The Serial Safety Net: Efficient Concurrency Control on Modern Hardware

Tianzheng Wang Ryan Johnson Alan Fekete* Ippokratis Pandis♯

University of Toronto
{tzwang,ryan.johnson}@cs.toronto.edu

*The University of Sydney
alan.fekete@sydney.edu.au

♯Cloudera
ippokratis@cloudera.com

ABSTRACT

Concurrency control (CC) algorithms must trade off strictness for performance, with serializable schemes generally paying high cost—both in runtime overhead such as contention on lock tables, and in wasted efforts by aborting transactions—to prevent anomalies. We propose the serial safety net (SSN), a serializability-enforcing certifier for modern hardware with substantial core count and large main memory. SSN can be applied with minimal overhead on top of various CC schemes that offer higher performance but admit anomalies, e.g., snapshot isolation and read committed.

We demonstrate the efficiency, accuracy and robustness of SSN using a memory-optimized OLTP engine with different CC schemes. We find that SSN is a promising approach to serializability with low abort rates and robust performance for various workloads.

1. INTRODUCTION

Concurrency control (CC) algorithms interleave read/write requests from multiple users simultaneously, while giving the (perhaps imperfect) illusion that each transaction has exclusive access to the data. Serializable executions are those that are equivalent to *some* serial executions, which would be desirable for users, because they never have anomalies (e.g., lost update) and can preserve integrity constraints over the data. Enforcing a cycle-free transaction dependency graph is a necessary and sufficient condition to achieve serializability, and is the focus of this work. Some CC schemes, such as two-phase locking (2PL) and serializable snapshot isolation (SSI) [5], forbid all cycles to guarantee serializability. But they also forbid many valid serializable schedules.

These serializable CC schemes are often implemented with centralized data structures, which hardly scale on today's massively parallel, large main memory hardware due to contention (e.g., on lock tables [14, 22]). That is, modern hardware, where it is common to fit the whole working set—even the whole database—in memory and I/O operations are completely out of the critical path, puts more pressure on CC and makes serializable CC schemes even more impractical. Many recent systems thus opt for lightweight optimistic concurrency control (OCC) [17, 29], or sacrifice serializability in favor of high concurrency and high throughput. For example, PostgreSQL's default isolation level is read committed [25], although SSI has been implemented to ensure full serializability [24]. OCC is known to be unfriendly to heterogeneous workloads that have a significant amount of analytical operations [13], an important application of modern memory-optimized systems. On the other hand, sacrificing serializability allows dependency cycles and permits data corruption from concurrency effects. In some most widely-used database systems, non-serializable CC is the default, and sometimes there is no available isolation level that guarantees serializability.

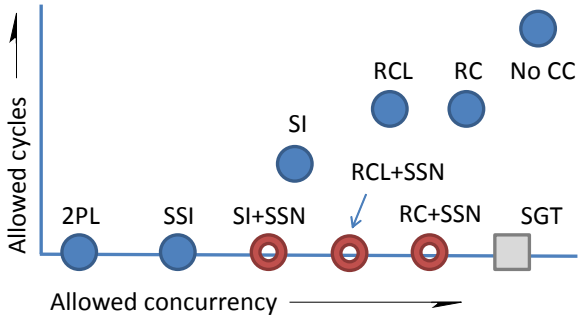


Figure 1: Relative merits of existing CC schemes (solid dots) vs. the serial safety net (hollow dots).

Figure 1 illustrates the relative strictness vs. performance trade-off for several well-known CC schemes. At one extreme, strict 2PL ensures serializability but offers low concurrency because readers and writers block each other. At the other extreme, a system with no CC whatsoever (No CC) offers maximum concurrency but admits often intolerable anomalies (e.g., dirty reads and lost writes). Read committed (RC) and its lock-based variant (RCL), offer much stronger semantics than No CC, with a low performance cost, and are often used in practice. Snapshot isolation (SI) makes a very attractive compromise, offering reasonably strict semantics and fairly high performance, while SSI offers full serializability but lowers concurrency significantly. Fully precise serialization graph testing (SGT) [6] allows all (and only) cycle-free executions, but is impractical as every commit requires an expensive search for cycles over the dependency graph.

DaMoN'15, June 1, 2015, Melbourne, VIC, Australia

 ISBN 978-1-4503-3638-3/15/06…$15.00
DOI: http://dx.doi.org/10.1145/2771937.2771949

This paper proposes the serial safety net (SSN), an efficient, general-purpose mechanism to enforce serializability on top of a variety of CC schemes that forbid dirty reads and lost writes. Most CC schemes, including SI and RC, meet these requirements. SSN does not dictate access patterns—the underlying CC scheme does that—but instead tracks dependencies and aborts transactions that might close a dependency cycle. SSN admits false positives, but is much more accurate than the prior practical serializable CC schemes (e.g., 2PL and SSI). As illustrated by the figure, SSN guarantees serializability with concurrency levels not drastically worse than the underlying CC scheme. In particular, RC+SSN allows higher concurrency than 2PL and SSI. See Appendix C for a comparison between SSN and other cycle prevention schemes on their relative merits.

SSN consists of two parts: an easily-computed *priority timestamp* $\pi(T)$ for transaction T that summarizes "dangerous" transactions that committed before T but must be serialized after T, and a conservative validation test that is applied when T commits: if transaction T tries to commit at time $c(T)$, and U has already committed and had a conflict with T (i.e., U must be serialized before T), then $\pi(T) \leq c(U) \leq c(T)$ is forbidden (because U might also need to be serialized after T, forming a cycle in the dependency graph). We prove that maintaining this *exclusion window* suffices to prevent all cycles in the serial dependency graph, and thus guarantees that all executions are serializable.

One unique aspect of SSN is that it works in spite of bugs, omissions, or unanticipated behaviors in the underlying CC scheme (so long as the basic requirements still hold). This protection is important, because CC schemes tend to be complex to implement, and bugs can lead to subtle problems that are difficult to detect and reproduce. Unanticipated behaviors are even more problematic. For example, a read-only anomaly in SI arises only if a reader arrives at exactly the wrong moment [8]. This anomaly was not discovered until SI had been in use for many years. Assuming SSN is implemented correctly—hopefully achievable, given its simplicity—bugs or unexpected behaviors in the CC scheme that would confuse applications, will instead trigger extra transaction aborts caused by SSN. The application sees only serializable executions that preserve data integrity.

SSN can be implemented efficiently with minimal overhead. Our evaluation on a four-socket, 24-core Xeon server with 64GB of main memory shows that SSN can scale as well as the underlying CC scheme (e.g., RC and SI). Compared to SSI, SSN can provide ∼50% lower abort rates in general, ∼2x higher commit rates for update-intensive transactions, and robustness against retrying aborted transactions.

We next give background on serial dependency graphs that we use throughout the paper to understand serializability properties, followed by the design and implementation of SSN. We then present evaluation results and conclude. Finally, an appendix that covers the formal proof of SSN and related discussions is provided for interested readers.

2. SERIAL DEPENDENCY GRAPHS

We model the database as a multi-version system [1], which consists of a set of items. Each transaction comprises a sequence of reads and writes, each dealing with a single item. In this model, each item is seen as a totally-ordered sequence of *versions*. A write always generates a new version at the end of the item's sequence; a read returns some version in the item's sequence. Insertions and deletions are represented using a special "invalid" value. Insertions are updates that replace invalid versions. Deletion flags an item as invalid without physically deleting it, and the item can continue to participate in CC if needed. The physical deletion is performed in background once the record is no longer needed [9].

Accesses by transaction T generate *serial dependencies* that constrain T's place in the global partial order of transactions. Serial dependencies can take two forms:

- $T_i \xleftarrow{w:x} T$ (read/write dependency): T accessed a version that T_i created, so T must be serialized after T_i.
- $T \xleftarrow{r:w} T_j$ (anti-dependency): T read a version that T_j overwrote, so T must be serialized before T_j.

A read implies a dependency on the transaction that created the returned version, and an anti-dependency from the transaction that (eventually) produces the next version of the same item (overwriting the version that was read). A write implies a dependency on the transaction that generated the overwritten version. Accessing two versions of the same item (e.g., a non-repeatable read) within a transaction implies a serialization failure: $T_1 \xleftarrow{r:w} T_2 \xleftarrow{w:r} T_1$.

We use $T \leftarrow U$ to represent a serial dependency of either case: either $T \xleftarrow{w:x} U$ or $T \xleftarrow{r:w} U$, and we say that T is a direct predecessor of U (i.e., U is a direct successor of T). The set of all serial dependencies between committed transactions forms the edges in a directed graph G, whose vertices are committed transactions and whose edges indicate required serialization ordering relationships. When a transaction commits, it is added to G, along with any edges involving previously committed transactions. T may also have *potential edges* to uncommitted dependencies, which will be added to G if/when those transactions commit.

Note that our notation puts the arrowhead of a dependency arrow near the transaction that must be serialized before the other; this is the reverse of the usual notation (as in [1]) but it makes the arrowhead look similar to the transitive effective ordering relation symbol we define next.

We define a relation $\prec$ for G, such that $T_i \prec T_j$ means T_i is ordered before T_j along some path through G (i.e., $T_i \leftarrow \ldots \leftarrow T_j$); we say that T_i is a predecessor of T_j (or equivalently, that T_j is a successor of T_i). When considering *potential edges*, we can also speak of potential successors and predecessors; these are transactions for which the potential edges (along with edges already in G) require them to be serialized after (or respectively before) T.

The key result of the serialization theory is that acyclic G ensures that the execution is serializable [1]. A cycle in G produces $T_i \prec T_j \prec T_i$, and indicates a serialization failure (because G then admits no total ordering).

The simplest cycles involve two transactions and two edges:

- $T1 \xleftarrow{w:x} T2 \xleftarrow{w:x} T1$. $T1$ and $T2$ saw each others' writes (isolation failure).
- $T1 \xleftarrow{w:x} T2 \xleftarrow{r:w} T1$. $T2$ saw some, but not all, of $T1$'s writes (atomicity failure).
- $T1 \xleftarrow{r:w} T2 \xleftarrow{r:w} T1$. $T1$ and $T2$ each overwrote a value that the other read (write skew).

In our work, a central concept is the relationship between the partial order of transactions that G defines, and the total

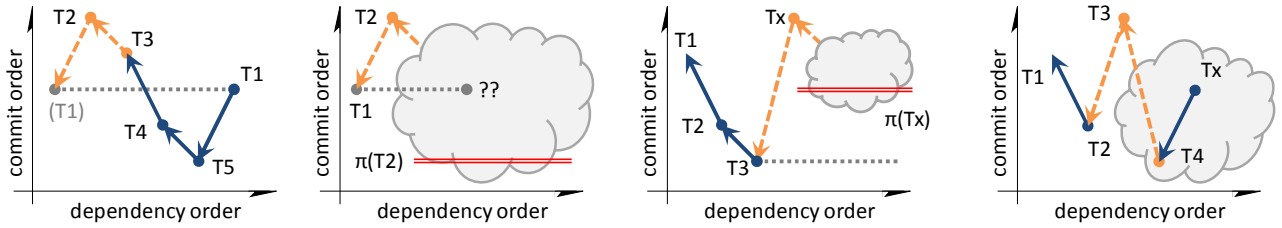


Figure 2: A pictorial motivation and description of SSN. Subsets of G are shown in a serial-temporal layout where forward and back edges always have positive and negative slopes, respectively.

order defined by their commit times. At the moment transaction T commits, we take a timestamp (a physical or virtual clock measured in time units, or simply a monotonically increasing sequence number), and call it $c(T)$. An edge in G is a *forward edge* when the predecessor committed first, and a *back edge* when the *successor* committed first. A forward edge can be any type of dependency, but (for the types of CC algorithms we deal with, with write isolation) back edges are always read anti-dependencies (where the overwrite committed before the read). We denote forward and back edges as $T_1 \overset{f}{\leftarrow} T_2$ and $T_1 \overset{b}{\leftarrow} T_2$, respectively. Let us write $T_0 \overset{b*}{\leftarrow} T_k$ for the reflexive and transitive back-edge situation where T_0 is reachable from T_k without following any forward edges, e.g., $T_0 \overset{b}{\leftarrow} T_1 \overset{b}{\leftarrow} T_2 \overset{b}{\leftarrow} T_3 \ldots \overset{b}{\leftarrow} T_{k-1} \overset{b}{\leftarrow} T_k$; note that $T \overset{b*}{\leftarrow} T$ always holds.

3. SSN: THE SERIAL SAFETY NET

Given a CC scheme that admits cycles in the serial dependency graph, SSN can be layered on top as a pre-commit protocol to abort transactions that will form cycles if committed. Although SSN can be overlaid on various CC schemes, we require the CC scheme forbid lost writes and dirty reads (unless it is the transaction reading its own writes), which is effectively as strong as RC.

In addition to the commit timestamp $c(T)$, SSN associates two other timestamps, $\pi(T)$ and $\eta(T)$, with T at commit. These values are, respectively, low and high watermarks used to detect conditions that might indicate a cycle. We define $\pi(T)$ as the commit time of T's oldest successor U reached through a path of back edges:

$$\pi(T) = min\left(c(U) : T \overset{b*}{\leftarrow} U\right) = min\left(\left\{\pi(U) : T \overset{b}{\leftarrow} U\right\} \cup \{c(T)\}\right)$$

The first equation captures the definition, where T's successor U that overwrote versions read by T, committed first, forming a back edge which represents a read anti-dependency. The second, equivalent recursive equation, shows how this would be computed from only the immediate successors of a transaction in G, without traversal of the whole graph.

Note that $\pi(T) \leq c(T)$, and the values of $c(T)$ and $\pi(T)$ are fixed once T has committed; π will not change because committed T only acquires new successors via forward edges, which do not influence $\pi(T)$.

The essence of SSN is a certification that prevents a transaction T from committing if an exclusion window check is violated for some direct predecessor U:

Definition 1. *A dependency edge $U \leftarrow T$ in G violates the exclusion window of T if $\pi(T) \leq c(U) \leq c(T)$.*

Intuitively, the first inequality certifies whether T's predecessor U could also be a successor (because U committed after T's oldest successor), indicating a cycle in G. When implementing exclusion window checks, we can use two observations to simplify the process. First, we need only consider predecessors that committed before T (the second inequality), which means the check can be completed during pre-commit of T (regardless of what happens later). Second, of those predecessors that committed before T, we only need to examine the most recently-committed one. Using the following definition of $\eta(T)$, an exclusion window violation occurs if $\pi(T) \leq \eta(T)$, so T must abort:

$$\eta(T) = max\left(\left\{c(U) : U \overset{f}{\leftarrow} T\right\} \cup \{-\infty\}\right)$$

We next illustrate visually why tracking $\pi(T)$ and enforcing exclusion windows might prevent cycles in G. Formal descriptions are provided in Appendix A.

Examples. Figure 2(a) gives a *serial-temporal* representation of a cycle in G. The x-axis gives the relative serial dependency order (as implied by the edges in G); the y-axis gives the global commit order. In this figure, forward edges have positive slope (e.g., $T5 \leftarrow T1$), while back edges have negative slope (e.g., $T4 \leftarrow T3$). A transaction might appear more than once (connected by dashed lines, e.g., $T1$ in Figure 2(a)), if a cycle precludes a total ordering.

Visually, it is clear that $T1$ violates the exclusion window of $T2$ (because $\pi(T2) = c(T5) < c(T1) = \eta(T2)$); Figure 2(b) depicts information that is available to $T2$ as local knowledge. Without knowing the predecessors of $T1$, $T2$ must assume that $T1$ might also be a successor. Figure 2(c) demonstrates a case where the exclusion window is satisfied: $T3$ committed before $\pi(Tx)$—even earlier than Tx's oldest successor—so $T3$ could not be a successor and Tx will not close a cycle if committed; $T1$ cannot have any predecessor newer than $\pi(Tx)$ as that would violate its own exclusion window; any later transactions that links $T1$ with Tx would suffer an exclusion window violation.

Finally Figure 2(d) illustrates a false positive case, where $T3$ aborts due to an exclusion window violation, even though no cycle exists. We note, however, that allowing $T3$ to commit would be dangerous: some predecessor to $T1$ might yet commit with a dependency on $T4$, closing a cycle without triggering any additional exclusion window violations.

Safe retry. Users submit transactions supposing they will commit, however, the underlying CC scheme might abort

transactions due to various reasons, such as write-write conflicts. Ideally, the CC scheme should ensure a transaction's successful commit: at least does so after retrying (the "safe retry property" [24]), unless it is the user's decision to abort.

SSN guarantees the safe retry property: Suppose SSN aborts transaction T because U violates its exclusion window, and that the user retries immediately with T'. An exclusion window violation requires $S \leftarrow T \overset{b}{\leftarrow} U$, where $\pi(U) < c(S) < c(T)$. Because U committed before T' began, T' will read the version U created without forming an anti-dependency.

The importance of safe retry is often overlooked, and many serializable schemes do *not* share this property, including 2PL (T' could deadlock with the winner of a previous deadlock) and OCC [17,29] that relies on read set validation (the overwriter could still be in progress, causing another failure).

Write-intensive workloads. We expect write-intensive workloads to perform better under RC+SSN than under SSI: A major source of transaction failures under SSI is temporal skew, where a transaction attempts to overwrite a version created after its snapshot. By allowing transactions to always access the latest version (except when forbidden by SSN), RC should lower the risk of encountering temporal skew in a short transaction.

4. IMPLEMENTATION

In this section, we describe how SSN can be implemented for a multi-version system, describing how each read, write and commit request is processed. We assume that there is storage associated with each version, and transaction, where we can put the information SSN requires. To overlay SSN on a locking-based single-version system, we will need to store information in lock entries as proxies for the versions, and keep some locks (in non-blocking modes) longer than the underlying CC would have done. We leave this as future work and focus on multi-version systems in this paper.

SSN can be implemented efficiently, requiring space and computation linear to a in-flight transaction's footprint, plus constant space for each version. SSN summarizes dependencies between transactions using various timestamps that correspond to commit times. For in-flight and recently-committed transactions, these timestamps can be stored in the transaction's context. For older transactions, the timestamps can be maintained in versions without a need to remember the committed transactions that influenced them. SSN supports early detection of exclusion window violations, aborting the transaction immediately if a too-new (too-old) potential predecessor (successor) dooms it to failure.

Table 1 summarizes the metadata which SSN tracks for each transaction T and version V. Version-related states persist for the life of the version, while transaction states are discarded after the transaction ends. Although SSN increases per-version space overhead, we note that many MVCC implementations already track some of these values.[1]

Read protocol. When reading versions, transaction T will record in `t.pstamp` the largest `v.cstamp` it has seen to reflect T's dependency on the version's creator. To record its read anti-dependency on the transaction that overwrote V (if any), T records the smallest `v.sstamp` in `t.sstamp`. The transaction then verifies the exclusion window, aborts if a violation is detected. If the transaction is aborted, the safe retry property means it can be retried immediately, minimizing both wasted work and latency. If the version has not yet been overwritten, it will be added to T's read set and checked for late-arriving overwrites during pre-commit.

Write protocol. When updating a version, T updates its predecessor timestamp `t.pstamp` with `v.pstamp` (rather than `v.cstamp`, because a write will never cause inbound read anti-dependencies, but it can trigger outbound read anti-dependencies). Then T records V in its write set for final validation at pre-commit, in case more reads came later. Additionally, we must remove V from T's read set, if present: updating $\pi(T)$ using the edge $T \overset{r:w}{\longleftarrow} T$ would violate T's exclusion window and abort unnecessarily.

Commit protocol. Before the actual commit happens, we conduct a *pre-commit* phase to check the exclusion window. A *post-commit* phase then propagates appropriate timestamps into affected versions. Pre-commit begins when T requests for a commit timestamp $c(T)$, which determines its global commit order, as depicted in Algorithm 1. After initializing $c(T)$, T is no longer allowed to perform reads or

Value	Meaning
`t.cstamp`	Transaction end time, $c(T)$
`t.status`	Status: in-flight, committed, or aborted
`t.pstamp`	Predecessor high-water mark, $\eta(T)$
`t.sstamp`	Successor low-water mark, $\pi(T)$
`t.reads`	Non-overwritten read set
`t.writes`	Write set
`v.cstamp`	Version creation stamp, $c(V)$
`v.pstamp`	Version access stamp, $\eta(V)$
`v.sstamp`	Version successor stamp, $pi(V)$
`v.prev`	Pointer to overwritten version

Table 1: Metadata required by SSN.

Algorithm 1 SSN commit protocol

```
1  def ssn_commit(t):
     t.cstamp = next_timestamp() # begin pre-commit

4    for v in t.writes: # finalize \eta(T)
       t.pstamp = max(t.pstamp, v.pstamp)

7    # finalize \pi(T)
     t.sstamp = min(t.sstamp, t.cstamp)
     for v in t.reads:
10     t.sstamp = min(t.sstamp, v.sstamp)
     ssn_check_exclusion(t)
     t.status = COMMITTED
13
     # update \eta(V)
     for v in t.reads:
16     v.pstamp = max(v.pstamp, t.cstamp)

     for v in t.writes:
19     v.prev.sstamp = t.sstamp # update \pi(V)
       # initialize new version
       v.cstamp = v.pstamp = t.cstamp
```

[1] For example, PostgreSQL also maintains the equivalent of `v.cstamp` and `v.prev`.

writes. It then computes $\pi(T)$, following the formula given in Section 3. The computation only considers $\pi(V)$ of reads that were overwritten before $c(T)$.

The transaction next computes $\eta(T)$ using a similar strategy, but must account for more dependency edge types. Recall that T can acquire predecessors in two ways: reading or overwriting a version causes a dependency on the transaction that created it; overwriting a version also causes a dependency on all readers of the overwritten version. The read and write protocols account for the former by checking $c(V)$, and pre-commit accounts for the latter using $\eta(V)$.

Once $\pi(T)$ and $\eta(T)$ are both available, a simple check for $\pi(T) \leq \eta(T)$ identifies exclusion window violations. Transactions having $\eta(T) < \pi(T)$ are allowed to commit. During post-commit, the transaction updates $c(V)$ for each version it created, $\pi(V)$ for each version it overwrote, and $\eta(V)$ for each non-overwritten version it read.

Phantom protection. An acyclic dependency graph implies serializability only in the absence of phantoms. Although SSN can be extended to detect phantoms by tracking coarse-grained dependencies in predicate-based selections, we leave that to future work. The current implementation extends the underlying CC scheme to prevent phantoms, using well-known techniques such as index versioning [29] and locking [10, 20, 21, 29].

5. EVALUATION

We implement SSN in our prototype OLTP system based on Silo [29], a representative memory-optimized database system that uses OCC. To support multi-versioning, we have modified Silo with global ordering and table-private indirection arrays [27], with each entry pointing to the head of the corresponding tuple's version chain. A preliminary description about our system can be found in [13]. In the same system, we implemented RC, SI, and SSI. An overview for each of them is given below.

1. **Read Committed (RC).** Reads return the newest committed version of a record and never block; writes add a new version that overwrites the latest one, blocking only if the latter is uncommitted. Allows dependency cycles but forbids isolation failures (dirty reads and lost writes).
2. **Snapshot Isolation (SI).** Each transaction reads from a consistent *snapshot*, consisting of the newest version of each item that predates a timestamp (typically, the transaction's start time). Writers must abort if they would overwrite a version created after their snapshot. Allows write skews, but forbids isolation failures and enforces write atomicity.
3. **Serializable Snapshot Isolation (SSI).** Like SI, but forbids the "Dangerous Structure": $T_1 \xleftarrow{r:w} T_2 \xleftarrow{r:w} T_3$ with T_3 committed first. No cycles are possible.

We apply SSN over both RC (RC+SSN) and SI (SI+SSN) to compare their performance. Our prototype OLTP system provides an efficient implementation of parallel pre-commit for SSI. The SSN implementation follows the same parallel pre-commit paradigm. We also show the performance numbers obtained when running the same workloads with original Silo (denoted as "OCC") for reference. We are interested in how SSN performs in general (commit throughput), SSN's performance for write-intensive transactions, accuracy (abort rate), and effectiveness of the safe retry property.

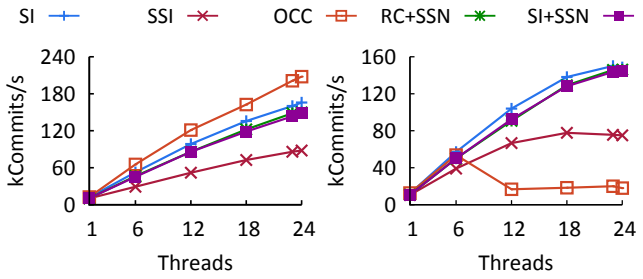


Figure 3: TPC-C Payment throughput when retrying is disabled (left) and enabled (right).

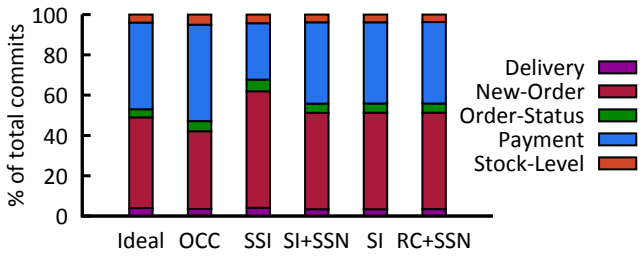
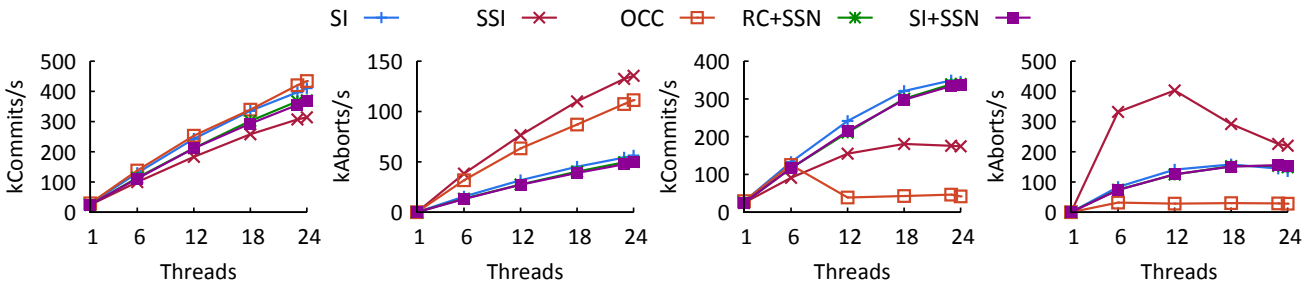


Figure 4: Throughput breakdown of the full TPC-C mix, when running at 24 threads and without retrying aborted transactions. The original breakdown in the TPC-C spec is labeled as "Ideal" for comparison.

We run experiments on a quad-socket server with four Intel Xeon E7-4807 CPUs (24 physical cores in total) and 64GB main memory. To evaluate SSN under a contentious scenario, we modify the TPC-C implementation so that each transaction uses a random warehouse. For each run, we fix the number of concurrent threads to the scale factor.

Write-intensive transactions. As we have mentioned in Section 3, compared to SSI, SSN+RC should be more friendly to preserving write-intensive transactions. We choose the Payment transaction in TPC-C to test this property. Figure 3 shows the throughput (commit and abort rates) of the Payment transaction when running the standard TPC-C transaction mix. On the left side of the figure we show how different CC schemes perform when the system does not retry aborted transactions (i.e., aborted transactions are dropped). Both SSN variants (RC+SSN and SI+SSN) perform almost 2x better than SSI, and OCC performs the best—which is expected as OCC is known to be friendly for write-intensive transactions.

However, as shown on the right side of Figure 3, the original Silo collapsed as core count increases when it needs to retry aborted transactions (until commit). It does not scale beyond one socket (6 physical threads in our system). Our profiling results show that Silo spent more than 60% of total CPU cycles on retrying index insertions (mostly for New-Order), minimizing the available cycles for Payment and other transactions. As mentioned in the beginning of this section, we use indirection arrays for multi-versioning in our prototype system. This design makes tuple insertion more efficient: Silo needs to first retry index insertion before finalizing the tuple write at commit time, putting tremendous pressure on the index, while our system only



(a) Commits/s. (b) Aborts/s. (c) Commits/s with retry. (d) Aborts/s with retry.

Figure 5: TPC-C commit/abort rates when transaction retry is disabled (a-b) and enabled (c-d).

needs to insert to the index after successfully appended an entry in the table's indirection array, amortizing most contention on the index. The relative performance of SSN and SSI remained similar to the case where we aborted transactions are dropped. Therefore, we conclude that Silo is not robust against retries, and SSN is significantly more friendly to write-intensive transactions than SSI.

Transaction breakdowns. To further understand how different types of transactions perform under SSN, the y-axis of Figure 4 presents the relative percentage of each transaction's commit in the TPC-C mix, for the different CC schemes in the x-axis, including the transaction mix specified by the TPC-C spec [28] for comparison. The experiment was conducted with 24 threads and aborted transactions are dropped. Consistent with our findings on the Payment transaction shown in Figure 3, compared to other schemes and the ideal case, SSI has a much smaller percentage of finished Payment transactions, indicating its bias against update-intensive transactions: SSI exaggerates the fact that SI could starve more updates compared to OCC schemes. Also, OCC is more friendly to write-intensive transactions, with around 136% and 39% more committed Payment transactions/s than SSI and SSN schemes, respectively. SSN's breakdowns in the figure are similar to SI's: under SI it does not starve updates like SSI does.

Commit and abort rates. We run the standard TPC-C mix to observe the commit and (especially) abort rates. As shown in Figure 5(a), almost all schemes scale well. In Figure 5(b) we show the corresponding abort rates: SSI and OCC both have abort rates more than 2x of SSN variants. Compared to SSI, SSN allows more serializable schedules. Though hard to see from the figures, RC+SSN performed slightly better than SI+SSN, as it allows certain serializable schedules that are not possible in SI.

When we enable the system to retry aborted transactions, the commit rates are similar to Payment's, with SSI being in the middle of SSN and OCC, as shown in Figure 5(c). The abort rates, however, differ drastically among different CC schemes. As shown in Figure 5(d), SSI's abort rate skyrocketed as we run more concurrent transactions, though it showed a declining trend beyond 12 threads. The abort rate of both SSN variants are comparable to that of SI. Note that OCC in this experiment has actually kept the lowest abort rate. However, as we explained earlier, the system spent most of its time contending on index insertion, getting little useful work done, as reflected by the extremely low commit rate shown in Figure 5(c).

6. RELATED WORK

Gray et al. [11] defined various isolation levels. The recent focus is on multi-version CC schemes, especially SI which is defined academically and proven non-serializable [3]. Definitions of isolation properties based on patterns of dependency edges are given by Adya [1].

Certification approaches to serializability are a form of optimistic CC [16]. The exactly accurate approach of SGT [6] was also extended to multi-version systems [12, 26]. A different approach tests for cycles before transactions start in a real-time database system [18]. SSI [5] is a certification that runs specifically along with SI. Various improved forms of SSI was also implemented [24]. Lomet et al. [19] choose a commit timestamp from an allowed interval, whereas we use the commit time as timestamp, and track excluded values. Hekaton [17] rejects all back-edges. Unlike SSN, these techniques cannot be combined with CC schemes that is not based on a snapshot for reading (e.g., RC).

A different class of proposals ensures serializable execution by doing static pre-analysis of the application mix [2, 7, 15]. Unlike SSN, they are not suitable with ad-hoc queries.

7. CONCLUSIONS

This paper presented and evaluated the serial safety net (SSN), a cheap certifier that can overlay a variety of concurrency control schemes and make them serializable. We prove the correctness of SSN, demonstrate its effectiveness in a real main-memory OLTP system on modern hardware with substantial core count and large main memory. We find that SSN is superior to prior state-of-the-art, in being more accurate (fewer aborts), more general (not requiring SI), more robust against retries and more friendly to write-intensive transactions.

Acknowledgments

The authors gratefully acknowledge Goetz Graefe and Harumi Kuno for their invaluable suggestions and input they provided during the development of SSN, as well as Kangnyeon Kim for his help building the prototype.

8. REFERENCES

[1] A. Adya. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions.* PhD thesis, MIT, 1999.

[2] M. Alomari, A. Fekete, and U. Röhm. Performance of program modification techniques that ensure

serializable executions with snapshot isolation DBMS. *Inf. Syst.*, 40:84–101, 2014.

[3] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil. A critique of ANSI SQL isolation levels. *SIGMOD*, pages 1–10, 1995.

[4] P. A. Bernstein, D. W. Shipman, and W. S. Wong. Formal aspects of serializability in database concurrency control. *IEEE Trans. Software Eng.*, 5(3):203–216, 1979.

[5] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable isolation for snapshot databases. *ACM TODS*, 34(4):20:1–20:42, Dec. 2009.

[6] M. Casanova and P. Bernstein. General purpose schedulers for database systems. *Acta Inf.*, 14:195–220, 1980.

[7] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM TODS.*, 30(2):492–528, June 2005.

[8] A. Fekete, E. O'Neil, and P. O'Neil. A read-only transaction anomaly under snapshot isolation. *SIGMOD Rec.*, 33(3), 2004.

[9] G. Graefe. A survey of B-tree locking techniques. *ACM TODS*, 35(3):16:1–16:26, July 2010.

[10] J. Gray. *Notes on data base operating systems. Advanced course: operating systems.* Springer, 1978.

[11] J. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Granularity of locks and degrees of consistency in a shared data base. *IFIP Working Conf. on Modelling in DBMS*, 1976.

[12] T. Hadzilacos. Serialization graph algorithms for multiversion concurrency control. *PODS*, pages 135–141, 1988.

[13] R. Johnson, K. Kim, T. Wang, and I. Pandis. Robust concurrency control in main-memory DBMS: what main memory giveth, the application taketh away. *IMDM*, pages 57–59, 2014.

[14] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: a scalable storage manager for the multicore era. In *EDBT*, 2009.

[15] S. Jorwekar, A. Fekete, K. Ramamritham, and S. Sudarshan. Automating the detection of snapshot isolation anomalies. *PVLDB*, pages 1263–1274, 2007.

[16] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM TODS*, 6(2):213–226, 1981.

[17] P.-A. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. *PVLDB*, 2011.

[18] V. C. Lee and K.-W. Lam. Conflict free transaction scheduling using serialization graph for real-time databases. *Journal of Systems and Software*, 55(1):57–65, 2000.

[19] D. Lomet, A. Fekete, R. Wang, and P. Ward. Multi-version concurrency via timestamp range conflict management. *ICDE*, pages 714–725, 2012.

[20] D. B. Lomet. Key range locking strategies for improved concurrency. *PVLDB*, pages 655–664, 1993.

[21] C. Mohan. ARIES/KVL: A key-value locking method for concurrency control of multiaction transactions operating on B-Tree indexes. *PVLDB*, pages 392–405, 1990.

[22] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *PVLDB*, 3(1), 2010.

[23] C. H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979.

[24] D. R. K. Ports and K. Grittner. Serializable snapshot isolation in PostgreSQL. *PVLDB*, 5(12):1850–1861, Aug. 2012.

[25] PostgreSQL Global Development Group. Chapter 13. Concurrency Control. *PostgreSQL 9.4.1 Documentation*, 2015.

[26] S. Revilak, P. E. O'Neil, and E. J. O'Neil. Precisely serializable snapshot isolation (PSSI). *ICDE*, 2011.

[27] M. Sadoghi, K. A. Ross, M. Canim, and B. Bhattacharjee. Making updates disk-I/O friendly using SSDs. *PVLDB*, 6(11):997–1008, Aug. 2013.

[28] Transaction Processing Performance Council. TPC benchmark C – Standard Specification.

[29] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. *SOSP*, pages 18–32, 2013.

APPENDIX

A. CORRECTNESS

We provide the formal proof for SSN in this section. Based on the database model we set up in Section 2, we first set the stage by giving the key result of serialization theory:

THEOREM 2. *Let an execution with schedule h have a serialization graph $G(h)$ with no cycles. Then the execution is serializable* [2].

As mentioned in previous sections, SSN requires the underlying CC scheme forbid lost writes and dirty reads:

DEFINITION 3. *Let a* certifiable *scheduler be any concurrency control scheme that forbids lost writes and dirty reads (other than a transaction reading its own writes).*

Definition 3 effectively allows any CC scheme at least as strong as read committed. In particular, the CC scheme is free to return any committed version from a read (not necessarily in a repeatable fashion), and can delay accesses arbitrarily.

Given a non-serializable schedule h produced by a certifiable scheduler, we first identify the "dangerous" edges in its dependency graph $G(h)$ that SSN targets at. We then prove that these edges exist in any dependency cycle that arises under a certifiable scheduler.

We argue the correctness of SSN as follows.

THEOREM 4. *Let h be any non-serializable history produced by a certifiable scheduler. Then the dependency graph $G(h)$ contains at least one exclusion window violation.*

PROOF. By Theorem 2 and the hypothesis that h is non-serializable, $G(h)$ must contain a cycle involving $n \geq 2$ transactions.[3] Name the transactions in that cycle, so that T_n

[2] This has many formulations such as [4] and [23], the presentation with this form of dependency definition is in [1].

[3] We ignore self loops, since our model excludes them. In reality transactions will be allowed to read their own writes.

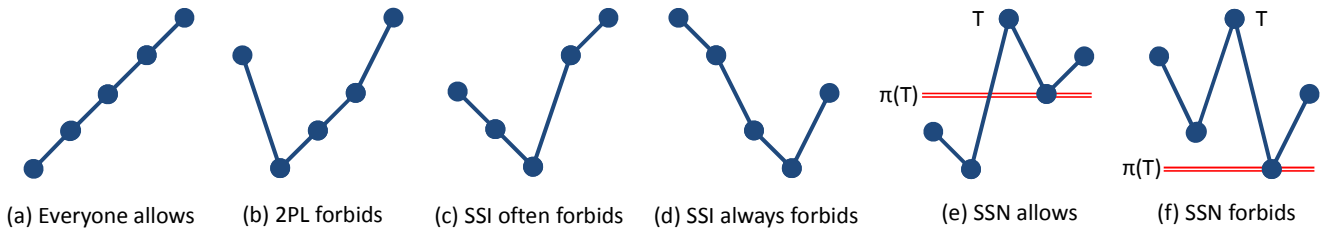


Figure 6: SSN allows all schedules that do not have "peaks," and also "peaks" where no predecessor of T violates the exclusion window. Other schemes tend to reject the "valleys" that arise frequently under MVCC.

committed first: $T_n \leftarrow T_1 \leftarrow T_2 \leftarrow \ldots \leftarrow T_{n-1} \leftarrow T_n$. Because T_n committed first in the cycle, its predecessor—which is also a successor—must be reached by a back edge; choose the lowest value of k such that $T_k \overset{b*}{\leftarrow} T_n$ holds. Then $\pi(T_k) \leq c(T_n)$. Further, the predecessor of T_k (T_{k-1}, or T_n if $k = 1$) must be reached by a forward edge. Combining the two facts shows an exclusion window violation: $\pi(T_k) \leq c(T_n) \leq c(T_{k-1}) < c(T_k)$. Since we have shown that T_k always exists and always has a predecessor that violates its exclusion window, we conclude that $G(h)$ always contains an exclusion window violation. □

Definition 5. *A certifiable scheduler is said to* apply SSN certification *if it aborts any transaction T that, by committing, would introduce an exclusion window violation into the dependency graph. That is, SSN forces T to abort if there exists a potential edge $U \leftarrow T$ where $\pi(T) \leq c(U) \leq c(T)$.*

Theorem 6. *Consider a certifiable scheduler that applies SSN certification. Then all executions produced by the scheduler are serializable.*

Proof. By contradiction: If there is any execution of the scheduler that is non-serializable, Theorem 4 shows that there is an edge in the dependency graph that violates the exclusion window; however the certification check in the scheduler does not allow any such edge to be introduced. □

B. SAFE RETRY

We formally prove SSN's safe retry property here: Suppose SSN aborts transaction T because U violates its exclusion window, and that the user retries immediately with T'. Then U cannot cause T' to abort (though newly arrived transactions could).

Theorem 7. *SSN provides the "safe retry" property, assuming the underlying CC scheme does not allow T to see versions that were overwritten before T began.*

Proof. An exclusion window violation requires $S \leftarrow T \overset{b}{\leftarrow} U$, where $\pi(U) < c(S) < c(T)$. Because U committed before T' began, T' will read the version U created and no anti-dependency will be created. □

C. DISCUSSION

We now compare SSN with other cycle prevention schemes, and reason about their relative merits. Figure 6 highlights several "shapes" that transaction dependencies can take when plotted in serial-temporal form. Of all the serializable schedules shown, SSN rejects only the last. In contrast, 2PL admits only the first (all others contain forbidden back edges). SSI always admits cases (a) and (b), always rejects (d), and often rejects (c) and (f).[4] Case (e) cannot even arise under SI, let alone SSI. Thus, the improved cycle test in SSN allows it to tolerate a more diverse set of transaction profiles than existing schemes, including schedules forbidden by SI.

[4]SSI allows (c) if the leftmost transaction is read-only and sufficiently old, but rejects (f) if a (harmless) forward anti-dependency edge joins T with its predecessor.