database Syst.*, 20(3), 1995.
22. Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *SOSP*, 2011.
23. D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. W. Welch. Session guarantees for weakly consistent replicated data. In *PDIS*, 1994.
24. L. Xiang and M. L. Scott. Software partitioning of hardware transactions. In *PPoPP*, 2015.
25. Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. Aguilera, and J. Li. Transaction chains: Achieving serializability with low latency in geo-distributed storage systems. In *SOSP*, 2013.