Exploiting Parallelism of Distributed Nested Transactions

Duane Niles Roberto Palmieri Binoy Ravindran
Virginia Tech
{duane9;robertop;binoy}@vt.edu

Abstract

We present SPCN, a framework that further extends the benefits of having distributed partially rollbackable (closed-nested) transactions by exploiting their parallel activation. SPCN provides support for executing each closed-nested transaction in parallel with others belonging to the same parent transaction. Their commit sequence is equivalent to the serial commit execution, but parallelism is leveraged to improve performance by reducing the amount of serial network communication. As we show in our evaluation study using 20 nodes on Amazon EC2 and three well-known benchmarks, SPCN provides performance improvement over the original closed nesting, gaining more than 2× in throughput.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; H.2.4 [*Systems*]: Transaction Processing

Keywords Distributed Transactions, Nesting, Parallelism

1. Introduction

A *transaction* is an abstraction in which a set of operations are grouped together as one individual *atomic* action, which executes all-or-nothing. Transactions typically keep all of the changes they have made hidden until they finish, or *commit*. A transaction commits if no conflicts have been detected between other transactions and itself. Upon detection of a conflict, one of the transactions must *abort* and prevent any of its changes from becoming visible. The aborting transaction can either restart from the beginning (*rollback*) and re-execute in its entirety, or, in the case that some of the operations executed are still valid, it can restart from an intermediate point (*partial rollback*).

A lightweight and less intrusive technique that allows for partial rollback is *nesting*, embedding transactions inside of one another. Nested transactions should be compositional, meaning that nested transactions do not perform operations that would break the parent transaction's operational correctness. Traditional parallel code (using explicit synchronization) is not composable, for two functions may conflict over the same synchronized data and then must be re-implemented in order to be used together. *Flat nesting* is the simplest form of nesting, where every inner transaction (*child transaction*) is executed as part of the top-level transaction (*root transaction*) and does not allow for intermediate executions. Any conflict detected aborts the entire transaction and all of the children, corresponding to the case of rollback. Although this method of nesting is the easiest to implement, other methods can improve transaction response time and throughput, but the gains must exceed any overheads introduced by allowing for partial rollbacks.

Closed nesting [11] treats each *parent transaction* as a container for its children. While sub-transactions are executing, they can abort independently from their parent, thereby potentially reducing the scope of rollbacks. When child transactions are finished and validated, they *speculatively commit*, merging their state into the state of their parent. The effects of any children are not seen until the entire root transaction (which has no parent) commits. If a conflict is detected after a child commits, its parent will abort in its entirety, clearing any actions of committed children. Further, a parent transaction can commit only after all the enclosed nested transactions have successfully committed.

Yet, even though closed nesting makes improvements with partial rollback [7, 12], it only allows one sub-transaction to be active at a given time, thus lowering internal concurrency. *Parallel nesting* [1, 2, 8], on the other hand, allows child transactions to execute concurrently with one another, ideally increasing the throughput of the whole transaction. However, parallel nesting raises some issues, with the main problem being that, in addition to already-present conflicts with external transactions, the system now allows conflicts internal to a transaction to occur between its children. Taking this further, because transactions can now have more pieces working in parallel, there are even bigger chances for conflicts between top-level transactions.

To give an example, let us consider the scenario where the parent transaction, T_P, has three closed-nested children: T_{N1}, T_{N2}, T_{N3}. Even though they are part of the same working transaction, T_{N1}, T_{N2}, and T_{N3} can either interact on the same resources (shared or private objects) or access all different objects such that their executions can be considered fully "independent." Here, we assume that T_{N2} conflicts with T_{N1} over an object, thus it must perform the read after T_{N1} has written. On the other hand, if T_{N3} is independent, it can finish early without conflict. Running these sub-transactions in

parallel allows the overlapping of their execution, shortening the parent transaction's critical path. Even with conflicts to resolve, the execution is ideally better than, or in the worst case equivalent to, the serial execution.

When applying the idea of parallel nesting to centralized systems, the above notions do not hold as well. Every operation inside of a transaction and its children is on the critical path, thus any overhead added to a transaction processing algorithm will delay the entire execution. On the other hand, in distributed systems, the critical path is mainly the network communication required for requesting data. Thus, adding an overhead for internal processing of transactions does not extensively add to the minimum time required to run the transaction. If transaction's children execute in parallel, then their network communication is overlapped and their processing and validation is all internal to the node, as they must validate against one another.

Parallelizing transactions within distributed systems is not well-explored, but because distributed systems hold less constraints than centralized systems, it is an intriguing path to pursue. In this paper we present *SPCN*, a speculative approach for executing distributed closed-nested transactions in parallel, capturing conflicts among them at run-time. The problem of activating multiple parts of the same transaction in parallel has already been studied in the past on centralized settings [1, 2, 8]. Unfortunately, for such systems it is not clear how to automate what should be in parallel without asking more of the programmer. Making even the slightest generalization about transactions in a centralized system could result in a decrease in performance if conflicting operations are placed in parallel. Doing so would result in more transactional aborts and, as all of the operations are on the critical path, would slow down the system. For a centralized system to make efficient usage of parallel nesting, the programmer would have to explicitly separate every piece of a transaction into independent sections, which would require extensive work in larger applications.

Instead, we rely on closed nesting for the benefit of partial rollback and to enforce an order of visibility on sub-transactions without requiring the programmer to make them entirely independent. If a programmer has created transactions using closed nesting, the general execution runs through the nested transactions in the order in which they are defined. To parallelize such nested transactions, enforcing an order of operation is still necessary if the contents of the transactions are unknown and not independent, as some constraint must be placed in order to process conflicts between the children. However, adding synchronization operations to enforce an order in centralized systems would again add to the critical path of the transactions, creating overheads that outweigh the possible benefits. On distributed systems, these operations, if properly organized, would not add an extensive overhead as the internal processing of transactions is not on the critical path, thus allowing more innovations.

SPCN provides two different protocols when it comes to the completion of sub-transactions, which are all assigned a *transaction order*, TO. The *Strict* protocol (Section 5.1) enforces sub-transaction order directly by keeping *future* children, those with a higher TO, from completing (although they are still activated in parallel) until all of the children prior to them have completed. In SPCN, when a parallel child has completed, we say that a transaction has *sub-committed*. As stated earlier, a nested transaction does not make its actions visible to anything outside of its parent transaction, thus a sub-commit means the same for parallel nested transactions.

The *Relaxed* protocol (Section 5.2), on the other hand, allows future children to sub-commit before all of their previous siblings have finished (assuming no conflicts have been found up to that point). However, the protocol allows them to be aborted afterwards if a prior sibling transaction detects a conflict at a later time. This protocol, while allowing a sub-transaction to be aborted after sub-committing, tries to treat the concurrency optimistically. Note that the Strict protocol still requires the conflicting sub-transaction to be aborted, but the Relaxed protocol creates overhead by needing to undo all of the aborted transaction's updates as well as managing additional data to synchronize child transactions and detect internal conflicts.

We chose to build SPCN upon a Distributed TM (DTM) system, *Hyflow2* [15], which is implemented in Scala. We evaluated SPCN using an experimental study with the benchmarks Bank, TPC-C [6], and YCSB [5], measuring the performance against closed nesting on up to 20 nodes in Amazon EC2. Our results reveal that SPCN consistently outperforms the sequential version of closed nesting by as much as 2.85× using Bank, 3.78× with TPC-C, and 2.9× when applied to YCSB.

The code is publicly available as an open source project located at https://bitbucket.org/duaneVT/reflow.

2. Related Work

Nested transactions (using closed nesting) originated in the database community and were thoroughly described by Moss in [11]. One of the early works introducing nesting to Transactional Memory was done by Moss and Hosking in [12]. They describe the semantics of transactional operations in terms of system states, which are tuples that group together a transaction ID, a memory location, a read/write flag, and the value read or written.

Closed nesting has also been shown as effective in distributed deployments by [7]. In this work authors execute sub-transactions sequentially without parallel activation, thus the cost of the distribution is always on the transaction's critical path. On the contrary, SPCN still exploits the effectiveness of closed nesting but also increases the system's concurrency by overlapping sub-transactions' executions.

Even though Transactional Memory (TM) promises to make concurrent programming easier to the wider programming community, nested transactions are generally not allowed to run in parallel. This is an important obstacle to the central goal of TM. Due to this issue, parallel nesting has been studied in centralized settings [1, 2, 8].

Agrawal et. al. [1] propose XCilk, a runtime-system design supporting transactions that themselves can contain nested parallelism and nested transactions. XCilk shows the first theoretical performance bound on a TM system that supports transactions with nested parallelism. Baek et. al. [2] present NesTM supporting closed-nested parallel transactions. NesTM uses eager version management and word-granularity conflict detections targeting the state and runtime overheads of nested parallel transactions.

Barreto et. al [3] utilize thread-level speculation and TM to execute operations out-of-order and detect inconsistencies at runtime. It requires the programmer to break an application down into coarse-grained transactions. The model assumes no nesting. Transactions are separate parts of sequential code and broken down further by using techniques such as compile-time code inspection. The system uses redo-logs and requires any write to immediately lock an object, thus preventing any other parallel transactions from accessing it. A contention manager is used if separate applications attempt to write the same object, and transactions will either signal the other to abort if it is in the future or will wait until the transaction completes if it is in the past.

Diegues et. al [8] build upon JVSTM [9] by allowing transactions to directly write to Version Boxes (VBox) which hold permanent versions of objects written by top-level transactions. The VBox is extended to allow for tentative versions written by currently-active transactions. When sub-transactions complete, all of the relative VBox entries swap ownership to their parent, and once the top-level transaction commits, the latest tentative versions become permanent. The system expects the application to already be broken into separate pieces that have no inherent ordering, thus they can commit before other sub-transactions.

Differently from the above proposals, SPCN activates transactions in parallel but it enforces the serialization order as their order of creation, since it does not assume sub-transaction independence. In addition, if a conflict arises, partial rollback can be leveraged for restarting just the executing sub-transaction rather than the whole transaction.

3. System and Execution Model

System Model. We consider a distributed system which consists of a set of nodes. The nodes communicate with each other by message-passing across a communications network. Nodes do not have globally-shared memory, nor do they have a consistent global clock or instance of time.

Distributed systems may utilize *replication*, meaning that application data will be placed on more than one node as a means for ensuring fault tolerance. However, SPCN is an orthogonal mechanism to replication. It can be deployed in systems where replication is employed, as well as where each node keeps just a partition of the shared resources. In this paper we focus on such latter systems, allowing us to better showcase the behavior of SPCN without additional overheads due to replication.

Transaction Execution Model. A set of distributed transactions $DT = \{T_1, T_2, \cdots\}$ is assumed. Transactions share a set of objects $Obj = \{O_1, O_2, \ldots\}$, which are assumed to be distributed (i.e., partitioned) on the nodes of the system. Transactions are modeled as a set of *begin*, *read*, *write*, *commit* and *abort* operations on Obj, and they define a total order in which these operations are executed. Transactions that do not execute a write operation are called read-only transactions; otherwise, they are called update (or equivalently write) transactions.

The transaction processing complies with the control-flow model [14], where objects are immobile and transactional operations are invoked on the owners of the accessed objects. In this model, the nodes responsible for maintaining an object are fixed since the creation of the object and until its deletion. When a transaction performs an operation on an object stored on a remote node, the operation is invoked as a remote procedure call on that node.

The object lookup mechanism relies on a directory distributed across all nodes. A directory performs lookup by using a consistent hashing function on an object's ID. The hash result represents the node in the system that stores the portion of the directory where the ownership of the requested ID is defined. Control-flow-based protocols do not necessarily need the deployment of a directory as just described. For instance, the consistent hashing function can directly return the owner of the object according to its ID. We decide to deploy the directory service, however, because through it we can decide the ownership of objects without letting the consistent hashing function decide. By querying the owner, the object is returned via a message to the requester, or a failure message will be sent if another node has locked the object.

4. Handling Contention of Root Transactions

When a transaction T requests an object, it receives a message from the owner containing either the object itself or a signal if it is locked. In the latter case, another transaction is updating the object and a conflict has been detected. In this case, T will abort and rollback (or, according to the closed nesting model, partially rollback if it is a child transaction). If the request is successful, then T retrieves the object along with the object's version, or *timestamp*. Each object in the system has a timestamp that is maintained by its owner. Each time a transaction commits a write (or update) to the object, the timestamp for that object is incremented. This versioning is used to detect conflicts during the validation phase of

a transaction, thus ensuring that the history observed by the transaction is consistent.

All transactions buffer their reads and writes into a private, per-transaction, *read-set* and *write-set* during execution. These sets are hash tables organized by using object IDs as the keys. Using a classical lazy approach, write operations are performed locally without interacting with any object owner (if the object has not been already read). The writing transaction, T, stores a pair of $(object, value)$ into its write-set. As said before, at the end of their execution, transactions must perform a validation. The validation is done after locking all the objects written, so that if the validation succeeds, then the updates can be safely applied. To do so, T groups the objects in their write-set by owner and sends messages to the owners to lock the objects. Locking objects on a node is only used during this commit-time as a means to synchronize updates to the objects (i.e., mutual exclusion); no lock is used during the execution of the transactions.

Similarly to when an object is requested, the owners return messages either stating a successful lock or a conflict where an object is already locked. If a conflict is detected, then the requesting transaction aborts and rolls back as before. The committing transaction also sends messages to read all of the latest versions of the objects in its read-set. If any of the objects are locked or have been updated since the transaction read them, meaning that their current version is larger than the version recorded by the transaction at the time of its execution, then the transaction aborts as well.

After locking and validating, the transaction increments the timestamp of all objects in its write-set, if it has any, then sends commit messages to the owners of the objects. In the control-flow model, all of the commit messages contain the updated versions of the written objects and the owners will replace the objects in their storage. Afterwards, the owners unlock the objects, thereby allowing other transactions to modify them. At this point, the transaction has committed.

Adding in the concept of nesting, transactions can also have children, allowing for partial rollback. The execution of transactions follows as before, but when sub-transactions complete, they simply merge their read-set and write-set with their parent transaction, the transaction directly above them in the hierarchy. In terms of closed nesting, no extra step is necessary, thus no temporary validation is performed nor are any objects queried for or locked by sub-transactions. Only when all sub-transactions in the activation tree are completed will the top-level, or root, transaction begin the full commit, following the same process of locking/validation as above. None of the actions of sub-transactions are visible until the root transaction commits.

What we have described so far implements the well-known Two-Phase Commit (2PC) [4] atomic commitment algorithm, which ensures atomicity on the commit of a transaction. Although 2PC is well-known to be blocking upon coordinator failure, the issue of how to ensure high availability of the transaction coordinator state is well understood, and a range of orthogonal solutions have been proposed in literature (e.g., Paxos Commit [10]).

The shown concurrency control is sufficient to guarantee a serializable execution [4]. We can prove that by relying on other solutions that use the same approach. SCORe [13] adopts the same combination of validating read objects after having locked written objects using 2PC. SCORe is more complex because it also provides non-blocking read-only transactions. Our proposal allows a smaller subset of the transactional schedules that SCORe allows, thus we can rely on it to claim our correctness criterion.

5. SPCN: Speculative Parallel Closed Nesting

In this section we detail our two versions of SPCN, one named *Strict* and the other *Relaxed*. The main distinguishing point between the two is the time at which sub-transactions are allowed to commit. In the former, a sub-transaction cannot commit until the previously-ordered one has already committed. In the latter, we let the sub-transactions commit in any order, but they can be subsequently aborted due to (internal) conflicts with other sub-transactions.

5.1 SPCN: Strict Protocol

SPCN begins by initiating a specific number of top-level transactions on a given node, each of which can spawn child transactions and begin executing them fully in parallel. The child transactions are assigned an incremental *transaction order* (TO) to enforce their serialization order. Sub-transactions are not allowed to sub-commit until all the previous siblings have sub-committed, in order to prevent changes from possibly appearing out-of-order. During execution, sub-transactions can internally validate themselves by checking for conflicts with previous siblings (i.e., sub-transactions with lower TO values).

Each transaction (and child transaction) manages its own read-set and write-set to track its own data and changes. When a child sub-commits, as is normally done in closed nesting, the child will merge its read-set and write-set into the parent transaction. Until the merge, however, none of a child transaction's changes are visible to any of the other siblings; otherwise, transaction isolation would be broken and aborts could potentially cascade.

Once a child transaction sub-commits, all of the other children can visibly read its data by observing the parent's data. Note that because of the order assigned to the sub-transactions, children with lower TO cannot observe the data of future siblings. For example, if transaction T_2 writes to any object, its previous sibling T_1 cannot read any of those changes ever. The Strict protocol accounts for this, as mentioned before, by simply stalling T_2 from sub-committing until T_1 has sub-committed. For this reason we can consider this version of SPCN as more pessimistic than its Relaxed counterpart. However, optimism usually comes with addi-

tional overhead, which is often higher than the achievable gain. As we will see in the evaluation, this is mostly the case with the Relaxed protocol, even though there are scenarios where Relaxed outperforms Strict.

Due to the sub-commit order enforced by Strict, the only conflict between siblings that needs to be accounted for is Write-After-Read, where a transaction reads an object, then a sibling with a lower TO writes the object at a later point, invalidating the read. This conflict is detected very easily by finding the intersection of the current transaction's read-set with the parent transaction's write-set, which contains the written objects of all previous siblings. Note that if two sub-transactions write to the same object without reading it, that is perfectly valid as the later transaction cannot publish the write until all of its previous siblings have finished. In most applications, however, it is likely that a transaction has already read an object if it goes on to write to it later.

Because sub-transactions are only allowed to sub-commit in their order of execution, each root transaction only needs to keep a single version of each object present at any point of time. When a sub-transaction writes to an object, say X, that version is overwritten if a future sibling sub-commits another write to X. An example is shown in Figure 1 where three sub-transactions write to the same object. In Relaxed, there would be three separate versions stored. However, in the Strict protocol, T_{N2} would not be able to publish its writes until T_{N1} had done so, and T_{N4} would similarly wait upon T_{N3}, meaning T_{N2} had already completed. Thus, the object X would be overwritten each time.

Algorithm 1 shows the initialization and commit of sub-transactions in Strict. In the first function, the parent transaction uses the current transaction order TO and gets the Future of the previous sub-transaction (line 4) or assigns null if there was no previous sub-transaction (line 6). Futures are an API in the Scala programming language as a means for executing a task and monitoring its completion. The parent transaction then creates the new sub-transaction by giving it a reference to itself as well as its previous sibling, and stores the new Future into the array by order (lines 9-10). Lastly, the parent executes the sub-transaction using a dispatcher *exec* and increments TO (lines 11-13).

The second function shows the sub-commit of child transactions. At this point, the sub-transaction waits on the immediately-prior sibling's Future to complete (line 16). Once the sibling is finished, then the current sub-transaction validates itself. Recall that the only conflict is Write-After-Read, thus the sub-transaction gets the intersection of its read-set along with the parent's write-set, which now contains the writes of all previous siblings (line 20). If there is an intersection, it checks to see if any of the read versions are different from the latest versions, and if so, it marks a sibling conflict as having occurred and aborts (lines 21-26). Upon successful validation, the sub-transaction simply merges its read-set and write-set with the parent (lines 29-30). Note

Algorithm 1: Strict Protocol

```
1  Procedure ChildBegin
   Input: TO, txnFutures, exec
   Output: None
2  ▷ Get the future of the previous sibling
3  if TO > 0 then
4      prevSibling = txnFutures.get(TO − 1);
5  else
6      prevSibling = null;
7  end if
8  ▷ Create a new sub-transaction and run it
9  subTxn = createSubTxn(this, TO, prevSibling);
10 txnFutures.set(TO, subTxn);
11 subTxn.run(exec);
12 ▷ Increment transaction order
13 TO++;

14 Procedure ChildCommit
   Input: parent, prevSibling, siblingConflict, RS, WS
   Output: None
15 ▷ Wait upon the previous sibling to commit
16 waitUntil(prevSibling);
17 ▷ Check if a previous abort was a sibling conflict
18 if siblingConflict == false then
19     ▷ If not, validate the read-set
20     overlap = intersection(RS, parent.WS);
21     for all objects obj in overlap do
22         if RS(obj) != parent.WS(obj) then
23             siblingConflict = true;
24             abortTxn();
25         end if
26     end for
27 end if
28 ▷ Upon success, merge with the parent
29 mergeRS(parent);
30 mergeWS(parent);
```

that the *siblingConflict* variable is a fast-path used to skip this validation (line 18). If a sub-transaction aborts because of a previous sibling, then it does not have to backoff or validate again when it comes back to commit. The reason is that the previous siblings are all done, so the sub-transaction can immediately re-read the data that conflicted.

Now, there is a general problem that arises from Strict's simple wait-and-commit methodology. If two sibling transactions are unrelated and do not have any potential conflicts, then stalling the later transaction until the previous sibling sub-commits is not fully effective; but as previously mentioned, independence of the nested transactions is not assumed, as that would have required the programmer to explicitly separate code into completely disjoint sections.

This consideration motivated us to design the Relaxed version of SPCN (Section 5.2). Supporting out-of-order sub-commits requires maintaining multiple versions of the same object, making reads visible, and monitoring a sub-committed transaction until all of its previous siblings are also sub-committed. These components are necessary to ensure the correctness of the transactions and they introduce

some overhead, none of which exists in the Strict version; thus, the system's workload should satisfy certain conditions so that the additional overhead's impact is reduced.

5.2 SPCN: Relaxed Protocol

The Relaxed protocol is an extension of Strict that requires more complex structures but also allows for more potential parallelism. Sub-transactions with higher TOs are allowed to sub-commit speculatively even if previous siblings are still active, so long as none of the previous siblings nor the sub-committing transaction found any conflicts up to that point. We name this sub-commit speculative because the serialization order is still not finalized until all previous siblings sub-commit, although the sub-committing transaction, T_C, can publish its changes to the parent before this occurs. Note that, because the sub-transactions still have an order enforced by TO, earlier siblings could still be active and discover a conflict with T_C. In terms of the conflict itself, the only problem is still the Write-After-Read conflict: if T_C read from a variable X and sub-committed before a previous sibling wrote to X.

In order to detect such conflicts with out-of-order sub-commits, multiple versions of objects must be stored and each transaction must store which version of each object it has read. When it comes to keeping track of writes, each object has its own structure, here named as "version tree" (*verTree*), that sorts writes by the order (i.e., TO) associated with the corresponding sub-transaction. The most sensible tree would be an AVL Tree, used for efficient insertion, deletion, and searches, all of which cost $O(log\ n)$ on average where n is the number of levels in the tree.

We also use a hash map (*readHash*) to keep track of transactions reading objects, making the system use *visible reads*, although the reads are only visible after the child has sub-committed. Each object has a hash map where the keys are the TOs representing a version of the object, and the values in each bucket are more TOs representing which transactions read the corresponding version of the object. While adding this structure uses more space per object, it results in faster conflict detection. The tree is better for reading and writing versions based on the TO while the hash is better for knowing which transaction has read a version, as iteratively traversing the trees would take more time.

Child Commit. Algorithm 2 shows the commit for sub-transactions and the parent thread processing the committing transactions. Starting with the child transactions, they first check if the root transaction is invalid and abort if so (lines 3-5). Normally, in a single-threaded transaction, a conflict with already-committed data would cause the whole transaction to abort and restart. However, in this parallel implementation, the children are all different threads and thus the root must set an invalid flag for the children to query. Then, the child waits until the *syncLock* variable is available (line 7). The variable is used as a barrier if multiple transactions are to reach this point at the same time.

Algorithm 2: Relaxed Protocol

```
1  Procedure ChildCommit
   Input: RS, WS, TO
   Output: None
2  ▷ Check the root's validity before committing
3  if root is invalid then
4      abortTxn();
5  end if
6  ▷ Synchronize with siblings for committing
7  waitUntil(syncLock);
8  ▷ Validate the read-set
9  for all objects obj in RS do
10     check = readPrevious(obj, TO);
11     if check.TO != obj.TO or check.data != obj.data then
12         ▷ The object is out-of-date or has incorrect data
13         abortTxn();
14     end if
15 end for
16 ▷ Publish the visible reads
17 for all objects obj in RS do
18     markAsRead(readHash, obj, TO);
19 end for
20 ▷ Publish the write-set
21 for all objects obj in WS do
22     ▷ Find out which siblings will be invalidated
23     invalidSiblings = getReaders(obj, previousVersion(TO));
24     for txn in invalidSiblings do
25         clearWrites(txn);
26         invalidSiblings += getReaders(txn.WS, txn.TO);
27     end for
28     ▷ Update the object
29     writeTree(obj, TO);
30 end for
31 ▷ Signal the parent and unlock
32 signal(childCommitted);
33 syncLock.next();

34 Procedure Commit
   Input: invalidSiblings
   Output: None
35 while numCommitted < numChildren do
36     ▷ Wait for a commit and restart invalid children
37     waitUntil(childCommitted);
38     for txn in invalidSiblings do
39         numCommitted−−;
40         restartTxn(txn);
41     end for
42     numCommitted++;
43 end while
44 ▷ Update the read-set with distinct objects from the children
45 updateReadSet(readHash);
46 ▷ Update the write-set with the latest object versions
47 updateWriteSet(verTree);
```

Once the transaction is able to commit, it must validate its read-set. For each object in the set, it checks the immediately-prior version available in the *verTree* for the object (line 10). If the version of the object is not equivalent to what the transaction has read, or if the version is the same but the data has changed, then the transaction must

abort as it is no longer correct (lines 11-14). If no conflicts are detected, the transaction makes its reads visible to other transactions in the *readHash* (lines 17-19), then moves on to publish its write-set. Note that the validation of the read-set and the publishing of the reads is separate. The reason is that if transactions published reads as they validated, then they would have to undo all of the publishes if a later conflict was detected, thus that would add even more overhead.

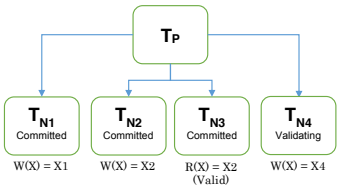


Figure 1. Example of SPCN.

The transaction must observe the *readHash* entries for each object it is writing and look at siblings which read the immediately-prior version of the object. If any of those readers have a larger TO than the committing transaction, the transaction removes the *readHash* entry for that invalid sibling and adds it to a list of invalid transactions (line 23). For instance, in Figure 1, transaction T_{N4} is committing and writing object X which has two versions, X_1 and X_2 corresponding to T_{N1} and T_{N2}. Transaction T_{N4} must only check the readers of X_2 and flag them as invalid if they have TO greater than 4. If T_{N4} sees that T_{N3} has read X_2, it is perfectly valid as T_{N3} is meant to occur before T_{N4}.

For each sibling marked as invalid, the transaction's writes are removed as they are no longer valid (line 25). Further, other transactions are marked as invalid if they read any of the objects from the invalid transaction's write-set (line 26), as their information is no longer valid as well. The transaction then publishes the object (line 29). Once it has completed all writes, the transaction signals that it has committed and then allows the next sibling if it is queued, or simply frees *syncLock* for another sibling's commit.

While the conflict detection may seem rather expensive, we optimized the design of the solution so that, in general, it does not slow down transaction response time significantly. In fact, as said above, transactions must only check the immediately-prior version. The reason is that all other possible conflicts would have been detected by other siblings if they occurred.

In the example from Figure 1, T_{N2} would detect future siblings that read X_1. Say that some sibling T_{N5} also existed and read X_1. Then if T_{N5} had committed before it, T_{N2} would detect the conflict and would mark T_{N5} as invalid. If T_{N2} had already committed before T_{N5}, then T_{N5} itself would see its incorrect read, even if T_{N4} was not yet finished, as it would notice that it read X_1 instead of X_2. Thus, T_{N4} itself does not need to detect that conflict, and T_{N5} would expectedly read X_4 by the time it finished.

Transaction Commit. The second function of Algorithm 2 is the commit of the transaction which simply handles restarting its child transactions as necessary. While the number of successfully committed children is less than the number of child transactions, the transaction waits until a child has signaled completion (lines 35-37).

Upon the commit of a child, the transaction simply runs through all of the sub-transactions marked as invalid, decreasing *numCommitted* and restarting the children (lines 38-41). In our implementation, the transactions store a block of code execution when they first begin, thus the restarting of a transaction simply re-executes the block, similar to aborting inside of the transaction while it is running. Once all of the children are committed, the top-level read-set is updated using the *readHash* across all of the children, and the write-set is updated with the latest versions of written objects (lines 44-47). Afterwards, the normal commit of a top-level transaction is executed as described in Section 4.

6. Evaluation

We built SPCN into the Hyflow2 DTM framework [15], which is a high-performance software infrastructure for implementing distributed synchronization protocols, written in Scala. As a testbed, we utilized *Amazon EC2*. The experiments were run on up to 20 nodes of *c3.8xlarge* machines, where each machine is equipped with Intel Xeon E5-2680 v2 (Ivy Bridge) processors, 32 vCPU, and 60 GB of memory. Each data point is an average of 5 runs.

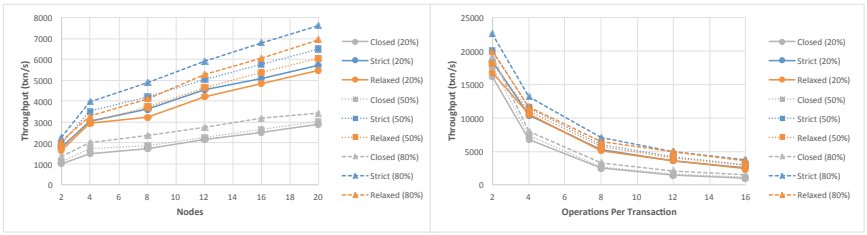
We contrast the performance of SPCN against the classical implementation of closed nesting, where sub-transactions are executed sequentially without overlap. In our deployment, each node runs a number of application threads.

Three benchmarks were utilized in experimentation: *Bank*, a benchmark that mimics traditional bank operations; *TPC-C* [6], a popular benchmark that simulates warehouse inventories and on-line processing of item orders; and *YCSB* [5], a benchmark that performs read and update operations on tables of data. We explored different configurations to show where SPCN is more effective.

6.1 Bank Benchmark

The Bank benchmark has two operations, *balance check*, which observes the values of bank accounts, and *transfer*, which withdraws money from one account and deposits it into another. We managed to vary the number of operations in order to show the impact of the number of sub-transactions per original transaction. The operations parameter (*ops*) defines how many nested sub-transactions there are. For transfer, each sub-transaction opens two bank accounts and transfers money between them. Thus, if *ops* is 4, then there are 4 sub-transactions, each operating upon 2 bank accounts, thus one root transaction uses 8 accounts.

In addition to *ops* and the number of application threads running on each node, the other parameters are the read



(a) Transfers per transactions = 8. (b) Number of nodes = 20.

Figure 2. Performance of Bank benchmark varying % of read-only operations using 500k accounts and 8 application threads per node.

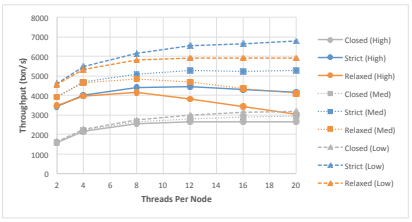
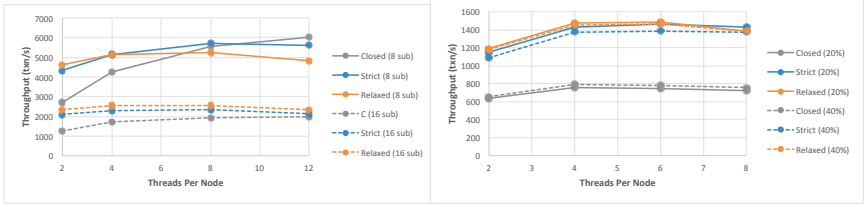


Figure 3. Bank benchmark using different contention levels with 8 operations.



(a) 50% conflict and 8/16 sub-tsx. (b) 4 variable-size sub-txs varying contention.

Figure 4. Bank; 20 nodes; 50% read; 50k accounts, with internal conflicts.

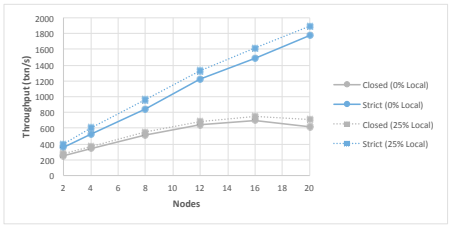


Figure 5. TPC-C; 8 threads per node.

percentage indicating how often balance check will occur as opposed to transfer, as well as the number of bank accounts created, which are spread randomly and evenly across the number of nodes running. Changing those parameters means changing the overall contention in the system (e.g., less number of objects means more contention).

Figure 2 shows the results of SPCN using the Bank benchmark. In both of the plots reported we configured Bank by deploying 500k total accounts in the system and each node runs 8 application threads. In these experiments we vary the contention in the system by changing the % of read-only transactions in the range of {20, 50, 80}. Figure 2(a) plots the total throughput of the system while increasing the number of nodes deployed. Strict substantially outperforms closed nesting by as much as 2.25× due to the exploitation of parallel activation of sub-transactions. The improvement increases along with the percentage of the read-only workload because clearly in this case conflicts are reduced and parallelism is more effective. Relaxed SPCN performs closely to Strict, though slightly worse, because all of the sub-transactions are balanced to the same size, thus the extra commit processing holds it down.

In Figure 2(b) we deploy 20 nodes and we increase the transactional load by increasing the number of operations made inside a single transaction. The performance clearly decreases while increasing the size of the transaction, but it is interesting to observe how SPCN constantly performs better than sequential closed nesting by as much as 2.72×. The speed-up of SPCN over closed nesting actually increases as the number of operations increases, although the total throughput decreases as the transactions become larger. With more parallel operations, Relaxed's processing does not hold as much weight as it did in the previous experiment, thus its performance is comparable to Strict.

In Figure 3 we vary the contention level by changing the number of deployed accounts rather than the percentage of read-only transactions, which is now fixed at 50%. We run on 20 nodes and also increase the number of application threads per node up to 20. Each transaction has 8 sub-transaction operations. From the plot it is clear that when the contention is low, thus sub-transactions are likely independent (we recall that accesses to accounts are random) and root transactions do not suffer from several aborts, then the parallel activation pays off. As the number of threads increases, the contention causes Relaxed SPCN to drop in performance, because the processing must be redone multiple times as there are more aborts. The maximum improvement we observed over closed nesting is 2.85×, 2.45×, and 2.2× in the low, medium, and high contention scenarios, respectively.

Lastly, we performed experiments that vary internal conflicts for parallel nesting. If two sub-transactions propagate data between each other (i.e., one writes an item and the other reads that new information), then parallel nesting encounters a conflict. The sub-transactions cannot go fully in parallel as the newly-written data needs the writer to sub-commit. For closed nesting, no overhead is added, as the sub-transactions always go sequentially. In Figure 4, we include

two experiments with internal conflicts running on 20 nodes, both using 50% read operations and 50k bank accounts.

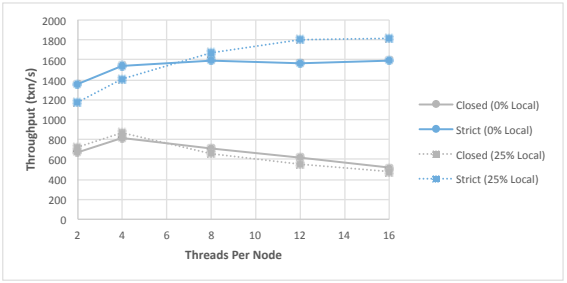
Figure 4(a) shows transactions running with a 50% chance of sub-transactions conflicting. The number of sub-transactions is changed from 8 to 16 per top-level transaction thread. With 8 sub-transactions, Relaxed starts roughly the same as Strict but performs worse as more transaction threads are added, as Relaxed's conflict processing adds overhead. Closed nesting even manages to overtake both of them in throughput. The reasoning here is that closed nesting essentially performs 4 operations, as 50% conflicts of shared data means that half of the sub-transactions only require re-reading local data. In order to execute in parallel, SPCN has multiple requests (likely at slightly different times) that could be for the same data. At 16 sub-transactions the parallelism pays off. Note that the throughput for each is less than before simply because each transaction is now longer. Here, Relaxed is the best. The parallel commits allow the internal conflicts to be processed earlier than they could be in Strict (due to the forced ordering of commits).

Figure 4(b) shows a different test with varied conflict chances of $\{20, 40\}\%$, with 4 sub-transactions that vary in size randomly from 1 to 8 operations. Each sub-transaction has a different computation time, which causes the in-order commit protocol of Strict SPCN to form a bottleneck, as shorter sub-transactions can potentially stall until earlier, longer sub-transactions commit. Relaxed's early processing overcomes that barrier and is also able to process conflicts earlier, allowing for better performance.

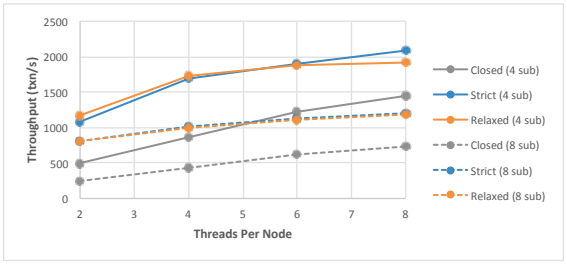
6.2 TPC-C Benchmark

TPC-C [6] is a larger benchmark simulating operations on warehouses. The default breakdown of its five different operations places the creation of new orders and the payment of previous orders as the most frequent. With this profile, 92% of the transactions executed are write transactions. We evaluated encapsulating any loop iterations as sub-transactions. As for parameters, we change the access locality skew, which is the percentage of transactions accessing the warehouse located on the same node where the transaction is executing. Decreasing this parameter increases the contention in the system because more network communication is required to execute and commit transactions. The benchmark is configured deploying one warehouse per node, thus contention in the system is high, as is usual for TPC-C. In the following plots (except for Figure 6(b)) we do not report the performance of Relaxed because it is very similar to Strict and we want to avoid overlapping lines.

Figure 5 displays results increasing the number of nodes (thus the number of warehouses and threads as well) and changing the transaction access skew from 0% (completely random accesses) to 25% (a slight bias to local accesses). As is shown, Strict is able to scale better than closed nesting, as it increases linearly while closed nesting decreases after 16 nodes. The reasoning here is the length of the TPC-C transactions and the high conflict rate. Because many transactions abort very often, and because closed nesting takes significantly longer to perform sets of transactions than Strict does, it is much slower in re-executing aborted transactions. Even though Strict faces high abort levels (even higher than closed nesting), its parallelized transactions can re-execute more quickly. In terms of locality, the scaling trends remain the same for both closed nesting and Strict, although 25% locality results in higher throughput for both of them. The largest performance increase over closed nesting in this chart is 2.9× and 2.68× for 0% and 25% local skew, respectively.



(a) Default settings using 20 nodes.

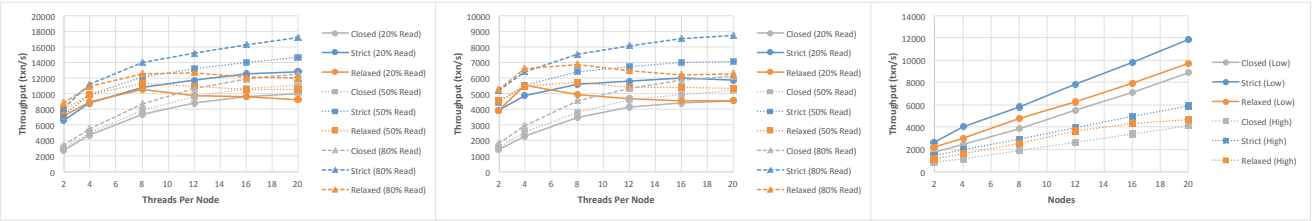


(b) Read-only transactions on 10 nodes.

Figure 6. Performance of TPC-C benchmark.

The results plotted in Figure 6(a) show the behavior of the two competitors while varying the application threads per node and the access skew, and we fix the system at 20 nodes. As is shown, at 0% local skew, Strict levels out due to the very high contention and high network communication from the majority of transactions often being external. Yet, Strict is able to sustain as the number of clients increases up to 320 total, while closed nesting decreases in performance, unable to handle the conflicts. Closed nesting decreases as well for 25% local skew, while Strict increases and performs better than itself at 0% local skew, leveling out at 12 threads per node. The reasoning here is that, with the local bias, transactions are less often external, thus Strict can re-execute aborted transactions even faster if they are local. The speed-up over closed nesting increases with the number of threads due to increasing contention and closed nesting decreasing in performance, with the maximum being 3.07× and 3.78× for 0% and 25% local skew, respectively.

Figure 6(b) demonstrates TPC-C configured with all read-only transactions. Here, there is no possibility of con-



(a) Low Contention (100 rows, 10 accesses). (b) High Contention (50 rows, 20 accesses). (c) 12 threads/node and 20% read-only.

Figure 7. Performance of YCSB varying contention and read-only %.

flict because no write is invoked. In the plot we report the performance of having 4 and 8 sub-transactions per original transaction. Clearly, Strict and Relaxed SPCN are nearly the same without potential write conflicts. Relaxed slightly worsens with more root threads per node because of its processing, thus slowing commits down as there are more threads than the cores can typically handle. The best speedup is 2.35× and 3.28× for 4 and 8 sub-txns, respectively.

6.3 YCSB Benchmark

The YCSB [5] benchmark originated as a means of testing database information with SQL-style operations. For testing SPCN, the benchmark simply represents tables of information with different ways to manage the setup and contention possibilities. The main parameters used for the benchmark are the number of rows within a table, the number of cells within a row (or number of columns), the number of cells that a transaction accesses or operates upon, as well as the locality skew and the percentage of read-only transactions. Note that in all of the below experiments, we kept the locality skew at 0%, thus random and more often external requests are made.

One table is allocated per node, and there are two main operations, read or update. Transactions begin by choosing a table from a node, then by choosing a row in the table. The number of rows strongly influences contention, as does the number of accesses performed by a single transaction. For low contention across 20 nodes, we gave each table 100 rows and 100 cells per row, with transactions accessing 10 cells at a time. For high contention, we cut the number of rows in half to 50, kept the cells at 100, and doubled the number of cells accessed to 20 per transaction.

In Figure 7, we hold the number of nodes in the system at 20 and vary the number of threads executing per node, the read % of transactions, and the contention level. At both contention levels, closed nesting and Strict follow similar trends, although the average throughput at high contention is roughly half the throughput at low contention. Further, Strict remains better than closed nesting at the vast majority of settings. In both charts, the bottom Strict line is at 20% read transactions and intersects with the top closed nesting line at a high number of threads. But that closed nesting line is at 80% read transactions, meaning that closed nesting only reaches similar performance to Strict when Strict has 4× the number of write transactions, therefore much more conflict. The best speed-up for Strict here is 2.46× for low- and 2.9× for high-contention. Relaxed cannot deal with the growing contention across 20 nodes, although it begins by outperforming Strict as seen in Figure 7(b) using 2/4 threads.

In Figure 7(c), we fix the number of threads at 12 per node and 20% read-only transactions (thus, more conflict), while we vary the contention level and the number of nodes. In this experiment, all competitors scale linearly with the number of nodes. Across the two contention levels, the maximum performance gain for SPCN is 1.69×.

7. Conclusion

In this paper we presented SPCN, an efficient technique for activating distributed (closed) nested transactions in parallel, thus overlapping their execution (including network interactions). As the evaluation shows, enforcing the same serialization order as in the original transaction is possible and very effective when transactions are distributed.

Acknowledgments

This work is supported in part by US National Science Foundation under grant CNS-1523558, and by US Air Force Office of Scientific Research under grant FA9550-15-1-0098.

References

[1] K. Agrawal, J. T. Fineman, and J. Sukha. Nested parallelism in transactional memory. In *PPoPP*, pages 163–174, 2008.

[2] W. Baek, N. Bronson, C. Kozyrakis, and K. Olukotun. Implementing and evaluating nested parallel transactions in software transactional memory. In *SPAA*, pages 253–262, 2010.

[3] J. a. Barreto, A. Dragojevic, P. Ferreira, R. Filipe, and R. Guerraoui. Unifying thread-level speculation and transactional memory. In *Middleware*, pages 187–207, 2012.

[4] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987. ISBN 0-201-10715-5.

[5] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, pages 143–154, 2010.

[6] T. Council. TPC-C Benchmark. 2010.

[7] A. Dhoke, B. Ravindran, and B. Zhang. On closed nesting and checkpointing in fault-tolerant distributed transactional memory. In *IPDPS*, pages 41–52, 2013.

[8] N. Diegues and J. Cachopo. Practical parallel nesting for software transactional memory. In *DISC*, pages 149–163, 2013.

[9] S. M. Fernandes and J. a. Cachopo. Lock-free and scalable multi-version software transactional memory. In *PPoPP*, pages 179–188, 2011.

[10] J. Gray and L. Lamport. Consensus on Transaction Commit. *ACM Transactions on Database Systems*, 31(1):133–160, 2006.

[11] E. B. Moss. Nested transactions: An approach to reliable distributed computing. Technical report, Cambridge, MA, USA, 1981.

[12] J. E. B. Moss and A. L. Hosking. Nested transactional memory: model and architecture sketches. *Sci. Comput. Program.*, 63:186–201, 2006.

[13] S. Peluso, P. Romano, and F. Quaglia. Score: A scalable one-copy serializable partial replication protocol. In *Middleware*, pages 456–475, 2012.

[14] S. Peluso, P. Ruivo, P. Romano, F. Quaglia, and L. Rodrigues. When scalability meets consistency: Genuine multiversion update-serializable partial data replication. In *ICDCS*, pages 455–465, 2012.

[15] A. Turcu, B. Ravindran, and R. Palmieri. Hyflow2: A high performance distributed transactional memory framework in scala. In *PPPJ*, pages 79–88, 2013.