Daniel Pfeifer

OCTP – An Optimistic Transactional Cache Protocol with Low Abort Rates

July 29, 2005

Abstract Since the early nineties transactional cache protocols have been intensively studied in the context of client-server database systems. Research has developed a variety of protocols and compared different aspects of their quality.

In this paper we present a new transactional cache protocol, called “Optimistic Caching Timestamp Protocol” (OCTP). OCTP is a pure optimistic protocol and represents a strong improvement over OCC – a classical optimistic transactional cache protocol. OCC is known to have very low message overhead but suffers from high transaction abort rates. In contrast, OCTP’s message overhead is the same as that of OCC but its abort rates are considerably lower. OCTP does not require locks to coordinate concurrent transactions but uses a backward validating timestamp-based approach instead. As opposed to all other known transactional cache protocols, it can allow transactions to commit which have read stale cached data elements while still asserting serializability. Its computational complexity is moderate and in particular, it does apply a potentially costly serializability graph test. We also present an extension of OCTP called “Semi-Optimistic Caching Timestamp Protocol” (SOCTP), which reduces abort rates further. In certain cases, SOCTP efficiently uses locks to prevent transaction aborts.

This paper explains the concepts behind OCTP and proves its correctness using a multiversion transaction formalism. It sketches an efficient implementation of OCTP and compares OCTP as well as SOCTP against two leading conventional protocols, namely OCC and CBL. Simulation experiments show that both SOCTP and OCTP outperform OCC and CBL given that the network represents the bottleneck of a related database system.

Keywords Caching · Client-Server · Protocol · Transaction Management

1 Introduction

In the early nineties database researchers have started to study client-server database systems. At this type of systems, clients may access a central database over the network in order to perform transactions. For read and write access, a client downloads a fixed unit of data from the server database and stores it in a local cache. Depending on the type of the system the data units may be pages or objects or both.

Fig. 1 illustrates the basic architecture of a client-server database system. A client may access several pages within the same transaction and store page content in a local cache even across transactions. At transaction commit, all changes performed by the transaction must eventually be propagated to the server and must be written to the server’s stable memory. For simplicity we make an abstraction by assuming that every client executes at most one transaction at a time.

It is widely accepted that client-server database systems should meet two important requirements:

– Client transactions should be serializable at the database server. (Serializability is a well-known consistency requirement for database systems [5].)
– Execution of client requests should be as efficient as possible. This includes high transaction throughput, low transaction latency and a low probability of transaction aborts.

Obviously, the protocol for processing client operations has a crucial impact on these requirements. Any such protocol which caches downloaded pages at the client side and which meets the first two of the above requirements is called a transactional cache protocol. In the following we refer to it more shortly as a protocol.

Another often less crucial requirement for protocols is the so-called external consistency [1]. It ensures that a valid serialization order of committing transactions is (about the

1 If a client is expected to perform several concurrent transaction locally it can coordinate its local transactions by applying a conventional concurrency protocol (e.g. two phase locking). Since this strategy can be well separated from the essential structure of a transactional cache protocol it is not in the focus of this paper.
same as the realtime commit order at the server. Often it is acceptable, if the serialization order and the realtime commit order deviate from each other within certain well-defined bounds.

One of the goals of various research contributions in the field of client-server database systems was to invent highly efficient protocols [1,6,8,11,12,16–18] and/or to compare existing protocols with respect to their efficiency [9]. Since many parts of a client-server database system are already fixed by its core architecture and by hardware parameters, the essential algorithm behind a protocol can only optimize a few efficiency-related parameters. These include:

- the number and the size of messages transferred between client and server,
- the degree to which message processing happens synchronously or asynchronously, respectively and
- the probability of transaction aborts.

A protocol can also strive to optimize server-side processor time consumption. However, for modern hardware settings, the client-server network and the server-side disks represent the most critical resources. Thus, even if a protocol slightly improves or worsens CPU cost, the effect on system efficiency remains small.

The two leading classical transactional cache protocols are CBL [8,16] and OCC [1]. CBL is a lock-based protocol which has a moderate message overhead and very low transaction abort rates. OCC is an optimistic backward-validating protocol which offers very low message overhead but can cause (intolerably) high transaction abort rates.

In this paper we present a new transactional cache protocol whose message overhead is as low as that of OCC, but at the same time, it offers much lower transaction abort rates. The new protocol is called "Optimistic Caching Timestamp Protocol" (OCTP) and may be considered an extension and an improvement of OCC.

OCTP does not require server-side locks to assert serializability but uses backward-validation instead. OCTP assigns timestamps to transactions at commit time. Moreover, it computes additional information about access conflicts that violate the well-known timestamp rule of timestamp-based protocols. Using this additional information, OCTP can assess more accurately than OCC whether a transaction must be aborted or not. In many cases OCTP can commit transactions which have read stale data elements given that these data elements have not been invalidated "too long ago". In comparison with OCC, this quality is the key to reducing the probability of transaction aborts. E.g., consider two transactions $T_1$ and $T_2$ where $T_1$ has just committed and $T_2$ is still active. Assume further that $T_1$ wrote a data element that $T_2$ read even before $T_1$ had started. Using OCC, $T_2$ would definitely be aborted. Using OCTP, $T_2$ is likely to be committed but also, the resulting history remains serializable.

OCTP can be implemented efficiently with respect to processor time and memory consumption. In particular it does not apply a potentially costly serializability graph test. OCTP cannot assert "perfect" external consistency, but the bounds for its worst-case deviation from perfect consistency are clearly defined and well controllable.

Compared to OCC, OCTP lowers transaction abort rates with respect to read/write conflicts of active transactions. However if there is write/write conflict between two active transactions, OCTP still aborts one of them. To accommodate this problem, we present an improvement of OCTP which uses server-side locks but only for write operations. This extension of OCTP is called "Semi Optimistic Caching Timestamp Protocol" (SOCTP). Under SOCTP a client uses an asynchronous approach to acquire write locks, whenever this is suitable.

The remainder of this paper is structured as follows: Section 2 describes the basic idea behind OCTP. In Section 3 we characterize the key qualities of OCTP using a formal approach and prove that these qualities guarantee serializability. In Section 4 we develop a basic but memory-efficient implementation of OCTP and explain the implementation details. Section 5 compares the complexity of OCTP and OCC and states the tradeoffs of the new protocol with respect to external consistency. Section 6 states some general improvements of the presented implementation and also discusses a more fundamental improvement of OCTP which leads to SOCTP. Section 7 studies important efficiency aspects of OCTP, SOCTP, OCC and CBL using simulation experiments. The experiments follow benchmarking approach which is established by various papers in the field of transactional cache protocols. Section 8 discusses OCTP’s links to related work in detail. The paper closes with a conclusion (Section 9).

### 2 Idea

This Section explains the basic idea behind the new protocol.

Just like OCC, OCTP is optimistic and detection-based according to the taxonomy of [9]. When applying optimistic protocols, different clients might cache and access different versions of the same data element. E.g. one client might have written a certain page while some other client still holds the former version of the same page in its cache. Multiversion serializability theory [4,5] has been developed to reflect just this type of situations and consequently we make use of it in our context. Unlike for many other transactional cache...
protocols, the correctness arguments behind OCTP are not straight forward to understand, and so we resort to a formal approach to study the protocol’s key qualities.

OCTP is timestamp protocol which means that it totally orders transactions using a timestamp function \( ts \). To understand the basics of OCTP, consider the multiversion history \( H_1 \) with the timestamp function \( ts(T_i) = i \) and the version order \( x_0 \ll x_1 \ll x_2 \):

\[
H_1 = r_0[x_0] \ w_1[x_1] \ c_1 \ r_2[x_1] \ w_2[x_2] \ c_2 \ r_3[x_1] \ c_3 .
\]

Here, \( r_3[x_1] \) does not read the last committed version of \( x \) (which would be \( x_2 \)) but an older version. Although the operations \( w_2[x_2] \ll r_3[x_1] \) are ordered according to the timestamp order \( ts(T_2) < ts(T_3) \) the resulting edge in \( H_1 \)'s multiversion serializability graph is \( T_3 \rightarrow T_2 \) and points "backwards" with respect to the timestamp order imposed by \( ts \). In general, we call edges whose direction does not match the transactions’ timestamp order reverse edges. Edges which do match the timestamp order are called regular edges.

Situations such as characterized by \( H_1 \) frequently occur in the context of optimistic transactional cache protocols: E.g., assume there are two clients \( C_1 \) and \( C_2 \) which together produce a prefix of the history \( H_1 \). At first, \( C_1 \) executes \( T_1 \), commits it and stores \( x_1 \) in its local cache. Afterwards \( C_2 \) executes \( T_2 \) and causes a cache miss when reading \( x \). Therefore it fetches \( x_1 \) writes \( x_2 \) and eventually commits. Finally \( C_1 \) starts to execute \( T_3 \), finds \( x_1 \) (a stale version \( x \)) in its cache and performs a read on it.

The multiversion history \( H_1 \) is clearly serializable (or more precisely 1-serializable) and so, none of the 3 transactions would have to be aborted by a protocol. However OCC does always abort transactions which read stale versions of data elements and so OCC would unnecessarily abort \( T_3 \). In contrast, OCTP would commit \( T_3 \).

In order to allow read access to stale versions of data elements for committing transactions, OCTP basically behaves as follows: Using the timestamp function, OCTP is able to distinguish reverse edges and regular edges. When an operation of a transaction \( T_i \) causes a reverse edge, OCTP tracks this and computes the timestamp of the transaction to which the reverse edge refers. OCTP avoids cycles in the resulting serializability graph, by asserting the following invariant: If an operation produces a reverse edge \( T_i \rightarrow T_j \) then no committing transaction with a timestamp larger than or equal to \( ts(T_j) \) may cause regular edges pointing to \( T_i \). Otherwise \( T_i \) will be aborted.

Fig. 2 aims to further illustrate this idea. It captures a multiversion serializability graph for the transactions \( T_1,\ldots,T_6 \) with the timestamp function \( ts(T_i) = i \). The dashed arrows represent reverse edges while the remaining arrows represent regular edges.

The graph edge \( T_3 \rightarrow T_6 \) violates the above stated invariant because of the reverse edge \( T_6 \rightarrow T_3 \). Therefore, it is crossed out in Fig. 2. Similarly, \( T_1 \rightarrow T_4 \) violates the invariant due to the reverse edge \( T_2 \rightarrow T_1 \). But how about \( T_4 \rightarrow T_5 \) ?

It adheres to the stated invariant and still leads to the graph cycle \( T_3 \rightarrow T_6 \rightarrow T_4 \rightarrow T_2 \rightarrow T_1 \). In this case it does not suffice to consider single reverse edges. Instead, the invariant from above needs to be generalized such that one considers paths of reverse edges. In Fig. 2 a path of reverse edges starting from \( T_6 \) leads back to \( T_2 \). Therefore, no transactions with \( ts(T_i) \ll ts(T_j) \) should point to \( T_6 \) which indeed excludes \( T_1 \rightarrow T_6 \) from the graph. By applying the generalized invariant we can eliminate all regular edges in Fig. 2 that lead to graph cycles.

The so-called fitting timestamp function \( ts_{fit}(T_i) \) computes the minimum timestamp of all those transactions that can be reached from a transaction \( T_i \) via paths consisting exclusively of reverse edges. OCTP computes \( ts_{fit} \) dynamically for all active transactions and uses it to assert the above stated generalized invariant. A detailed Definition of \( ts_{fit} \) will be given in Section 3.2.

With the help of \( ts_{fit} \) the generalized invariant can be expressed more formally for multiversion histories:

\[
(T_i \rightarrow T_j \in MVSG \land ts(T_j) < ts(T_i)) \Rightarrow ts(T_i) < ts_{fit}(T_j) .
\]

Here, \( MVSG \) represents a history’s multiversion serializability graph. As we will see below, this is the crucial quality to ensure that OCTP generates serializable histories.

### 3 A Formalism for OCTP

In this section we formalize the key qualities of OCTP and prove that they guarantee serializability. To do so, we make use of a well-established multiversion formalism from [5].

In order to present this paper’s contribution in a sound and self-contained way, Appendix A briefly states those formal basics of multiversion transaction histories, which are most relevant in our context. E.g., Appendix A defines conventional multiversion histories and multiversion serializability graphs according to [5]. Moreover, we formally introduce timestamp functions and recoverable multiversion histories. To keeps things short, we omit the definition of serializable (or more specifically 1-serializable) multiversion histories and assume that the reader is familiar with the corresponding serializability theorem (see [5]).

#### 3.1 Histories for Transactional Cache Protocols

This section clarifies, how the standard multiversion formalism can be utilized to reflect the operations of a transactional cache protocol.
When considering transactional protocols, data elements of a history such as \( x \) or \( y \) represent database pages if the server is a page server according to [7] and objects if the server comes as an object server. Since most parts of this paper’s contribution, it does not matter whether the considered data units are objects or pages, we simply assume that the system deals with pages.

When a database client writes a page \( x \) as part of a transaction \( T_i \), one has to distinguish two cases:

1. The transaction \( T_i \) has already performed a read operation \( r_i[x_h] \) on the version \( x_h \) of page \( x \). In this case the write operation is simply represented by \( w_i[x_h] \).

2. \( T_i \) has not yet read page \( x \). Since for the case of transactional caching, writing always implies a prior read of the page, the client’s write operation must be represented formally by \( r_i[x_h]w_i[x_h] \), where \( x_h \) is some version that the client either reads from its local cache or fetches from the server.

If a client writes the same page more than once as part of the same transaction, we only need to represent it by one write operation \( w_i[x_h] \) in the corresponding history. The reason for this is that, if at all, only the last written version of the page will ever become visible to other transactions.\(^3\)

To model read operations of transactional cache protocols appropriately, it is useful to distinguish cache hits and cache misses:

- At a cache miss, a client transaction \( T_i \) reads the last committed version \( x_h \) of page \( x \) from the server.

- At a cache hit, the client transaction \( T_i \) reads the version \( x_h \) of page \( x \) which is currently in the client’s cache. \( x_h \) might well be different from the last committed version of \( x \), either because \( x_h \) is a stale version of the page or because \( T_i \) has written \( x_h \) itself (and so \( i = k \)).

Note that in the case of an optimistic transactional cache protocol, the order of read and write operations of different transactions cannot always be determined because the clients do not (necessarily) synchronize their access operations with each other. However, it can always be determined which version of page a client reads or writes, but this is sufficient to construct a respective multiversion serializability graph. Commits operations are synchronized at server and so they are totally ordered.

### 3.2 OCTP

This section defines qualities of multiversion histories which are specific to OCTP. Below, we will prove that these qualities lead to serializable histories.

For our protocol we use a version order that arranges versions of data elements according to a timestamp order:

\[^3\text{Remember that we assume that a client can only execute one transaction at a time.}\]

**Definition 1** Let \( H \) be a multiversion history with transactions \( T_1, \ldots, T_n \) and a timestamp function \( \tau \). A version order \( \ll \) is a timestamp version order with respect to \( \tau \) iff \( \forall x \in D : \forall x_i, x_j \in V(x) : (x_i \ll x_j \wedge i \neq j) \Rightarrow \tau(T_i) < \tau(T_j) \).

We expect commits to be totally ordered and choose a timestamp function which arranges timestamps according to the commit-order:

**Definition 2** A multiversion history \( H \) is commit-ordering, iff \( \forall c_i, c_j \in H : c_i \prec c_j \Rightarrow \tau(c_i) < \tau(c_j) \). A timestamp function \( \tau \) for a multiversion history \( H \) is commit-ordering, iff \( \forall c_i, c_j \in H : c_i \prec c_j \Rightarrow \tau(c_i) < \tau(c_j) \).

For clarity, we define the terms “reverse edge” and “regular edge” such as introduced in Section 2 more formally.

**Definition 3** Let \( H \) be a multiversion history with transactions \( T_1, \ldots, T_n \), a timestamp function \( \tau \), a version order \( \ll \) and a resulting serializability graph \( MVSG \). An edge \( T_i \rightarrow T_j \in MVSG \) is called a regular edge, iff \( \tau(T_i) < \tau(T_j) \). Otherwise it is called a reverse edge.

In the following, we assume that a protocol produces commit-ordered and recoverable histories which is easy to realize. Moreover, we apply a commit-ordered timestamp function \( \tau \). In practice this means that timestamps are assigned at commit time. Since one may choose a version order \( \ll \) when constructing a multiversion serializability graph of a history, we apply a timestamp version order. Under these conditions, a reverse edge \( T_i \rightarrow T_j \) can only occur if the following pattern of operations exists in a respective history: \( w_i(x_j) < w_j(x_j) < c_j < r_i(x_j) < c_i \) with \( c_k < c_j \). The pattern states that a committing \( T_k \) has read an invalidated and committed version of a data element \( x \) via \( r_i[x_h] \). The proof of Theorem 1 from below shows that the above pattern indeed characterizes all situations leading to reverse edges.

As explained in Section 2, the fitting timestamp function \( ts_{fa}(T_i) \) computes a timestamp \( \tau(T_i) \) of a transaction \( T_i \) at the end of a path consisting entirely of reverse edges and starting at \( T_i \). If there are several such paths \( ts_{fa}(T_i) \) chooses the one leading to the oldest \( T_j \). \( ts_{fa} \) operates on a history to find corresponding reverse edges and in essence, it tries to match the above stated pattern. \( ts_{fa}(T_i) \) is recursively defined on the set of committing transactions. The definition uses the minimum function to find the oldest \( T_j \) which is reachable from \( T_i \) via reverse edges. If there are no reverse edges starting at \( T_i \), the path length is zero and \( ts_{fa}(T_i) = \tau(T_i) \) follows.

**Definition 4** Let \( H \) be a multiversion history with transactions \( T_1, \ldots, T_n \) and a timestamp function \( \tau \). The fitting timestamp function

\[
\tau : \{ T_i \ | \ i \in \{ 1, \ldots, n \} \wedge c_i \in T_i \} \rightarrow \mathbb{N}
\]

is computed as follows:

\[
\tau(T_i) = \min \left( \{ \tau(T_j) \cup \{ ts_{fa}(T_j) \ | \ \exists w_j[x], r_i[x] \in H : ts(T_k) < \tau(T_j) < ts(T_i) \wedge c_j, c_k \in H \} \right).
\]
Obviously, fitting timestamps are well-defined.

As a simple example of applying $ts_{fu}$, consider the history $H_1$ from Section 2: $H_1$ is obviously commit-ordered and the timestamp function $ts(T_i) = i$ is also commit-ordered. Moreover, $x_9 < x_1 < x_2$ is a timestamp version order for $H_1$. The above stated pattern matches with respect to $w_1[x_1] < c_1 < w_2[x_2] < c_2 < r_1[x_1] < c_3$ and results in the reverse edge $T_3 \rightarrow T_2$. Similarly one obtains $ts_{fu}(T_2) = 2$ when applying the formula from Definition 4. The fitting timestamp for $T_1$ and $T_2$ are $ts_{fu}(T_1) = 1$ and $ts_{fu}(T_2) = 2$.

We are now able to formalize the generalized invariant that was stated in Section 2. If a corresponding history fulfills this quality, we called it “$t$-fitting”.

**Definition 5** Let $H$ be a multiversion history with transactions $\{T_1, \ldots, T_n\}$ and a timestamp function $ts$. Further let $\ll$ be a timestamp version order with respect to $ts$ and $MVSG$ be the corresponding serializability graph for $H$. Then $H$ is $t$-fitting with respect to $ts$ iff

$$\forall i,j \in \{1,\ldots,n\} : (T_i \rightarrow T_j \in MVSG \land ts(T_i) < ts(T_j))$$

$$\Rightarrow ts(T_i) < ts_{fu}(T_j).$$

**Theorem 1** A recoverable, commit ordering and $t$-fitting multiversion history $H$ with a commit-ordering timestamp function $ts$ is serializable (or more specifically 1-serializable).

In the following, we sketch the main idea behind the proof of Theorem 1 with the help of Fig. 3. A detailed proof of the theorem can be found in Appendix B. The goal of our considerations is to find an edge $T_i \rightarrow T_j$ in an imaginary serializability graph cycle $C$, such that $T_i \rightarrow T_j$ violates the quality $t$-fitting of $H$. We start our search for this edge on the transaction $T_k$ whose timestamp is minimal in $C$. Since $C$ consists of at least two nodes, there must be a reverse edge in $C$ pointing to $T_k$. We even look for the longest path of reverse edges in $C$ leading to $T_k$. As illustrated in Fig. 3, this path starts at some transaction $T_j$. Because $C$ cannot consist of reverse edges only, there must also be a regular edge $T_j \rightarrow T_i$ in $C$. Due to the path of reverse edges $T_j \rightarrow \ldots \rightarrow T_k$ it follows that $ts_{fu}(T_j) \leq ts(T_k)$. We also have $ts(T_k) \leq ts(T_i)$ because $T_i$’s timestamp is minimal in $C$. Thus, $ts_{fu}(T_j) \leq ts(T_i)$ follows and so $T_i \rightarrow T_j$ violates the quality $t$-fitting of $H$. Therefore $C$ cannot exist.

Obviously, there is still a gap between the definition of qualities that lead to serializable histories and the specification of respective protocol. Fortunately, a useful implementation which computes $ts_{fu}$ and which asserts $t$-fitting, is not too difficult and will be discussed in the next section.

### 4 Implementation

In this section, we present a basic but memory-efficient implementation of OCTP. To develop the protocol, we first discuss a simple implementation of OCC which serves as a basis. Afterwards it will be extended for the requirements of OCTP.

One major challenge when implementing the check for $t$-fitting in OCTP is to determine whether two conflicting operations lead to a regular or to a reverse edge. In Section 4.3 we will explain the non-trivial rationale behind a related $if$-clause in the presented code for OCTP.

#### 4.1 Base Protocol

As mentioned above, we use a simple variant of OCC as a basis to present an implementation of OCTP. Fig. 4 presents the related Java pseudo-code.

The classes defined in Line 1 to 5 are used to represent operations, messages and pages, respectively. Pages as well as clients are uniquely identified by numbers. The implementation distinguishes four types of messages:

- a message to fetch a page which is not in a client’s cache (FetchPageMsg).
- the server’s response to a “fetch page message”, which is simply the page itself (Page).
- a message that enables a client to try a commit (CommitMsg).
- the server’s response to a commit try (TryCommitResMsg).

From the client’s perspective, all messages are sent synchronously using the method sendSync() method. (The method is not further detailed.) For simplicity we also disregard message loss. The class OCCClient represents the client part of the database system. A client-side application would invoke op() to access a page and tryCommit() to initiate a commit for the current transaction. Afterwards, a new transaction implicitly begins at the client. The field cache represents the client’s cache which maps page numbers to page objects. For simplicity we assume that the cache’s size is unbounded. (In reality the corresponding system would apply a replacement strategy such as LRU.)

The server is represented by an instance of the class OCCServer. For convenience, we assume that there is fixed number of clients which is known to the server and stored

\[\text{\footnotesize{4 In order to represent data types conveniently, the code applies\par\allowbreak{}parametric polymorphism (also known as "generics" in the Java world\par\allowbreak{}[10]).}}\]
in the field CLIENTS. The field dir represents the page directory. It maps page numbers to client IDs and holds the type MultiMap because multiple page numbers may be associated with a single client ID. An entry in dir tells that the referenced page is cached at the referenced client. The field disk helps to model disk access but it is not further detailed. In this context, we also disregard the existence of a server-side page cache. The field invalidPidsPerClient associates a list of page numbers with client IDs. An entry in a respective list means that the corresponding client holds a version of the referenced page in its cache and the page got invalidated by a committed transaction. The server keeps the lists up-to-date and notifies a client about invalidated pages as part of a commit response message.

The methods handleFetchPageMsg() and handleCommitMsg() handle incoming client messages. We assume that there is a global message handler in place at the server, which dispatches client messages and invokes the two methods appropriately.

The presented implementation of OCC lacks many optimizations which are relevant to real-world implementations of the protocol. (E.g. it does not support early aborts such as described in [11].) However, in this section we disregard from these features because we want focus on the core structure of the protocol.

### 4.2 Implementation of OCTP

Using the code from Fig. 4 this section is going to present a basic but memory-efficient version of OCTP. Fig. 5 extends the classes OCCClient and OCCServer accordingly. At the client side the class OCTPClient does not add any new code. At the server side the class OCTPServer mainly introduces additional fields and overrides the method handleCommitMsg(). Moreover, the class Tx is used to store information about committed transactions at the server. The data in a Tx-object includes a transactions’ timestamp (ts), its fitting timestamp (tsfit) and its operations (ops). The use of the field poisoned will be explained below.

The field OCTPServer.nextTs acts as a counter to generate the next timestamp that will be assigned to a validating transaction. The array invalidatingTxsPerClient stores lists of transaction objects, whereby a respective transaction has invalidated a certain page stored at a certain client. The array is organized in the same way as the field OCCServer.invalidatedPisPerClient. E.g. if a client with the ID 1 has cached page 23 and the page got invalidated, there will be a respective entry in the list invalidatedPisPerClient[1] at position p. At the same time the list invalidatingTxsPerClient[1] contains an entry representing the transaction which caused the invalidation. The transaction entry is also stored at position p. If there is more than one invalidating transaction, a respective entry in invalidatingTxsPerClient is guaranteed to refer to the invalidating transaction with the oldest timestamp.

```java
class Op { int page, boolean read; ... }
class FetchPageMsg { int cid; int pid; ... }
class CommitMsg { int cid; List<Op> ops; ... }
class CommitResMsg { List<int> invalidPids; boolean abort; ... }
class Page { ... }
class OCCClient { int cid; // A client’s ID Map<int,int> page->cache; // Maps page numbers to pages List<Op> ops = new List<Op>(); // OCTPClient.tends the classes a basic but memory-efficient version of OCTP. Fig. 5 extends the classes OCCClient and OCCServer accordingly. At the client side the class OCTPClient does not add any new code. At the server side the class OCTPServer mainly introduces additional fields and overrides the method handleCommitMsg(). Moreover, the class Tx is used to store information about committed transactions at the server. The data in a Tx-object includes a transactions’ timestamp (ts), its fitting timestamp (tsfit) and its operations (ops). The use of the field poisoned will be explained below.

The field OCTPServer.nextTs acts as a counter to generate the next timestamp that will be assigned to a validating transaction. The array invalidatingTxsPerClient stores lists of transaction objects, whereby a respective transaction has invalidated a certain page stored at a certain client. The array is organized in the same way as the field OCCServer.invalidatedPisPerClient. E.g. if a client with the ID 1 has cached page 23 and the page got invalidated, there will be a respective entry in the list invalidatedPisPerClient[1] at position p. At the same time the list invalidatingTxsPerClient[1] contains an entry representing the transaction which caused the invalidation. The transaction entry is also stored at position p. If there is more than one invalidating transaction, a respective entry in invalidatingTxsPerClient is guaranteed to refer to the invalidating transaction with the oldest timestamp.
```
transaction occupies position 0 in the list. The code in handleCommitMsg() ensures that recentlyCommitted contains at most RECENT_MAX entries.

The size limitation of recentlyCommitted is the key to the protocol’s memory efficiency. OCTP ensures that no active transaction causes a reverse edge referencing a committed transaction which is not contained in recentlyCommitted. Otherwise the active transaction will be aborted.

By adjusting RECENT_MAX the protocol can be made more or less tolerant with respect to active transactions reading stale versions of pages. A larger value for RECENT_MAX allows potentially committing transactions to read rather “old” invalidated pages. This has a positive influence on the transaction abort rate but potentially lowers the degree of external consistency. A related discussion can be found in Section 5.2. If RECENT_MAX is set to 0, the system never tolerates access to stale pages. As a matter of fact, at RECENT_MAX = 0 the presented OCTP and OCC implementations behave identically. OCC may therefore be considered a special case of OCTP.

The method OCTPServer.handleCommitMsg() behaves as follows: In Lines 25 to 27 it creates local copies of the invalidation lists addressing the client that sent the commit message. In Line 30 the method creates a transaction object tx for the transaction Ti that is to be validated. tx is assigned a timestamp which also serves as the initial value for the fitting timestamp (Line 31). Afterwards all operations of Ti are inspected in order to compute the correct value for tsFit(Ti). If an operation o of Ti accessed an invalidated page p, the algorithm determines the transaction Ti with the oldest timestamp that invalidated p and stores it in invTx. Obviously a respective invalidation leads to a reverse edge Ti → Tj. If o is a write operation, then Tj must be aborted (Line 35). (The reasons for this are detailed in Section 4.3.)

Using the field invTx.poisoned the algorithm checks if the invalidating transaction Tj is still contained in the list recentlyCommitted. If not, the reverse edge Ti → Tj cannot be accepted and Tj must be aborted. Moreover, if Tj is in the list recentlyCommitted, it might itself hold a reverse edge to an even older transaction which is not in recentlyCommitted anymore. In this case Tj’s poisoned-field is also true and Tj will be aborted as well.

In Line 39 Ti’s fitting timestamp is updated, if the detected reverse edge to Tj produces a path of reverse edges referencing an older transaction than tx.tsFit. After passing line 40 the algorithm has considered all reverse edges from Ti to recently committed transactions and so the value tx.tsFit is correct. Based on this result, the next part of the validation ensures that t-fitting holds for Ti.

From Line 41 to Line 57 all operations of Ti are examined to see whether they cause regular edges between recently committed transactions and Ti. Line 42 determines the invalidating transaction Th for the currently considered operation o and assigns it to invTx. If no Th exists, invTx is set to null. Line 45 loops over all recently committed transactions. Let Tj be such a transaction, then Line 46 loops over Tj’s operations. The Lines 47 to 53 check if the considered operations of Ti and Tj produce a conflict and if the conflict leads to a regular edge. The rationale behind the check is intricate and will be discussed separately in Section 4.3. If the algorithm detects a regular edge, it checks the invariant for t-fitting in Line 54 and aborts Tj if necessary.

After Ti has passed the check for t-fitting, it is ready to be committed (Line 58). At first, the tx-object is assigned Ti’s operation list and tx.poisoned is set to false. Afterwards Tj is added to the list of recently committed transactions (Line 61). If recentlyCommitted has become too long, Line 64 removes the oldest transaction from the list. The removed transaction’s poisoned-flag must be set to true (Line 66). If a recently committed transaction valTx holds a reverse edge to the removed one, valTx does become “poisonous” too (Line 69 to 70).

The last part of the algorithm updates the data structures invalidatingTxsPerClient and invalidatingTxsPerClient. This process is similar to the one in OCCServer.handleCommitMsg(). In Line 78, the protocol checks if there already exists an invalidation entry for the considered page and the considered client. If so, the corresponding invalidating transaction is guaranteed to have an older timestamp than Ti because it must have committed before Ti. In this case the data structure invalidatingTxsPerClient must remain unchanged because as mentioned above, a respective entry in invalidatingTxsPerClient[occid] is supposed to refer to the oldest invalidating transaction.

4.3 Handling Reverse and Regular Edges

One important question regarding the OCTP implementation is how the protocol can efficiently determine whether a conflict between a recently committed transaction and a transaction under validation produces a regular edge or a reverse edge. In Fig. 5 the Lines 50 to 53 ensure that a considered conflict indeed produces a regular edge. In the following we explain the rationale behind this code on a formal basis. To do so the next definition clarifies the event of an invalidation with respect to multiversion histories.

Definition 6 Let H be a multiversion history with transactions T = {T1, ..., Tn}. A transaction Ti ∈ T invalidates a version xk of a data element x iff wi[xi], ci ∈ H and xk ≪ xi.

Let T be the transaction which the server is validating and Tj be a recently committed transaction. Further, let pi and qj be two conflicting operations with pi ∈ Ti and qj ∈ Tj. Obviously, we have ts(Tj) < ts(Ti) because timestamps are assigned in commit order. When the algorithm reaches line 50, there can only occur one of the following situations with respect to Ti and Tj:

1. pi = r[xi] is a read operation and qj = w[j][xj] is a write operation: There might be a third committed transaction Th which has invalidated xj. If there is no such Th, then qj
Fig. 5 A Simple Implementation of OCTP (Based on Fig. 4). Includes Server-Side Memory Management for Validated Transactions

has the last implemented version of x and so \(j = k\) or \(x_j \ll x_k\). Therefore \(p_i \parallel q_j\) results in a regular edge. If, on the other hand, a \(T_k\) exists, we require \(T_k\)'s timestamp to be minimal (amongst all corresponding candidates). Assume \(p_i \parallel q_j\) results in a regular edge, then \(ts(T_k) < ts(T_j)\) must hold. If \(ts(T_k) \leq ts(T_j)\) holds, \(h = j \lor k\) will hold because of the chosen version order. But also \(x_j \ll x_k\) must hold because of the regular edge. This leads to \(x_j \ll x_k\) and so \(T_k\) could not have invalidated \(x_k\) (contradiction).

Assume alternatively that \(p_i \parallel q_j\) results in a reverse edge. In this case \(T_j\) also invalidates \(x_j\). Then \(ts(T_j) \geq ts(T_k)\) must hold because otherwise \(T_j\)'s timestamp as chosen above would not be minimal.

Altogether, this leads to the conclusion that \(p_i \parallel q_j\) results in a regular edge if and only if \(T_k\) does not exist or \(ts(T_j) < ts(T_k)\) holds. The condition is covered by the Lines 52 and 53 of Fig. 5.

2. \(p_i = w_j[x_i]\) is a write operation and \(q_j = r_j[x_j]\) is a read operation: \(T_k\) must have committed since the protocol asserts recoverability and so \(x_j \ll x_k\) follows due to the chosen commit order. Thus, the resulting edge is a regular edge. This case is covered by the expression (valOp.read && !op.read) in Line 50 of Fig. 5.

3. \(p_i = w_j[x_i]\) is a write operation and also \(q_j\) is a write operation: For this case, note that the protocol always implies that a page, which is written by \(T_j\), will first be read by \(T_j\). So with respect to \(p_i\) we actually have \(p_i = r_j[x_i] w_j[x_i]\). The operations \(w_j[x_i]\) and \(w_j[x_j]\) do not conflict but, \(r_j[x_j]\) and \(w_j[x_j]\) do. Thus the situation is the same in 1) and it is also covered by the Lines 52 and 53 of Fig. 5.

If an invalidating transaction \(T_k\) exists with respect to \(x_k\), then one obtains the reverse edge \(T_k \rightarrow T_j\) because of \(r_j[x_j]\). Moreover the operation \(r_j[x_j]\) (which must precede \(w_j[x_j]\)) and \(w_j[x_j]\) cause a regular edge \(T_k \rightarrow T_j\) because \(x_j \ll x_k\) holds. Therefore \(T_j\) should be aborted if a related \(T_k\) exists. The algorithm follows this policy due to the check !op.read in Line 35 of Fig. 5.

The code from Fig. 5 does not store version tags of pages to determine the direction of conflict edges but uses information about invalidating transactions instead. The advantage of the latter option is that on average, it is less memory consuming than version tags. When using version tags, the system has to store one tag per entry in the page directory plus one tag per every database page (which is stored on disk). It was argued in [1] that the added version tags might considerably increase the physical size of a database.

Therefore, the approach from Fig. 5 only stores entries for invalidated cached pages at the server. This is more efficient, since usually, the client caches contain a lot more up-to-date pages than invalidated pages. In essence our ap-
5 Important Qualities of OCTP

5.1 Complexity of OCTP Versus OCC

In the following we briefly compare the memory and the runtime complexity of OCC and OCTP.

In comparison with OCC the (server-side) memory overhead of OCTP is low: Let \( c \) be the number of clients and \( S \) be the maximum size of a client’s cache. OCC stores at most \( c \cdot S \) entries in the page directory. At worst, \( c \cdot S \) entries will be found in the server-side invalidation lists.

In addition, OCTP stores at most \( \text{RECENT\_MAX} \) Tx-objects in recentlyCommitted. Since in practice \( \text{RECENT\_MAX} \) is a small constant, the related memory overhead is small. Apart from this, no more than \( c \cdot S \) Tx-objects will be found in invalidatingTxsPerClient and a respective Tx-object references no operation list unless it is also contained in invalidatingTxsPerClient.

As a result, the memory cost for OCC lies in \( O(c \cdot S) \) and the one for OCTP lies in \( O(c \cdot S + \text{RECENT\_MAX}) \).

To compare the runtime complexity of OCC and OCTP we assume that the algorithms from Fig. 4 and Fig. 5 are improved by some basic optimizations, which we describe below. The only interesting part for comparing the protocols’ runtime complexity is obviously the validation of a transaction. Let \( \text{ops} \) be a transaction’s number of operations. In Line 58, OCC loops over every operation and checks if the operation is contained in an invalidation list. If we use a hash table to represent an invalidation list, this process has the complexity \( O(\text{ops}) \). Afterwards \( O(c) \) invalidation lists must be updated per operation. Thus, the total complexity for OCC is \( O(\text{ops} + \text{ops} \cdot c) = O(\text{ops} \cdot c) \).

In the case of OCTP, the computation of \( t_{tsfa} \) has the complexity \( O(\text{ops}) \), again assuming that an invalidation list is represented by a hash table (Lines 32 to 40). During the test for \( t \)-fitting the loops from Line 41 and Line 45 cause the complexity \( O(\text{ops} \cdot \text{RECENT\_MAX}) \). In an optimized version of the algorithm the loop of Line 46 can be entirely avoided. Instead, one can use a hash table to determine if a recently committed transaction has accessed a certain page. Since the complexity for the commit part of OCTP is the same as for OCC, the total runtime complexity for OCTP results in \( O(\text{ops} + \text{ops} \cdot \text{RECENT\_MAX} + \text{ops} \cdot c) = O(\text{ops} \cdot (c + \text{RECENT\_MAX})) \).

In practice, even a relatively small value for \( \text{RECENT\_MAX} \) will provide OCTP’s benefits of reduced transaction abort rates. In this context, Section 7.2.3 studies the quantitative influence of \( \text{RECENT\_MAX} \) on a system’s transaction abort rate. It turns out that \( \text{RECENT\_MAX} = 100 \) is usually sufficient. Other experiments from Section 7 reveal that the added computational complexity of OCTP does not affect system efficiency.

5.2 External Consistency

External consistency asserts that a valid serialization order of (committed) transactions does not deviate “too much” from the realtime commit order of the respective transactions. At perfect external consistency a database system guarantees the existence a valid serialization order which is identical to the transactions’ realtime commit order [1]. Giving a formal definition of (non-perfect) external consistency is beyond the scope of this paper. Fortunately we are still able to discuss OCTP’s effect this consistency criterion.

External consistency might suffer under OCTP because with respect to a serialization order, an active transaction \( T_i \) that reads an invalidated page version (and commits) will range behind the transaction \( T_j \) that invalidated the corresponding page version. In the worst case, \( T_i \)’s position in the serialization order might lag up to \( \text{RECENT\_MAX} \) timestamps behind if \( T_j \) is stored at position 0 in the list recentlyCommitted from Fig. 5.

A page version read by \( T_i \) might have been invalidated up to \( \text{RECENT\_MAX} \) times given that all transactions in recentlyCommitted have written the respective page. Although in practice, this case is extremely unlikely, it might still be desirable to set a lower limit \( l \) for the number times a page version may be invalidated but without changing the value of \( \text{RECENT\_MAX} \). The OCTP implementation of Fig. 5 can be easily extended to cover this requirement. When updating the array invalidPidsPerClient (in Line 74 to 82) the system can also maintain counters that store, how often a particular page version, which is cached at a certain client, got invalidated. This information can be used during the validation process of a transaction in Line 35 of Fig. 5. A transaction will then also be aborted, if it has read a page version which got invalidated more than \( l \) times.

The above considerations lead to the conclusion that if perfect external consistency represents premium requirement, OCTP is not the protocol of choice. In all other cases though, OCTP’s worst case deviations from perfect external consistency are clearly defined and can be well controlled.

As already mentioned in Section 4.2, the protocol never tolerates access to stale pages, if \( \text{RECENT\_MAX} \) is set to 0. In this case OCC and OCTP behave identically. (Using the formalism from Section 3 this can also be proven.)

6 Improvements

6.1 General Improvements

For real-world scenarios, the protocol implementations from Fig. 4 and 5 should be optimized in several ways:

- As mentioned in the previous section, one may apply more efficient data structures e.g. to manage invalidation lists.
One can integrate the support of early aborts according to [11]. When early aborts are enabled, a transaction is partially validated whenever it performs server access. The partial validation is based on the operations that a transaction has executed so far. Early aborts avoid wasted work resulting from transactions which have to aborted anyway but they increase a transaction’s total validation cost.

As proposed in [11] client-side before-images of written pages ensure that a page must not be discarded from a client cache if the client’s current transaction gets aborted.

As opposed to the code from Fig. 4 and 5, information about invalidated pages maybe piggy-backed on every message addressing a client and not just commit response messages. [11] describes this strategy for OCC and classifies it as “eager reactive”.

In general, all of these improvements are well-understood from other contributions in the field and their application to OCTP is straightforward. Therefore, we do not discuss them here in depth. Note however, that in order to get useful results with respect to the experiments from Section 7, we realized all of the above mentioned improvements.

As has been done for CBL and OCC, one may also derive an adaptive variant of OCTP. When using adaptive protocols such as AOC or ACBL [11, 19] the granularity of data elements may dynamically change between pages and objects in order to minimize message overhead and the chance conflicts. Developing and studying an adaptive variant of OCTP is part of our future work.

6.2 SOCTP

As explained in Section 4.3, OCTP always aborts a transaction which (read and) wrote an invalidated version of a page. To avoid this type of aborts, OCTP can be extended such that write access to pages is coordinated by means of server side write locks.

The related locking mechanism is partially asynchronously and will explained below. To a certain extend it is similar to the locking mechanism of CBL but it only affects write operations. An (explicit) read operation of a transaction is still performed optimistically and without any locks. However for write operations, clients try to acquire exclusive locks which they release at the end of a transaction. When compared to CBL, the peculiarity of the locking mechanism is that lock requests may be performed synchronously or asynchronously and therefore, we focus on this aspect in the description from below. The idea of mixing synchronous and asynchronous lock requests is inspired by [12, 15] (see also Section 8).

The resulting variant of OCTP is called “Semi-Optimistic Caching Timestamp Protocol” (SOCTP) because it only handles read/write conflicts in a purely optimistic way.

When a client C writes a page using SOCTP, it always tries to acquire a respective write lock from the server. If the page is not in C’s cache, it simply performs a lock request as part of the page request. The more interesting case is a cache hit: At a cache hit the client sends a lock request to the server either in a synchronous or in an asynchronous manner. (What request mode it chooses when, will be explained below.)

If the client sends a synchronous lock request, it waits for a response message from the server. In this context, the response message either grants the requested lock or tells the client to abort its current transaction. A transaction abort might be necessary for two reasons: Either the server an early abort check for the transaction fails or the transaction causes a deadlock when waiting for the requested write lock. If some other transaction already owns the write lock requested by C, then the server delays the response message until the lock becomes available. (The other transaction releases the lock at termination.)

If C sends an asynchronous lock request, it does not wait for a response message from the server but proceeds immediately with its current transaction. When the server gets the asynchronous lock request it checks whether another client already owns the lock. If the lock is available, then client C obtains it and the server creates a corresponding entry in the page directory. If some other client owns the lock, C’s current transaction will be aborted. If the server informs C about the transaction abort by means of an asynchronous message.

To decide whether C acquires a write lock synchronously or asynchronously, C maintains a data structure which is called a write warning list. The write warning list contains information about all pages which are cached by C and which were written by active transactions. If a page that C intends to write is referenced in its write warning list, then it performs a synchronous lock request for that page. Otherwise it sends an asynchronous lock request.

The write warning list is updated via information which is piggy-backed on messages sent from the server to C. So, whenever the server sends a message to C, it is also in charge of delivering the latest write warnings. The server knows early about write intentions from other clients due to their respective lock requests (which they send synchronously or asynchronously). Still, C’s write warning list might be out of date e.g. because some other client has obtained the write lock for a page but the server has not yet had chance to inform C about it. In this case, if C intends to write this page, it sends an asynchronous lock request where it would have better sent a synchronous one. As explained above, C’s current transaction will then be aborted.

SOCTP avoids many of situations where OCTP would cause an abort due to a write/write conflict of two active transactions. If such a situation is likely to occur, SOCTP makes one of the participating clients wait until the other client’s transaction has terminated. This is done via a synchronous lock request. Whenever it is more likely that a transaction will obtain the requested lock right away, it does not make sense to wait for the server’s lock acknowledg-
ment. In this case the client takes the risk and performs an asynchronous lock request. Note that OCTP as well as SOCTP still must abort a transaction, which has read an invalidated version of page and which tries to write the same page in a following operation.

SOCTP incurs a larger message overhead than OCTP but most of the additional messages are asynchronous and so they do not block a corresponding transaction. Moreover, most of the additional synchronous messages indeed prevent transaction aborts. In following section we study the related effects in detail.

7 Evaluation

7.1 Experiments

7.1.1 Simulation System

As in former studies on transactional cache protocols we use a simulation system to evaluate the quality of OCTP. The simulation system models a page-based client-server database system such as described in Section 1. Fig. 6 presents the overall structure of the simulator: A client consists of a transaction generator which produces sequences of access and commit operations according to a workload model. The workload specifies the fixed length of committing transactions, the fixed probability of transaction restarts (in case a transaction was aborted), the probability of read and write operations and the related distributions for referenced pages.

The client protocol manager executes transaction operations and either produces a hit at its local cache or delegates the request to the server. The client cache has a fixed size and applies an LRU replacement strategy. Moreover, a client has a CPU component for modeling client-side processor time consumption. The number of clients can vary – it is equivalent to the number of concurrent transactions causing load on the simulated database system.

A CPU component manages incoming instruction requests that represent certain protocol-related activities using a FIFO wait queue. A protocol related activity, such as handling a cache hit, is associated with a fixed amount of instructions and depending on the CPU speed, the system derives the amount of CPU-related simulation time which it charges for performing the activity.

If a client cannot perform an operation locally, it delegates the operation to the server via an emulated network. Much as the CPU components, the network charges simulation time for transferring messages between clients and the server and it is also represented by a FIFO wait queue. The message cost depends on the simulated network bandwidth, the message size and a random network delay. As in [12, 15, 18] a network delay happens at a certain probability but with a fixed delay time. A delay does not affect waiting times of other messages in the network’s FIFO wait queue.

At the server the related protocol manager handles incoming operation requests. Page read operations are performed via a server-side cache. The cache has a fixed size and applies an LRU replacement strategy. At a page miss, the related operation is forwarded to one of the disk components. Page writes cause a cache reference but are always delegated to a disk component because the system models a write through strategy for pages.

There are several server-side disks which are all represented by independent FIFO wait queues. Every database page is uniquely assigned to a disk on which it is stored and every disk is in charge of about the same amount of pages. Disk access times are determined by a uniform random distribution with a fixed lower and upper bound.

Apart from disk access cost, the execution of a server-side operation incurs cost at one of the server-side CPUs. The related CPUs share a single FIFO wait queue. The system distinguishes "user activities" such as validating a page and "system activities". The CPUs give higher priority to system activities. Moreover, sending a message from a client to the server or vice versa does include CPU cost on both sides of the communication channel. For every message there is a fraction of fixed CPU cost and a variable CPU cost, where the latter one depends on the message size. Disk access and message sending are the only types system activities.

Table 1 states the values for the system parameters discussed in the previous paragraphs. The parameter setup represents a situation where the network is slow and message delays are frequent and relatively long. This is typically the case when client and server communicate via a wide area network, e.g. the Internet. The related values are taken from [15]. In this case the network becomes the bottleneck and message overhead dominates system performance. Note that the situation may change when the network is fast. In the latter case the server-side disk components tend to become the system bottleneck and protocols with low message overhead loose a significant advantage over protocols causing higher message overhead.

The transaction protocols such as implemented for the experiments all support early aborts according to [11]. They also store before images of an active transaction’s written pages at the client side [11]. This way the related pages can be easily restored at the client in case of a transaction abort.

As in [1, 11] we charge server-side CPU cost for a validation step at optimistic protocols. With respect to OCC there are at least ops validation steps at a transaction’s commit time, where ops is the transaction’s number of access operations. Since early aborts are enabled, a transaction might also be partially validated before it commits which results in additional validation steps (see also Section 6). With respect to OCTP, there are ops \* \( \text{RECENT_MAX} \) validation steps at commit time and potentially additional validation steps resulting from early abort checks. Due to its nature, CBL does not cause validation cost. As in [1, 11] we do not charge cost for CBL’s deadlock detection. Similarly, there is no cost involved for deadlock detection under SOCTP.

\[5 \text{Regardless of early abort checks, all ops operations must be revalidated at commit time.}\]
When varying the number of clients for an experiment, we set $\text{RECENT}_\text{MAX} = 100$ for OCTP and SOCTP. As it will discussed in Section 7.2.3 the value 100 is sufficient to provide the intended reduction of abort rates under OCTP and SOCTP.

7.1.2 Workloads

For the sake of a more compact presentation we only discuss the results of two important workloads which are well-known from other contributions [6,8,9,11,18].

The UNIFORM workload has no per-client locality and a high degree of data contention. At this workload, every client accesses every database page with the same probability. The HOTCOLD workload has a high degree of locality per client and a moderate amount of data sharing and data contention among clients. Most of the time a client accesses data from its private region but it may also access data from the rest of the database. Clients may update data in both regions.

Fig. 2 presents the parameters characterizing the workloads. Similar settings for the two workloads can be found in [6,8,9]. The parameter $\text{restartProb}$ states the probability that a transaction that got aborted is restarted (with the same sequence of access operations). According to [11] and [12] a restart probability of 1 favours OCC whereas $\text{restartProb} = 0$ favours CBL. As in [12] we therefore chose the value $\text{restartProb} = 0.5$ for the HOTCOLD workload. Concerning the UNIFORM workload, the value 0.5 still favours OCC (and OCTP) due to a heavily increased cache hit rate for restarted transactions. Therefore we set $\text{restartProb} = 0$ for the UNIFORM workload.

To obtain a data point, the system was first warmed up until all data structures had reached (up to an epsilon) a fixed point with respect to their filling size under the given workload. Afterwards the measuring phase began and lasted until a thousand commits were observed. Given a certain number of clients, this procedure was repeated ten times with different random seeds for the transaction generators. An entry in the resulting graph displays the average result of the six data points as well as error bars for the 90% confidence interval.

Table 1 System Parameters for the Simulation Experiments

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{pageSize}$</td>
<td>Size of a database page</td>
<td>4 KByte</td>
</tr>
<tr>
<td>$\text{dbSize}$</td>
<td>Size of database in pages</td>
<td>2000</td>
</tr>
<tr>
<td>$\text{clientCacheSize}$</td>
<td>Client cache size in pages</td>
<td>250</td>
</tr>
<tr>
<td>$\text{serverCacheSize}$</td>
<td>Server cache size in pages</td>
<td>1000</td>
</tr>
<tr>
<td>$\text{clientCPU}$</td>
<td>Client CPU speed</td>
<td>300 MIPS</td>
</tr>
<tr>
<td>$\text{serverCPU}$</td>
<td>Server CPU speed</td>
<td>300 MIPS</td>
</tr>
<tr>
<td>$\text{serverDisks}$</td>
<td>Number of Server CPUs</td>
<td>2</td>
</tr>
<tr>
<td>$\text{maxDiskTime}$</td>
<td>Minimum time for page access on disk</td>
<td>3ms</td>
</tr>
<tr>
<td>$\text{maxDiskTime}$</td>
<td>Maximum time for page access on disk</td>
<td>6ms</td>
</tr>
<tr>
<td>$\text{serverDisks}$</td>
<td>Number of server disks</td>
<td>8</td>
</tr>
<tr>
<td>$\text{bandWidth}$</td>
<td>Network bandwidth</td>
<td>80Mbps</td>
</tr>
<tr>
<td>$\text{delayProb}$</td>
<td>Probability of msg. delay</td>
<td>0.5</td>
</tr>
<tr>
<td>$\text{delayTime}$</td>
<td>Time of msg. delay</td>
<td>10ms</td>
</tr>
<tr>
<td>$\text{fixedMsgInstrs}$</td>
<td>Fixed number of instrs. per msg.</td>
<td>20000</td>
</tr>
<tr>
<td>$\text{perByteMsgInstrs}$</td>
<td>Additional instrs. per msg. byte</td>
<td>4</td>
</tr>
<tr>
<td>$\text{registerInstrs}$</td>
<td>Instrs. for (un)registering page at client cache</td>
<td>300</td>
</tr>
<tr>
<td>$\text{lookupInstrs}$</td>
<td>Instrs. for page lookup at client cache</td>
<td>300</td>
</tr>
<tr>
<td>$\text{validateInstrs}$</td>
<td>Instrs. for a validation step</td>
<td>600</td>
</tr>
<tr>
<td>$\text{dirLookupInstrs}$</td>
<td>Instrs. for access of page dir.</td>
<td>600</td>
</tr>
<tr>
<td>$\text{diskAccessInstrs}$</td>
<td>Instrs. for disk access</td>
<td>5000</td>
</tr>
<tr>
<td>$\text{earlyAborts}$</td>
<td>Enable early aborts for optimistic protocols</td>
<td>yes</td>
</tr>
<tr>
<td>$\text{saveBeforeImage}$</td>
<td>Cache replacement strategy at client and server</td>
<td>LRU</td>
</tr>
<tr>
<td>$\text{replStrategy}$</td>
<td>When written committed pages are saved to disk (write through or write back)</td>
<td>Write through</td>
</tr>
<tr>
<td>$\text{toDiskStrategy}$</td>
<td>Max. number of recently committed transactions which are stored in memory</td>
<td>100 (or varying)</td>
</tr>
</tbody>
</table>

7.2 Results

7.2.1 Number of Messages

This paragraph discusses the average number of messages sent per committing transaction with respect to the studied protocols.

Analysis: Before presenting the experimental results, we briefly develop an analytical model, which aims to predict the measured numbers for the UNIFORM workload. The model is simple but still important because it helps to explain the experimental results and increases trust in their correctness. Surprisingly, previous studies of transactional cache protocols have not tried to substantiate their simulation results this way.

Let $c$ be the number of clients running concurrent transactions under the UNIFORM workload. Let $p_c = \text{coldWrtProb}$ be the probability that an operation performs a
Table 2 Workload Parameters for the Simulation Experiments

<table>
<thead>
<tr>
<th>Workload</th>
<th>UNIFORM</th>
<th>HOTCOLD</th>
</tr>
</thead>
<tbody>
<tr>
<td>transSize</td>
<td>20 pages</td>
<td>20 pages</td>
</tr>
<tr>
<td>hotBounds</td>
<td>-</td>
<td>p to p + 49 with p = 50 \cdot i \in {0, \ldots, 39}</td>
</tr>
<tr>
<td>coldBounds</td>
<td>All of DB</td>
<td>Rest of DB</td>
</tr>
<tr>
<td>hotAccProb</td>
<td>0</td>
<td>0.8</td>
</tr>
<tr>
<td>coldAccProb</td>
<td>1</td>
<td>0.2</td>
</tr>
<tr>
<td>coldWrtProb</td>
<td>0.2</td>
<td>0.2</td>
</tr>
<tr>
<td>perPageInstrs</td>
<td>30000</td>
<td>30000</td>
</tr>
<tr>
<td>thinkTime</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>restartProb</td>
<td>0</td>
<td>0.5</td>
</tr>
<tr>
<td>c (Number of Clients)</td>
<td>{1, 2, 5, 10, 15, \ldots, 40}</td>
<td></td>
</tr>
</tbody>
</table>

read access and \( ops = transSize \) be the number of operations per transaction. Given that a client cache fills up completely, the corresponding hit rate is \( p_{hit} = \text{clientCacheSize/dbSize} \).

For OCC and OCTP, estimating the number of messages \( m_{OCC/OCTP} \) for a committing transactions is straight forward. \( m_{OCC/OCTP} \) only depends on the number of operations \( ops \) per transaction and the cache hit rate:

\[
m_{OCC/OCTP} = 2 \cdot ops \cdot (1 - p_{hit}) + \frac{2}{\text{Commit Operation}}
\]

At CBL, a client \( C \) needs to acquire a write lock from the server for almost every write operation (and this results in two additional messages). The case where the client already possesses the write lock is ignored because it is rare: The client must have written the respective page before but in the same transaction. This event is unlikely since \( ops \ll D \).

About \( p_{hit} \cdot (c - 1) \) clients cache the page that \( C \) is about to write, which results in callback messages. Thus, one can estimate the number of messages \( m_{CBL} \) as follows:

\[
m_{CBL} = 2 \cdot ops \cdot \left( \frac{(1 - p_{hit})}{\text{Cache Miss}} + \frac{(1 - p_{r}) \cdot p_{hit}}{\text{Cache Hit but Writing}} \right) + \frac{2}{\text{Commit Operations}}
\]

In contrast to CBL, SOCTP generates lock request messages for write operations but no callbacks. A synchronous lock request from a client causes two messages but an asynchronous request causes only one message unless the server immediately aborts the respective transaction.

Assume a client \( C \) intends to write a page \( p \). The probability that \( p \) has not yet been written by another client’s active transaction is about \( P_{\text{unlocked}} = (1 - \frac{1}{\text{dbSize}})^{0.5 \cdot \text{ops} \cdot (1 - p_{r}) \cdot (c - 1)} \). The formula implies that the remaining \( c - 1 \) clients have each finished their active transactions about half way. For simplicity, we assume that write warning lists are always up-to-date. Thus, the average number of messages for a lock request of a page cached by \( C \) is \( P_{\text{unlocked}} \cdot 1 + (1 - P_{\text{unlocked}}) \cdot 2 = 2 - P_{\text{unlocked}} \cdot 1 \). The total number of messages \( m_{SOCTP} \) sent for a committing transaction is about

\[
m_{SOCTP} = \text{ops} \cdot \left( 2 \cdot (1 - p_{hit}) + \frac{(1 - p_{r}) \cdot p_{hit}}{\text{Cache Miss}} \right) + \frac{2}{\text{Commit Operation}}.
\]

Fig. 7 presents analytical results for the number of message per committing transaction under the UNIFORM workload in a graph.

Simulation Results: Fig. 8 presents the measured results for the number of message per committing transaction under the UNIFORM workload. As one can see, there is an excellent match with the analytical forecasts from Fig. 7.\(^6\) With regard to our analysis from above, it is not surprising that OCTP indeed produces the same number of messages as OCC. Since cache hits are rare under the UNIFORM workload, SOCTP rarely produces extra messages to request write locks at the server. (Remember that under SOCTP, additional write lock requests only occur if a write operation causes a cache hit.) The rising number of messages under CBL is caused by callback messages.

Fig. 9 presents similar results as Fig. 8 but for the HOT-COLD workload. Again, OCC and OCTP cause about the same number of messages. SOCTP produces more messages.

\(^6\) We decided to present the analytical and the experimental results in separate figures because the graphs match so well that otherwise they would be hard to discern.
than OCTP and OCC because under the HOTCOLD workload, cache hits are frequent and only a write operation hitting the cache will cause an additional lock request message. However, most of the additional messages produced by SOCTP are asynchronous thanks to the use of write warning lists. Therefore these messages incur little blocking at a requesting client. We measured that from all write lock request messages produced by SOCTP, no more than 2% were synchronous under the HOTCOLD workload.

7.2.2 Throughput

Fig. 10 shows the throughput of the four protocols in committing transactions per second under the UNIFORM workload. For the given system configuration with a slow network and fast server disks (see Section 7.1.1) the optimistic protocols clearly outperform CBL. OCTP and SOCTP attain an even higher throughput than OCC because of their lower transaction abort rates. (Details on transaction abort rates follow in Section 7.2.3).

When more than 35 clients run concurrent transactions under the UNIFORM workload, then for CBL, the client caches do not fill up to their maximum size of 250 cachable pages. The effect also occurs for OCC, OCTP and SOCTP but at slightly higher numbers of concurrent clients. The effect is a result of competing page roll-ins and page removals at client caches. In [13], we studied this phenomenon in detail and explained the related system behavior by means of a simple but accurate analytical model. Under the UNIFORM workload the throughput of CBL is very sensitive to lowered cache filling sizes and so the performance degrades for \( c \geq 35 \). Surprisingly the phenomenon of lowered cache filling sizes has never been examined by other studies in the field transactional caching, although its existence is even detectable in some of the graphs of [9].

Fig. 11 depicts transaction throughput under the HOTCOLD workload. Again, the optimistic protocols outperform CBL. The performance difference between OCC and OCTP is not as distinct as under the UNIFORM workload.
because the HOTCOLD workload causes more moderate transaction abort rates even for OCC. Moreover, the small message overhead for write lock requests hardly affects the performance of SOCTP when compared to the other two optimistic protocols.

### 7.2.3 Abort Rate

Fig. 12 and 13 present transaction abort rates in “aborts per commits” under the UNIFORM workload and under the HOTCOLD workload, respectively. In both cases, OCTP offers considerably lower abort rates than OCC and SOCTP lowers these rates even further. However, CBL still offers better results. Thus when comparing SOCTP and CBL, there remains a trade-off between high performance and low abort rates but it is less extreme as in the case of CBL and OCC. In order to summarize the relative improvements of OCTP and SOCTP, Table 3 shows the average reduction of abort rates (in “aborts per commits”) with respect to OCC across client numbers that range between 5 and 40.

Fig. 14 presents the transaction abort rate when the parameter $RECENT_{MAX}$ from Section 4 is varied under the UNIFORM workload. The number of clients remained fixed at $c = 25$ for all data points. As one can see, values as little as $RECENT_{MAX} = 20$ provide good improvements of the abort rate already. At $RECENT_{MAX} = 0$ OCTP behaves exactly as OCC and causes the same transaction abort rate. SOCTP behaves slightly better already because it avoids aborts related to certain write/write conflicts. At $RECENT_{MAX} = 50$ the abort rate becomes stable for both OCTP and SOCTP. The graphs “OCTP, $t$-Fitting Violated” and “SOCTP, $t$-Fitting Violated” display the number of transaction aborts resulting from violations of the quality “$t$-fitting”. For SOCTP, almost
all of the transaction aborts at $\text{RECENT}_\text{MAX} \geq 50$ result from violations of "$t$-fitting".

As described in Section 2 and 3 "$t$-fitting" is the essential criterion that OCTP and SOCTP assert for committing transactions. Bigger values for $\text{RECENT}_\text{MAX}$ enable a reasonably accurate check for $t$-fitting but they cannot avoid the actual violations of this criterion. Therefore, the curve "SOCTP, $t$-Fitting Violated" initially rises and then reaches about the level of the curve "SOCTP". For OCTP, the difference between the curves "OCTP" and "OCTP, $t$-Fitting Violated" at $\text{RECENT}_\text{MAX} \geq 50$ is due to the additional aborts resulting from write/write conflicts of active transactions. (As explained before, OCTP does not prevent this type of aborts.)

8 Related Work

In this Section we organize aspects of related work as follows: At first, we try to categorize SOCTP and OCTP according to a well-known taxonomy of transactional cache protocols established in [9]. Afterwards, we discuss two more recently published protocols, namely ADCC and AACC. At last, we consider relationships between OCTP and pure server-based transaction protocols.

According to [9] OCTP is best classified as a "detection based protocol" which in the worst case, defers its validity check "until commit". Moreover, change notification hints are given "after commit" and the respective remote update action is "invalidation". Therefore OCTP appears in the same position of the taxonomy as "Cache Locks" [17].

SOCTP cannot be well categorized along the taxonomy of [9] because it is detection based with respect to read/write conflicts and avoidance based with respect to write/write conflicts. On top of this, the avoidance based aspect uses both "synchronous" and "asynchronous validity checks".

As it became clear in Section 4, OCTP is an improvement and an extension of OCC. In contrast to OCC it can validate transactions even though they have accessed invalidated cached pages.

As mentioned in Section 6.1, there exist adaptive variants for CBL and OCC which are called AOCC or ACBL [19,11]. Adaptive protocols dynamically change the granularity of managed data elements between page and object in order to lower message sizes and the chance conflicts. We believe that it is well possible to develop an adaptive variants of OCTP and SOCTP. The corresponding task is part of our future work.

ADCC [18] is the most recently published transactional cache protocol which uses direct client-to-client communication in order to offload the server and to increase total system performance. The authors of [18] compared ADCC with CBL using simulation: ADCC consistently outperformed CBL while offering the same low abort rate as CBL. Direct client-to-client communication forms the basis of ADCC's efficiency improvements but it is also its major handicap. Client-to-client communication is not desirable if clients should remain anonymous with respect to each other. (E.g. this is typically the case for rich client applications that communicate over the Internet.) Also, if the number of clients is high and/or fluctuating, there is considerable communication and management overhead to ensure that every client is registered at all its peers.

AACC [12,15] is an asynchronous, adaptive and pure server-based protocol whose abort rates are as low as those of ACBL. Using simulation, the authors of [12,15] showed that AACC outperforms ACBL for a given, fixed number of concurrent clients. Moreover, AACC also outperforms AOCC for the same number of clients but only under SH-HOTCOLD workload. The authors state that these results depend on the relative speed of the server CPUs and disks as well as on the probability of transaction restarts (with respect to aborted transactions).

Unfortunately a direct comparison between AACC and OCTP is difficult because we have not implemented AACC and so there are no experimental results available for a comparison. Note that AACC is an adaptive protocol while OCTP is not. However, the descriptions of AACC in [12,15] make it obvious that the protocol can be easily modified to work in a non-adaptive way. (Interestingly, this point counters a whole line of arguments from [18] claiming that AACC is not applicable in the context of pure page servers because it is adaptive.)

In the following, we discuss essential differences of AACC, OCTP and other protocols on a theoretical basis.

---

7 SH-HOTCOLD is a variant of the HOTCOLD workload catering to adaptive protocols.
Table 4 How Different Protocols Handle Important Conflict Situations

<table>
<thead>
<tr>
<th>Situation</th>
<th>rₙ₁ &lt; w₂₁, T₁ not committed, T₂ tries to commit first</th>
<th>w₁₂ &lt; w₂₁, T₁ and T₂ not committed</th>
</tr>
</thead>
<tbody>
<tr>
<td>CBL</td>
<td>not possible, T₁ blocks at w₂₁ instead</td>
<td>T₂ blocks at w₂₁</td>
</tr>
<tr>
<td>AACC</td>
<td>T₂ blocks at commit until</td>
<td>T₁ blocks at w₂₁ or either T₁ or T₂ aborts</td>
</tr>
<tr>
<td>OCC</td>
<td>T₂ commits, T₁ aborts</td>
<td>either T₁ or T₂ aborts</td>
</tr>
<tr>
<td>OCTP</td>
<td>T₂ commits, T₁ likely to commit</td>
<td>either T₁ or T₂ aborts</td>
</tr>
<tr>
<td>SOCTP</td>
<td>T₂ commits, T₁ likely to commit</td>
<td>either T₁ or T₂ aborts</td>
</tr>
</tbody>
</table>

while assuming a non-adaptive version of AACC is at hand. Table 4 characterizes important situations where conflicts between active transaction lead to blocking or transaction aborts. The table compares the behaviour of several protocols regarding these situations.

To keep things short, we only discuss the situation w₁₂ < w₂₁ from Table 4 for AACC: AACC acquires write locks in an asynchronous or deferred manner. This means, a client tries to acquire a respective write lock from the server but it does not wait for it and continues its transaction processing instead. Problems arise if two concurrent transactions T₁ and T₂ write the same data element x. If client T₁ writes x much earlier than T₂, then the client executing T₂ will receive a related callback message before it executes w₂₁ and acknowledge this message. Afterwards T₂ will have to wait until T₁ has terminated (and released the write lock). If T₂ writes x before the callback message arrives, then either T₁ or T₂ will be aborted.

As explained in Section 6.2, SOCTP handles the situation w₁₂ < w₂₁ from Table 4 in a similar way as AACC. However, SOCTP does not use synchronous callbacks to inform concurrent clients of write lock requests. Instead it piggy-backs the related information as write warning lists on other messages. As opposed to callbacks, write warnings incur less potential blocking but they are more likely to arrive “too late”, which may lead to higher transaction abort rates than callbacks.

When compared to CBL, AACC considerably improves the time a transaction must wait for a server response or for other transactions to finish certain operations (e.g. to commit). However, waiting still occurs under AACC. The optimistic protocols from Table 4 trade in waiting times for potential transaction aborts. E.g. under OCC and OCTP, transactions almost never wait for each other. If the network is the major system bottleneck, then communication latency increases waiting times for CBL and AACC more than for the optimistic protocols. High network latency frequently occurs when client server communication is Internet-based. In such a case, OCTP and SOCTP offer very good performance because they produce little waiting times but also moderate transaction abort rates.

OCTP is remotely related to multiversion timestamp transaction protocols such as introduced in [14] and described in [5]. A multiversion timestamp scheduler treats an incoming operation rₙ₁ of a transaction T₁ as follows: To execute rₙ₁, the scheduler tries to find the version x₂₁ written by a committed transaction T₂ such that T₂ has the largest timestamp with ts(T₂) ≤ ts(T₁). The protocol implies that an underlying database system stores several versions of the data element x along with a version tag. If x₂₁ from above is not stored in the database anymore (e.g. because it is too old) then the scheduler will reject rₙ₁.

The situation is different in the context of OCTP: A client-based transaction T₁ simply reads the version x₂₁ that is cached at the client. In this case, the server-side protocol manager does not have “the luxury” of choosing an appropriate x₂₁ for T₁. Instead it adjusts T₁’s fitting timestamp accordingly and checks if T₁ “can get away” with reading x₂₁. Also note, that our protocol does neither store version tags nor several page versions in the server database.

[2] presents an optimistic timestamp-based multiversion concurrency control scheme. In essence, it is an optimistic variant of the multiversion timestamp protocol introduced by [14] and therefore the differences to OCTP are alike.

The authors of [2] suggest the concept of “dynamic timestamps”: A dynamic timestamp is not necessarily assigned at transaction begin or commit but when the first conflict with another active transaction occurs. Dynamic timestamps eliminate certain cases where a plain one-version timestamp protocol aborts transactions. E.g. consider the (one-version) history rₙ₁[w₂₁[y]c₁r₁[y]]. If timestamps are assigned at a transaction’s first operation, then one obtains ts(T₁) < ts(T₂) and rₙ₁ will be rejected under the one-version timestamp protocol. (Similar problems can occur if timestamps are assigned at commit time instead.) A dynamic timestamp is related to the fitting timestamp from Section 3 because both are computed on the basis of the prefix of a transaction history. However our approach differs from the concept suggested in [2], because OCTP still relies on fixed timestamps which are assigned at commit time. Moreover fitting timestamps rely on a recursive definition while dynamic timestamps do not.

9 Conclusion and Future Work

This paper has presented two new optimistic transactional cache protocols – OCTP and SOCTP. OCTP may be considered an extension and improvement of OCC – the currently leading optimistic transactional cache protocol. SOCTP is in turn an improvement of OCTP. As opposed to OCC, OCTP and SOCTP considerably reduce transaction abort rates while attaining higher transaction throughput.

In contrast to all other known transactional cache protocols, OCTP and SOCTP allow many transactions to commit which have read invalidated versions of pages from a
client’s cache. This quality is the key to the reduced abort rates. When two active transaction cause a write/write conflict, OCTP still must abort one of them. To accommodate this, SOCTP efficiently uses server-side locks but only with respect to write operations.

Since the correctness of the new protocols is not trivial to see, we applied multiversion transaction theory in order to prove that they indeed produce serializable histories. Multiversion transaction theory is well suited in our the context because it can reflect the fact that different clients may concurrently cache and access different versions of the same page.

The new protocols essentially differ from existing, pure server-based multiversion transaction protocols. Unlike a classical multiversion scheduler, the server-side manager for OCTP and SOCTP can rarely choose the version of a page to communicate over the Internet. In addition to OCTP, SOCTP the network forms the bottleneck of the system. This situation clearly dominates the one of CBL whenever the transaction was already sufficient to provide the benefits of transactional caching. Much as in the case of OCTP, a respective SGT protocol might reduce the probability of transaction aborts even further. However there is a tradeoff between the potentially reduced abort rate and the added cost and complexity for cycle checks regarding the serializability graph. The related dependencies deserve in-depth considerations and represent a part of our future work.

Acknowledgements Thanks to Michael Klein for a thorough reading of this paper.

A Multiversion Histories

Definition 7 Let \( \{ T_1, \ldots, T_n \} \) be a set of transactions. A multiversion history \( H \) is defined as \( H = \{ h(p) \mid p \in \bigcup_{i=1}^{n} T_i \} \) with a partial ordering relation \( \prec \). Further, the function \( h \) must fulfill the following criteria:
\[
\begin{align*}
&\forall a_i, c_i, w_i[x] \in \bigcup_{i=1}^{n} T_i : h(a_i) = a_i \land h(c_i) = c_i \land h(w_i[x]) = w_i[x], \\
&\forall r_i[x] \in \bigcup_{i=1}^{n} T_i : \exists j \in \{1, \ldots, n\} : h(r_i[x]) = r_j[x], \\
&\forall i \in \{1, \ldots, n\} : \exists p, q \in T_i : p < q \Rightarrow h(p) < h(q), \\
&\forall w_i[x], r_i[x] \in \bigcup_{i=1}^{n} T_i : w_i[x] < r_i[x] \Rightarrow h(r_i[x]) = r_i[x], \\
&\forall r_i[x] \in \bigcup_{i=1}^{n} T_i : h(r_i[x]) = r_j[x] \Rightarrow (i = 0 \lor \exists w_i[x] \in H : w_i[x] < r_j[x]), \\
&\forall r_i[x] \in \bigcup_{i=1}^{n} T_i : (h(r_i[x]) = r_j[x] \land i \neq j \land c_j \in H) \Rightarrow c_i \in H.
\end{align*}
\]
An \( x_i \) is called a version of the data element \( x \).

The definition assumes that prior to any write operation, there already exists an initial version \( x_0 \) for every data element \( x \).

Definition 8 Let \( H \) be a multiversion history for the transactions \( T = \{ T_1, \ldots, T_n \} \). Let \( D \) be the set of data elements in \( H \), so \( D = \{ x \mid \exists T_i \in T : r_i[x] \in T_i \lor w_i[x] \in T_i \} \) and let \( V(x) \) be the set of versions of \( x \) in \( H \), so \( V(x) = \{ x_0 \} \cup \{ x \mid \exists w_i[x] \in H \} \). A version order \( \ll \) establishes for every data element \( x \in D \) a total order of its versions, such that \( x_0 = \) the smallest version:
\[
\forall x \in D : x_0, x_j \in V(x) \setminus \{ x_0 \}, \\
x_0 \ll x_j \Leftrightarrow (i \neq j \Rightarrow x_i \ll x_j \lor x_j \ll x_i).
\]

Definition 9 Let \( H \) be a multiversion history for the transactions \( T = \{ T_1, \ldots, T_n \} \) and \( \ll \) be a corresponding version order. The serializability graph \( MVSG \subseteq D^2 \) for \( H \) and \( \ll \) is given be the following predicate:
\[
(T_i, T_j) \in MVSG :\Leftrightarrow c_i \in T_i \land c_j \in T_j \land \exists x_i[w_i[x]], w_m[x_m] \in H : \\
(i \neq j \land m = i \land k = j) \lor \\
(i \neq j \land m = i \land l = j \land x_m \ll x_l) \lor \\
(i \neq j \land k = i \land m = j \land x_k \ll x_m).
\]

Instead of writing \( (T_i, T_j) \in MVSG \) we simply write \( T_i \ll T_j \). If one of the last two disjunctive clauses holds, then \( T_i \ll T_j \) is called a version order edge.

Definition 10 Let \( H \) be a multiversion history with transactions \( T = \{ T_1, \ldots, T_n \} \); \( ts : T \rightarrow N \) is a timestamp function iff \( \forall i, j \in \{1, \ldots, n\} : ts(T_i) = ts(T_j) \Rightarrow i = j \).

Definition 11 A multiversion history \( H \) is recoverable, iff \( \forall w_i[x], r_j[x] \in H : c_j \in H \Rightarrow c_i \in H \).

A promising approach that we want to investigate is the development of an optimistic (multiversion) serializability graph test protocol tailored to the requirements of transactional caching. Much as in the case of OCTP, a respective protocol should inspect only a limited list of recently committed transactions. In comparison with OCTP and SOCTP an SGT protocol might reduce the probability of transaction aborts even further. However there is a tradeoff between the potentially reduced abort rate and the added cost and complexity for cycle checks regarding the serializability graph. The related dependencies deserve in-depth considerations and represent a part of our future work.
Proof of Theorem 1

Proof Let $H$ consist of the transactions $\{T_1, \ldots, T_n\}$. When constructing $H$’s multiversion serializability graph $MVSG$ we use the timestamp version order $\leq$ of the t-fitting history $H$.

At first, we prove that for a reverse edge $T_i \rightarrow T_j$ in $MVSG$, the quality $ts_f(T_j) \leq ts(T_i)$ must hold. To do so, we consider the three disjunctive clauses from Definition 9 which produce edges in the serializability graph.

If the reverse edge $T_i \rightarrow T_j$ is created by the first disjunctive clause of Definition 9, one has $x_i \leq x_j$ which implies $ts(T_j) < ts(T_i)$ because $\leq$ is recoverable. However, $x_j < x_i$ implies $ts(T_i) < ts(T_j)$ because $ts$ is commit ordering, but the latter contradicts the assumption that $T_i \rightarrow T_j$ is a reverse edge. Thus, this case cannot occur.

If the reverse edge $T_i \rightarrow T_j$ is created by the second disjunctive clause of Definition 9, one has $x_i \leq x_j$ which implies $ts(T_j) < ts(T_i)$ because $\leq$ is a timestamp version order. Again this contradicts the assumption that $T_i \rightarrow T_j$ is a reverse edge. Thus the reverse edge cannot result from the second disjunctive clause of Definition 9.

If the reverse edge $T_i \rightarrow T_j$ is created by the third disjunctive clause of Definition 9, one has $r_i[x_j] < r_j[x_i]$ and $x_j < x_i$. The timestamp version order $\leq$ implies $ts(T_i) < ts(T_j)$, which gives $ts(T_i) < ts(T_j) < ts(T_k)$ because $T_i \rightarrow T_j \rightarrow T_k$ is a reverse edge. Since $H$ is recoverable, $c_i \in H$ must hold and therefore $ts_f(T_j) \leq ts(T_i)$ follows according to Definition 9.

The rest of the proof shows that a cycle in the serializability graph is impossible. Assume $H$’s serialization graph $MVSG$ was cyclic. A cycle in $MVSG$ has at least a length of 2, because according to Definition 9, $i \neq j$ holds for a corresponding edge $T_i \rightarrow T_j$. A cycle in $MVSG$ consists of at least one reverse edge. Otherwise one would obtain a cycle $T_i \rightarrow \ldots \rightarrow T_k$ with regular edges only and so $ts_f(T_k) < ts(T_i)$ would hold (contradiction).

Now, let $C$ be a cycle in $MVSG$ and $T_0$ be the node in $C$ with the smallest timestamp. There must be a reverse edge $T_0 \rightarrow T_1 \in C$ for some $T_0$ because otherwise $T_1$’s timestamp would not be minimal with respect to $C$. Further, let $T_1 \rightarrow \ldots \rightarrow T_n$ be the longest acyclic path in $C$ consisting entirely of reverse edges. Then, there must be an edge $T_n \rightarrow T_1 \in C$ which is a regular edge. If $T_n \rightarrow T_1$ did not exist, $C$ would consist of reverse edges only and one would obtain $C = T_n \rightarrow \ldots \rightarrow T_i$ with $ts(T_i) < ts(T_k)$ (contradiction).

Since $T_n \rightarrow T_1$ is a regular edge, one has $ts_f(T_i) < ts(T_i)$ and even $ts(T_i) < ts_f(T_k)$ due to $H$ being t-fitting. Since $T_i \rightarrow \ldots \rightarrow T_n$ only consists of reverse edges, an inductive application of the considerations from above results in $ts_f(T_k) \leq ts(T_i)$. This leads to $ts(T_k) < ts(T_i)$ and contradicts the assumption that $T_k$’s timestamp is minimal in $C$. Thus, $MVSG$ must be acyclic. □

References