NestedBTO: A Timestamp-Based STM Protocol for Closed Nested Transactions with Opacity Guarantees*

Nischay Ranjan[1,2][0009−0009−0203−9201], Rohit Kapoor[1][0009−0001−5855−3367], and Sathya Peri[1][0000−0002−3471−7929]

[1] Indian Institute of Technology Hyderabad, India
[2] GEC Sheohar, Department of Science, Technology and Technical Education, Government of Bihar, India
cs21resch11012@iith.ac.in,nischayranjan.dstte@bihar.gov.in, sathya_p@cse.iith.ac.in, rohitkapoor9312@gmail.com

Abstract. Software Transactional Memory (STM) provides a high-level abstraction for managing concurrency, alleviating the complexity of traditional lock-based programming. While nesting support is critical for enabling composability and reducing the vulnerability window of long transactions, existing STM protocols for closed nesting often fall short in either offering rigorous correctness guarantees or demonstrating practical performance benefits. This paper introduces NestedBTO, a new STM protocol for closed nested transactions that uses timestamp ordering to allow child transactions to be executed in parallel while maintaining consistency according to the opacity criterion. NestedBTO satisfies opacity for all transactional histories, as demonstrated by our formal proof of correctness. We conduct extensive experimental evaluations using application-level workloads STAMP and concurrent red-black tree and hashtable benchmarks to assess its effectiveness. The results reveal that NestedBTO achieves substantial improvements in throughput and abort rates compared to other state-of-the-art STM, i.e, NesTM. These findings demonstrate that NestedBTO offers a scalable and practical solution for high-performance transactional memory systems.

Keywords: Software Transactional Memory · Opacity · Closed Nesting.

1 Introduction

The multicore architecture has gained predominance in the design of modern computing systems. To gain the most out of it, concurrent programming has gained interest. However, developing a concurrent program is significantly more complex than developing its sequential counterpart. Locks have traditionally been used broadly for implementing parallelism. However, lock-based programs pose complex issues with synchronization and are susceptible to various problems such as deadlocks and priority inversion. Moreover, the composition of

* This work is partly supported by Core Research Grant CRG/2022/009391

larger software systems by integrating smaller components with locks creates serious complications[11]. A Software Transactional Memory (STM) system [23, 12] enables concurrent processes to coordinate and communicate by performing transactions on shared memory. A transaction represents a sequence of operations that executes atomically—appearing as a single, indivisible action—even when other transactions are running concurrently. Within a transaction, memory operations can be performed on transactional objects (*t-obj*). If a transaction completes successfully, it commits, making all its updates visible to other transactions. If a transaction fails, it aborts, causing all its modifications to be undone, ensuring that no changes are observed by other transactions. Access to *t-obj* is permitted exclusively through transactions.

When, during the execution of a transaction, another transaction is invoked, the process is said to be a transaction nesting [10]. Nested transactions take the elementary transactional model to a multi-level structure, in which a group of sub-transactions can recursively call additional sub-transactions, forming a transactional tree. Different varieties of nesting exist, notably closed, open, and flat nesting [17]. In closed nesting, the failure (abort) of a child transaction does not directly affect the outcome of its parent transaction. However, it is significant to note that the updates of a child transaction become visible to other transactions only after a successful parent transaction is committed.

A primary reason for instituting transaction nesting into STM systems was to obtain composability requirements. **Composability** is the essential property of modular programming, allowing independent components to be assembled into a larger, more complex structure. In the case of transactional systems, this means placing together several subtransactions into one larger entity called a transaction. Another driving force is to mitigate the problems long-running transactions pose, particularly those with extended vulnerability periods. In this regard, the vulnerability period is when a transaction is still vulnerable to conflicts with other transactions that run concurrently. A practical remedy would be to decompose a long transaction into several smaller subtransactions, thereby executing them over multiple cores so that both execution time and vulnerability windows are minimized. Such a strategy is starting to gain traction in the context of transactional data structure library implementation[1, 24, 6, 15].

Considering the widespread nested parallelism, such as nested parallel loops, recursive function calls, and calls to parallel libraries in real-world applications, these nested situations also motivate the developers of nested STM systems[25]. Nested calls are equally important in smart contract execution in the blockchain; that is, they aid in composability but also in interaction complexities. Designing an STM protocol which is able to support transactions nesting is really a non-trivial task; the difficulty of figuring out contention across several different levels of nesting, while at the same time making certain that the executions of such nesting are consistent in terms of sequential semantics, adds complexity to this problem[9, 13, 18, 7]. When multiple transactions—nested or otherwise—access shared data, it is essential that their executions remain correct. Opacity has been widely accepted as the key correctness criterion for concurrent transactions in

STM systems. Although several STM implementations have been proposed to support nested transactions, existing efforts focus mainly on two nesting models: linear nesting[11, 16, 22, 5] and parallel nesting[2, 3, 14, 25, 20]. However, none of these models provides a formal guarantee of opacity. Although systems like HparSTM[14] offer an informal discussion on supporting opacity, they do not provide a rigorous proof. Addressing this gap, Peri et al.[18] formally extended the notion of opacity to nested transactions and identified a class of schedules that preserve opacity under closed nesting.

In concurrent transaction execution within software transactional memory systems, **Opacity**[8] has stood as a fundamental correctness criterion. In addition to ensuring serializability, Opacity requires that even the aborted transaction can have the illusion of no concurrency, i.e, the aborted transaction should also view the consistent state of the shared memory. The challenge of ensuring opacity becomes particularly sophisticated within the context of nested transactions, where an aborted parent transaction may have committed children and vice versa.

This paper introduces **NestedBTO**, a Software Transactional Memory system that supports parallel closed nesting based on timestamp ordering. It extends the timestamp-based approach introduced in our earlier study[21]. Our key contribution lies in proposing a parallel-nested STM system and formally proving that our protocol adheres to the class of schedules defined by Peri et al. for closed nested transactions. This proof of correctness establishes that **NestedBTO** satisfies opacity under arbitrary nesting depths—something not demonstrated in prior works.

Our core contributions (in section 3) are as follows:

1. We present a novel STM system that supports **closed nesting to arbitrary depth** while **formally guaranteeing opacity.**
2. The system enables **parallel execution of subtransactions**, supports **adaptive flattening** of shallow nests, and allows **work-stealing** across independent subtrees to improve performance.
3. We design a **new timestamp protocol** that ensures **serializability and correctness** across unrestricted levels of nested transactions.

In addition, we conduct an extensive performance evaluation of our STM system, comparing it against flat nesting and serial execution on benchmark data structures such as hash tables, counters, Red Black tree, and STAMP. Our results demonstrate that **NestedBTO** maintains theoretical correctness and delivers practical scalability.

Roadmap: The remainder of this paper is structured as follows: The preliminary data and the system model are presented in Section 2. The **NestedBTO** algorithm, as designed and with its operating semantics, is addressed in greater detail in Section 3. Section 4 presents an experimental algorithm evaluation that compares its performance with existing approaches. Finally, Section 5 concludes the paper and discusses potential directions for future research.

2 Preliminaries & System Model

2.1 Transaction Tree

A transaction consists of a sequence of read and write operations on transactional objects *t-obj* , and may invoke subtransactions that can themselves recursively spawn further subtransactions. This hierarchical execution naturally forms a transaction tree (see Figure 1), where each node represents a transaction or an operation, with simple memory operations as leaves. Transactions perform two types of operations: *memory operations* and *transactional operations*, including *commit* and *abort*. When a transaction t_p invokes a subtransaction t_{pi}, a new child node is added under node t_p in the transaction tree. If t_p completes successfully, it issues a commit; otherwise, it ends with an abort. Aborted subtransactions (e.g., t_{qi}) lead to the removal of their subtree, while committed ones remain until garbage-collected. The root transaction t_0 sits at level 0 and manages globally shared *t-obj* . Transactions directly under t_0 are referred to as top-level transactions. Any transaction t_x with k children is denoted as t_{x1}, t_{x2}, ..., t_{xk}. Transactions use two local buffers: a *read_list* and a *write_list* . For a read $r_x(y)$, the transaction first checks its local *write_list* , then the read list, and finally queries ancestors up to t0 if necessary. Writes $w_x(y)$ add the object to the *write_list* . Each transaction begins with memory operations and concludes with subtransactions and terminal operations. A parent transaction executes its terminal operation only after all its subtransactions complete. On commit, a subtransaction's *write_list* merges into its parent's. On abort, its updates are discarded.

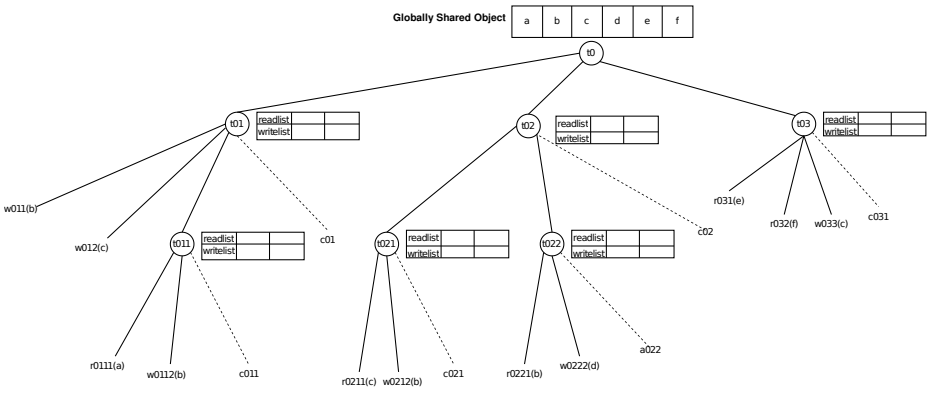


Fig. 1: Transaction Tree

2.2 Computation Model

We have developed a system that enables a set of n processes (or threads), denoted as $p_1, p_2, \ldots, p_n$, to interact with a shared collection of transactional ob-

jects (t-obj) using atomic transactions. Each process can perform a predefined set of operations within a transactional context. A new transaction is started by the *begin* operation. The value of the object x is retrieved by the *read(x)* operation from either the local state of the current transaction or, in the event that it is absent, from one of its ancestor transactions. The transactional object x's value is locally updated to v within the transaction by the *write(x,v)* operation. To enable nested execution, the *assignthread(t_i)* operation creates a subtransaction as a child of transaction t_i. The *tryC()* operation is used to try to commit a transaction. If it succeeds, the updates from the transaction are sent to the buffer of its parent, and the operation returns a *commit* (in short, $\mathcal{C}$); if not, it returns an abort $\mathcal{A}$). The *tryA()* operation can also be used to explicitly abort a transaction, ending it and returning an abort.
It's important to note that both the *read* and *TryC* methods have the potential to return $\mathcal{A}$ which signifies that the underlying algorithm has forcefully aborted the transaction. However, in cases where $\mathcal{A}$ is not returned, it indicates that the operations have been executed successfully without any forced termination. Additionally, each transaction adheres to the following constraints: (1) a transaction executes one operation at a time, and (2) a parent transaction must wait until all of its children have terminated before entering the validation phase for its commit.

2.3 Histories and Base Formalism

A history is a sequence of events encompassing invocations and responses to transactional operations. The set of events is denoted as $H.evts$. In the context of Transaction history, considering an execution sequence, let H' represent the set of transactions invoked during that execution. The order relation between transactions, denoted as $\prec_{H'}$, is defined as follows:

1. $t_1 \prec_{H'} t_2$;t_1 terminates before t_2
2. $t_1 \not\prec_{H'} t_2$;t_1 and t_2 are concurrent.

At the transaction level, the execution is defined as the partial order $\widehat{H'} = \langle H', \prec \rangle$, referred to as the **transaction history**. A transaction history is deemed *sequential* if no (sub)transactions within it occur concurrently. A sequential transaction history is *legal* if each read operation returns the last write on the same object.
Level-wise history: In the context of closed nested STM, each transaction node t is associated with a local buffer, which functions as a shared memory for its descendants(i.e its child and sub-child transactions). Consequently, every level or node in the transaction tree possesses a distinct, *level-wise history*, which reflects the operation performed within the scope of that transaction and its descendants. We denote this history as $\widehat{H'_t} = \langle H'_t, \prec_{H'_t} \rangle$ representing a level-wise transaction history at t, where $\widehat{H'_t}$ represents the set of operations visible at node t and $\prec_{H'_t}$ captures the partial order among those operations. Similarly, a level-wise sequential history at node t, representing a total order consistent with

the observed operations, is denoted as$\widehat{H'}_t^{seq} = \langle {H'}_t^{seq}, \prec_{{H'}_t^{seq}} \rangle$. Here, references to other transactions (e.g., parent or sibling) are restricted to those within the same transaction tree—i.e., sharing the same root and nested structure—not from separate threads or trees.

External read: An external read operation within a transaction t denotes the reading of a transactional object x when it is not locally present within the transaction's buffer. Instead, t accesses x from the nearest ancestor (in the bottom-up hierarchy), possessing a local copy of x. In cases where x is not found in any ancestor's buffer, the transaction t resorts to the root transaction, where it would certainly find the *t-obj* x. Illustrated in Figure 1, the operation $r_{0211}(c)$ exemplifies an external read for both t_{021} and t_{02}.

Propagated read objects: Propagated read objects refer to instances where a subtransaction t reads a transactional object x from the local buffer of an ancestor t', which is different from its immediate parent t_p. In executing the read operation, t propagates the read list of all its ancestors until t', ensuring that x is included in the read list. Consequently, the value of x retained in t_p and other ancestors' read lists aligns with the value obtained during t's read operation. This propagated value of x remains accessible to subsequent descendants of t_p and its ancestors. Illustrated in Figure 1, the operation $r_{0211}(c)$ propagates the read buffer of t_{02}, thereby enabling its utilization by t_{022}.

Commit write: In close nested transactions, the commit write operation occurs when a transaction t commits its operations. During this process, the *write_list* of t is merged with that of its parent transaction and updates all the *t-obj* , which is common in parent and child transactions. If a transactional object is absent in the parent's write list, it is added accordingly. Concerning Figure 1, consider t_{021}; upon committing, the transactional object b in its write list integrates into t_{02}'s write list. Similarly, upon t_{011} committing, the transactional objects in its write list update the corresponding values in t_{021}'s write list.

Correctness of the closed nested transactions: Guerraoui and Kapalka [8] introduced opacity as a correctness criterion for software transactions, ensuring schedule consistency. Opacity requires that an equivalent serial schedule maintains the original schedule real- time ordering and lastWrites of read operations from the original schedule, even accounting for aborted transactions treated as read-only in serial schedules. In nested transactions, complexity arises from aborted transactions that potentially have committed sub-transactions, impacting the correctness of subsequent operations. Committed sub-transactions of aborted ones can have commit-writes, affecting other transactions. The **Closed Nested Opacity (CNO)** definition [19] extends the traditional notion of opacity to system with closed nested transactions. *CNO* ensures that there exists a serial schedule preserving the order of operations, the partial order among transactions, and the lastWrite equivalence for read operations. Unlike standard opacity, *CNO* accounts for the added complexity of nested transactions by guaranteeing consistent reads across all transactions, including those within aborted subtransactions. This redefined definition is essential for maintaining correctness in systems that support nested transactional execution.

3 Nested Basic Timestamp Ordering Algorithm

This section presents a timestamp-based algorithm designed for the closed nested STM system, facilitating the parallel execution of nested transactions. Referred to as **Nested BTO**, this algorithm has been developed to enhance the efficiency of transactional processing within a closed nested environment. Subsequently, we establish our algorithm's robustness by providing proof demonstrating its adherence to the opacity correctness criteria. Graph characterization achieves this validation by showcasing the soundness and reliability of the Nested BTO algorithm in ensuring opacity.

The Working Principle

The algorithm employs an invariant approach for assigning unique identifiers (a string), which also function as timestamps, to transactions. For instance, if a transaction holds the identifier i, it will also be its timestamp. Intuitively, the timestamp tells about the "time" the transaction began. Adding numerical suffixes to the parent transactions' identifiers helps in the generation of string identifiers for their child transactions. Hence, this design enables the lexicographic comparison of transaction identifiers or timestamps, providing a structured means of considering their chronological order.

The invoking transaction's timestamp accompanies its every read and write operation within the system. Each transaction is associated with the #child, term_count, and status variables. The variable #child aids as an indicator of the number of children associated with a given transaction. The variable term_count accounts for the count of child transactions that have terminated either successfully committed or aborted. The variable status provides information about the current state of the transaction. Furthermore, every *t-obj* is associated with two variables: *max-r* and *max-w*. These variables store the timestamp of the most recent transaction that successfully executed a read and write operation on the respective transactional object.
Now, we will discuss the core concepts underlying the read, write, and tryC operations performed by a transaction T_i. The inspiration for these principles is drawn from the timestamp algorithms crafted by Bernstein and Goodman for databases[4]. The following operational principles are presented first informally; their full algorithmic realization (with timestamps and pseudocode) appears later.

1. **Read Principle:**When transaction T_i invokes a *read(x)* operation, it retrieves the value v from either its own local buffer or its ancestors' local buffers (say T_p). If x is found in the *write_list* of an ancestor T_p, the read principle checks whether the same object x has been concurrently updated in T_p's *write_list* by another transaction T_j, where T_j's timestamp is greater than that of T_i. If such a conflict exists, a recursive abort is triggered on T_i, which cascades to abort all of its descendants. Otherwise, the *max_r(x)* field in the T_p's *write_list* is updated with T_i's identifier. If x is instead found in

the ancestor's *read_list* , the *max_r(x)* field in the local *read_list* is updated with T_i. It is important to emphasize that T_i and T_j are part of the same transaction (sub)tree at T_p.

2. **Write Principle:** Transaction T_i updates data in its local buffer, precisely to *write_list*.
3. **Commit Principle:** Before committing, a transaction T_i invokes a series of checks:
   (a) *Child Termination Check:* At first, T_i verifies whether all of its child transactions (if any) have terminated. It is done by comparing the invariants #child count with term_count. If not all children are done, T_i waits.
   (b) *Validation for Each t-obj x in Its write_list :*
      i. If a transaction T_k (which is a sibling or a descendant of a sibling of T_i, and part of the same transaction tree rooted in parent(T_i)) has read x from parent(T_i), and the timestamp of T_i is less than that of T_k (that is, $i < k$), then T_i must abort.
      ii. If a sibling transaction T_k (i.e., a transaction with the same parent as T_i) has written to x in parent(T_i), and $i < k$, then again, T_i must abort.
   (c) *Commit Approval:* If none of the above conflict conditions are met, the transaction T_i is allowed to commit successfully.

The Protocol

Let T represent the set of transactions, X denotes the set of objects, and V signifies the possible values associated with these objects. Furthermore, we assume a thread pool comprises n threads, with each transaction invoked by a thread from this designated pool.
The STM system comprises the following operations or functions, each executed in response to a transaction's initiation, reading, writing, or attempt to commit.

initialize(): This operation initializes the STM system. In this process, it assigns an identifier of 0 to the root transaction and designates its parent as NULL. Furthermore, the STM system is presumed to possess knowledge of all *t-objs* ever accessed. During initialization, these *t-objs* are assigned a value of 0 by the root transaction and subsequently included in its writelist.
$\mathbf{begin_i()}$: The operation allows a transaction to invoke other transactions involving a thread from the thread pool, invoking another (sub)transaction. Notably, all top-level transactions are invoked by t0. In the case of a (sub)transaction t_j invoked by another transaction t_i, its identifier is assigned by appending its position in the child list of t_i to the identifier of t_i. This operation also sets the status of any newly invoked transaction to ACTIVE.
$\mathbf{read_i()}$: Through this operation, a transaction initiates a read operation; it first searches for the corresponding *t-obj* in its local buffer, *write_list* followed by the *read_list* It searches in its ancestor transactions if it fails to locate the required *t-obj* in its local buffer. This search process begins with its immediate parents

and may extend upwards in the transaction hierarchy, eventually reaching the root transaction t_0. Once the transaction successfully locates the required *t-obj* , it traverses back down the order, populating the local read buffers of all the ancestor transactions it encountered with its timestamp information.

write$_i$(): The write operation creates and appends *t-obj* x to the *write_list* of the t_i, if it is not there. Otherwise, it updates the value of the *t-obj* x.

traverse_up$_i$(x): This operation examines whether the transactional objects *t-obj* x are present in the ancestors of transaction t_i. A transaction stack stores the ancestors of t_i, and the search may extend up to t_0.

traverse_down$_i$(): Upon locating the necessary *t-obj* in the ancestors' buffer, it is propagated downward into the local read buffer *read_list* of all the intermediate transactions until t_i.

TryC$_i$(): This operation is invoked when a transaction t_i intends to commit, aiming to update the content of *t-objs* in its *write_list* to its parent's *write_list* The process involves several steps:

1. *Check for Children Termination:* Firstly, t_i checks whether all its children (if any) have terminated.
2. *Iterative Check for Updates*: For each *t-obj* x in the *write_list* of t_i, the following checks are performed:
   Suppose t_i and t_k are siblings with t_p as their parent-
   (a) *read check:* Suppose t_k has read x from t_p, *max-r(x)* in t_p's *read_list* or *max-w(x)* in t_p's *write_list* (depending on the source of the read) stores the timestamp of t_k. If the timestamp of t_i is less than the timestamp of t_k, t_i is considered too late for the update, leading to its abortion.
   (b) *update check:* If t_k has already performed an update on t_p's *write_list* for *t-obj* x, *max-w(x)* in the *write_list* stores the timestamp of t_k. If the timestamp of t_i is less than the timestamp of t_k, t_i is deemed too late for the update, resulting in its abortion.
3. *Reflect Updates to Parent*: If the checks are successful for all *t-objs* in t_i's *write_list* , the proposed updates by t_i are reflected in its parent's *write_list*.

Note that for Root Transaction t_0, the tryC protocol ensures all child transactions have terminated, commits without conflict validation (no siblings/parent), and merges its *write_list* into the global state.

Abort$_i$(): This operation is responsible for aborting the current transaction. It is invoked when the validation within the *TryC* operation for a specific transaction t_i fails. In such cases, the system triggers this operation to handle the abortion of the transaction t_i.

Rec_abort$_i$(): In the event of a transaction aborting at a higher level of the hierarchy, all its children must also be aborted. This scenario unfolds when a transaction traverses upward to read a particular *t-obj* say x, and encounters an ancestor, say t_p, that undergoes abortion. The entire subtree rooted at t_p is also aborted in this case. It's essential to note that this mechanism differs from the standalone Abort() operation, where a transaction independently aborts itself without affecting its children or ancestors.

Consider the following scenario depicted in the figure 2 to explain the functionality of the algorithm: Suppose t_p is a top-level transaction with t_{p1} and t_{p2}

as its two children. Following our assumption that t_p invokes memory operations first and then subtransaction operations, it initiates a read operation on *t-obj* y and z, adds them to its *read_list* The *max-r* of these variables is then set to t_p. Subsequently, when t_p attempts to invoke t_{p1} and t_{p2}, and if threads are available in the thread pool, t_{p1} and t_{p2} are concurrently invoked by different threads. Assuming enough threads are available, t_{p1} and t_{p2} are invoked concurrently. In this context, consider the scenario where t_{p1} has a descendant t_i that initiates a *read(x)*. The read operation of t_i traverses upward, with each ancestor attempting to locate the transactional object x in its parent's buffer. Concurrently, t_{p2} is engaged in a *write(x)* operation on its local *write_list*

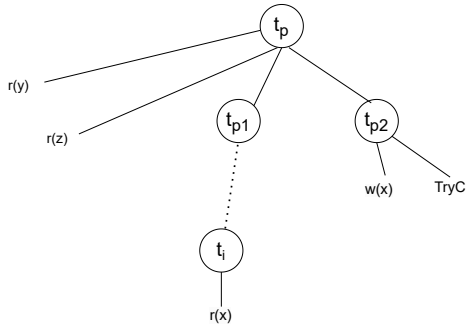


Fig. 2

In this context, let's examine different cases:

1. **TryC - read conflict**: If t_{p2} successfully updates before t_{p1} completes its search for x in t_p's buffer, x is added to the *write_list* of t_p *max-w(x)* $= t_{p2}$. When t_{p1} attempts to locate x in t_p's buffer (checking *write_list* first and then *read_list*), it discovers that a later transaction (t_{p2} in this) case has appended x to t_p's *write_list* Consequently, *Rec_abort()* of t_{p1} is invoked, leading to the abortion of t_{tp1} along with all its descendants.
2. **Read - TryC conflict**: If t_{p1}'s search for x precedes t_{p2}'s update operation, t_{p1} can traverse upward until t_0 where it definitively finds x. Subsequently, it traverses down, adding x to the *read_list* of t_p, while t_{p2} appends x to the *write_list* of t_p. Any subsequent $read(x)$ operations by descendants of t_{p1} retrieve the propagated value in t_{p1}'s and its descendant's *read_list*
3. **TryC - TryC conflict**: As updates from (sub)transactions are reflected in the parent's *write_list* , consider a scenario where a descendant of t_{p1} executes a $write(x)$, leading to x being appended to t_{p1}'s *write_list* Consequently, when t_{p1} invokes the *tryC* operation and conflicts with t_{p2}, t_{p1} has to invoke *Rec_abort()* if t_{p2} appends x to t_{p1}'s *write_list* first. Alternatively, both can terminate successfully if t_{p1} commits first, with t_{p2} overwriting t_{p1}.

Theorem. *The history generated by the NestedBTO algorithm satisfies the* ***Conflict preserving close nested opacity(CP-CNO)*** *[19].*

4 Evaluation

Experimental Setup: To evaluate the efficacy of NestedBTO, we compared its performance with NesTM using the STAMP benchmark suite and also against microbenchmarks such as concurrent red-black tree and concurrent hashtable. We have chosen the NesTM as it is the closest point of comparison. The experiments were carried out on a 16-core Intel Xeon processor (32 threads with hyperthreading) with 64GB of RAM. All implementations were compiled using GCC 11.3 with -O3 optimization flags.

4.1 STAMP

For the **STAMP** benchmark, the transactional workload was designed with 3–5 operations per nesting level, reflecting fine-grained parallelism common in realistic applications. For top-level transactions, NestedBTO consistently exhibited a 5–10% reduction in normalized performance difference (NPD) compared to NesTM. This improvement is attributed to NestedBTO's use of timestamp-based conflict resolution, which avoids lock contention during commit phases. In contrast, NesTM's commit-lock mechanism and repeated read-set validation introduce additional overheads, particularly in benchmarks with frequent barriers, such as intruder and vacation.

When evaluating scalability with increased nesting levels (NL ≥ 1), NestedBTO demonstrated superior performance. Its lexicographic timestamp ordering led to fewer redundant aborts by enforcing a deterministic serialization order. NesTM, on the other hand, experienced a performance bottleneck due to its lock-based commit protocol, especially at nesting levels of 3 or more, where throughput dropped by 15-20% in high-contention scenarios. Furthermore, NestedBTO's smaller transactional footprints resulted in better cache efficiency, reducing L1/L2 cache misses by approximately 12–18% based on simulation traces. NesTM's approach of merging metadata accesses adversely affected temporal locality. In benchmark-specific evaluations, both protocols performed similarly in low-conflict workloads such as the labyrinth, with less than a 2% difference in NPD. However, in high-conflict workloads, such as intruders, NestedBTO significantly reduced abort rates by 25 to 30% through early conflict resolution using recursive timestamp checks. The aggressive validation strategy of NesTM in such workloads led to more frequent rollbacks.

The figure 3 summarizes the normalized performance difference (NPD) for three representative STAMP benchmarks, measured using 16 threads:

The evaluation results affirm that NestedBTO's design, including timestamp-based conflict resolution and lightweight metadata, offers measurable performance gains over NesTM in systems with fine-grained transactional workloads. However, there are two key limitations to note. First, while NestedBTO performs well for nesting depths up to two, deeper nesting levels may introduce timestamp comparison overheads that require further optimization. Second, the current results are derived from simulations; validating the findings using real hardware

Algorithm 1 NestedBTO Protocol

enum State ABORT = -1, ACTIVE = 0, COMMIT = 1
SUCCESS = 1, FAILURE = -1

State of the transaction object
1: *oid* ∈ I
2: *val* ∈ V
3: *lock obj*
4: *max_r, max_w* ← **NULL**
State of the transaction
5: *tid* ← **NULL**
6: *status* ← **ACTIVE**
7: *lock trans*
8: *#child, term_count* ← 0
9: *read_list,write_list* ⊆ X
10: *parent's id* ← NULL
11: *child_list* ⊂ T
STM_initialize()
12: *t0.id* ← 0
13: *t0.parent* ← NULL
14: *objcounter* ← 0
15: **for all** x ∈ X used by the STM system **do**
16: *x.id* ← ++*objcounter*
17: add *x* to *t0.wl*
18: **end for**
STM_begin$_i$()
19: new trans t
20: t_j*.parent* ← t_i
21: t_j*.id* ← t_i*.id.append(++#child)*
22: t_j*.status* ← **ACTIVE**
STM_read$_i$(x)
23: **if** t_i*.status* ← **ABORT then**
24: return **FAILURE**
25: **end if**
26: search locally, *write_list* , and *read_list*
27: **if** *x is present locally* **then**
28: return *x.val*
29: **else**
30: *val* ← *traverse_up$_i$()*
31: return *val*
32: **end if**
STM_write$_i$(x,val)
33: **if** t_i*.status* ← **ABORT then**
34: return **FAILURE**
35: **end if**
36: **if** *x is in the writelist* **then**
37: *x.val* ← *val*
38: **else**
39: *create t-obj x in the writelist*
40: *x.val* ← *val*
41: *x.max_w* ← t_i*.id*
42: **end if**
STM_TryC$_i$()
43: **if** t_i*.status* ← **ABORT then**
44: return **FAILURE**
45: **end if**
46: **while** t_i*.#child* < t_i*.term_count* **do**
47: wait
48: **end while**
49: **for all** *d* in *ti.write_list()* **do**
50: /* Lock the *t-objs* in predefined order to avoid deadlocks */
51: *lock obj(d)* in t_i*.parent writelist and readlist*
52: **end for**
53: **for all** *d* in t_i*.write_list()* **do**
54: **if** t_i*.parent.readlist[d].max_r* > t_i*.id* OR t_i*.parent.writelist[d].max_r* > t_i*.id* OR t_i*.parent.writelist[d].max_w* > t_i*.id* **then**
55: **for all** *d* in *ti.write_list()* **do**
56: *unlock obj(d)* in t_i*.parent writelist and readlist*
57: **end for**
58: *Try_abort*(t_i)
59: return **FAILURE**
60: **end if**
61: **end for**
62: **for all** *d* in *ti.write_list()* **do**
63: update t_i*.parent.writelist[d].val* ← *d.val*
64: update t_i*.parent.writelist[d].max_w* ← t_i*.id*
65: **end for**
66: **for all** *d* in *ti.write_list()* **do**
67: *unlock obj(d)* in t_i*.parent writelist and readlist*
68: **end for**
69: t_i*.parent.term_count++*
70: t_i*.status* ← *COMMIT*
71: t_i*.parent.term_child_list.remove*(t_i)
72: delete t_i
73: return **COMMIT**

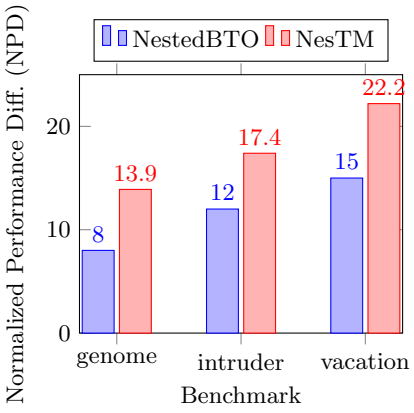
performance counters, such as cycle counts and cache miss rates, would further strengthen the analysis.

4.2 Red Black Tree

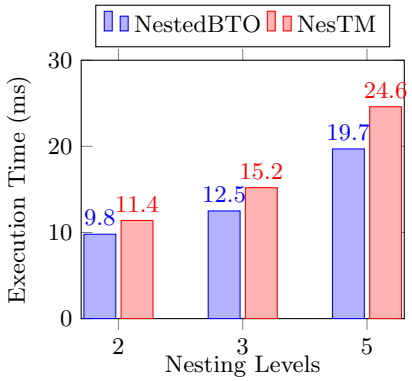
Experimental Setup: We evaluated both protocols using a concurrent red-black tree microbenchmark under the following conditions: a thread pool containing 128 threads and a Non-uniform memory access (NUMA) architecture. The workload consisted of 20% writes (insert and delete operations) and 80% reads (lookup) per transaction. Performance metrics comprise throughput (in transactions per millisecond), abort rates and latency distributions.
Performance Analysis

1. **Throughput scaling with nesting depth:** NestedBTO showed significantly better scalability than NesTM as nesting levels increased. For example, with a nesting depth of 2, NestedBTO completed the benchmark in 9.8 milliseconds compared to 11.4 milliseconds by NesTM, resulting in a performance gain of 16.3%. At levels 3 and 5, the performance improvements were 21. 6% and 24. 8%, respectively. This performance gain can be attributed to NestedBTO's timestamp-based conflict detection, which minimizes synchronization overhead between parent and child transactions. By contrast, NesTM's commit-locking mechanism suffers from increasing validation complexity as the nesting depth grows, roughly scaling quadratically with the nesting level.
2. **Abort Behaviour:** NestedBTO produced more predictable and lower abort rates than NesTM, primarily due to its deterministic timestamp ordering strategy. At a nesting depth of 3 with 64 threads, abort rates for NestedBTO were 6%, 14%, and 19% for levels 1, 2, and 3, respectively. NesTM showed greater abort rates of 8%, 15%, and 22% for the same levels. The differences arise from the different design decisions: NestedBTO avoids unnecessary aborts by early conflict resolution, whereas NesTM's locking and validation procedures might lead to cascading aborts amid a dispute.
3. **Latency Feature:** NestedBTO was superior in terms of latency performance, especially in terms of high-tail-latency reduction. At the 50th percentile, NestedBTO had a transaction latency of about 29.1 ms; NesTM had 34.6 ms. At the 99th percentile, the latencies were 42.3 and 53.6 ms for NestedBTO and NesTM, respectively. NestedBTO's non-blocking validation, which prevents commit-phase bottlenecks, is the fundamental reason for this improvement. On the other hand, NesTM's locking during the commit phase introduces contention, especially during tree rebalancing, which led to observed latency variance of 15%, compared to only 8% for NestedBTO.

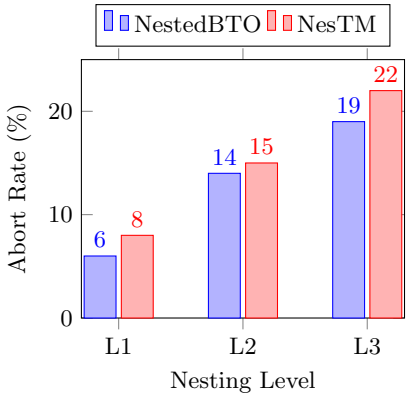
The experimental results demonstrated two advantages of NestedBTO. First, NestedBTO achieves a reasonably good throughput, even with deep nesting levels, with a capacity throughput of about 25% at nesting level 5 compared to a high degradation of up to 38% with NesTM. Second, NestedBTO's conflict



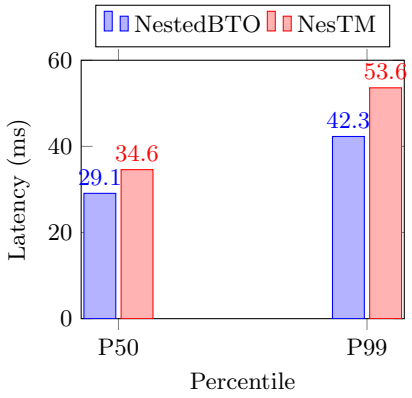
STAMP Throughput Comparison



(a) RBTree Throughput



(b) RBTree Abort Rate per Level



(c) RBTree Latency Distribution

Fig. 3: Performance comparison of NestedBTO and NesTM across STAMP and Red-Black Tree benchmarks. (a)–(c) illustrate throughput, abort rates, and tail latencies in red-black tree workloads.

resolution reduces cascading aborts by 20–30% for heavily nested transactions. However, there are two points to keep in mind: First, with a nesting level beyond five, the timestamp comparisons have overhead issues, and there will be a runtime penalty of around 12%. Second, for shallow nesting depths (levels 1–2), NesTM at times outperformed NestedBTO up to 5%-due mainly to its lock granularity being more suitable for hardware.

5 Conclusion & Future work

The paper presents **NestedBTO**, which is considered a new STM protocol for closed nesting. Under the opacity consistency model, the algorithm permits parallel execution of nested transactions while guaranteeing correctness. **NestedBTO** is demonstrated to have major performance advantages over NesTM via extensive benchmarking with STAMP, concurrent Red Black Tree, and concurrent hashtable. One possible direction is to extend **NestedBTO** to allow multi-versioned transactional objects. The extension would bring about more concurrency by permitting object versions to coexist, hence allowing more flexible and efficient transactional operations. One can also delve deeper into optimizations for specific application scenarios and do further benchmarking on various workloads to gather further insights into the performance characteristics of **NestedBTO**. On the other hand, pondering integration with emerging hardware architectures and tapping into possible distributed implementations make for interesting areas for future work. Moreover, investigating the integration of **NestedBTO** with emerging hardware architectures and exploring its applicability in distributed systems are intriguing areas for future exploration. Overall, the versatility and robustness of **NestedBTO** lay a solid foundation for continued research and innovation in transactional memory systems.

References

1. Assa, G., Meir, H., Golan-Gueta, G., Keidar, I., Spiegelman, A.: Using nesting to push the limits of transactional data structure libraries. arXiv preprint arXiv:2001.00363 (2020)
2. Baek, W., Bronson, N., Kozyrakis, C., Olukotun, K.: Implementing and evaluating nested parallel transactions in software transactional memory. In: Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures. pp. 253–262 (2010)
3. Barreto, J., Dragojević, A., Ferreira, P., Guerraoui, R., Kapalka, M.: Leveraging parallel nesting in transactional memory. ACM Sigplan Notices **45**(5), 91–100 (2010)
4. Bernstein, P.A., Goodman, N.: Multiversion concurrency control—theory and algorithms. ACM Transactions on Database Systems (TODS) **8**(4), 465–483 (1983)
5. Dalessandro, L., Spear, M.F., Scott, M.L.: Norec: streamlining stm by abolishing ownership records. ACM Sigplan Notices **45**(5), 67–78 (2010)
6. Dickerson, T.D., Gazzillo, P., Herlihy, M., Koskinen, E.: Proust: A design space for highly-concurrent transactional data structures. arXiv preprint arXiv:1702.04866 (2017)

7. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Towards formally specifying and verifying transactional memory. Formal Aspects of Computing **25**, 769–799 (2013)
8. Guerraoui, R., Kapalka, M.: Opacity: A correctness condition for transactional memory. Tech. rep. (2007)
9. Guerraoui, R., Kapalka, M.: On the correctness of transactional memory. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming. pp. 175–184 (2008)
10. Haines, N., Kindred, D., Morrisett, J.G., Nettles, S.M., Wing, J.M.: Composing first-class transactions. ACM Transactions on Programming Languages and Systems (TOPLAS) **16**(6), 1719–1736 (1994)
11. Harris, T., Marlow, S., Peyton-Jones, S., Herlihy, M.: Composable memory transactions. In: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming. pp. 48–60 (2005)
12. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: Proceedings of the 20th annual international symposium on Computer architecture. pp. 289–300 (1993)
13. Kobus, T., Kokocinski, M., Wojciechowski, P.T.: The correctness criterion for deferred update replication. Program of TRANSACT **15** (2015)
14. Kumar, R., Vidyasankar, K.: Hparstm: A hierarchy-based stm protocol for supporting nested parallelism. In: the 6th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT'11) (2011)
15. Lebanoff, L., Peterson, C., Dechev, D.: Check-wait-pounce: Increasing transactional data structure throughput by delaying transactions. In: Distributed Applications and Interoperable Systems: 19th IFIP WG 6.1 International Conference, DAIS 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Kongens Lyngby, Denmark, June 17–21, 2019, Proceedings 19. pp. 19–35. Springer (2019)
16. Moravan, M.J., Bobba, J., Moore, K.E., Yen, L., Hill, M.D., Liblit, B., Swift, M.M., Wood, D.A.: Supporting nested transactional memory in logtm. ACM SIGARCH Computer Architecture News **34**(5), 359–370 (2006)
17. Moss, J.E.B., Hosking, A.L.: Nested transactional memory: Model and architecture sketches. Science of Computer Programming **63**(2), 186–201 (2006)
18. Peri, S., Vidyasankar, K.: Correctness of concurrent executions of closed nested transactions in transactional memory systems. Theoretical Computer Science **496**, 125–153 (2013)
19. Peri, S., Vidyasankar, K.: Correctness of concurrent executions of closed nested transactions in transactional memory systems. In: Distributed Computing and Networking: 12th International Conference, ICDCN 2011, Bangalore, India, January 2-5, 2011. Proceedings 12. pp. 95–106. Springer (2011)
20. Ramadan, H., Witchel, E.: The xfork in the road to coordinated sibling transactions. In: 4th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT 2009) (2009)
21. Ranjan, N., Kapoor, R., Peri, S.: Short paper: An efficient framework for supporting nested transaction in stms. In: International Conference on Networked Systems. pp. 204–210. Springer (2024)
22. Saha, B., Adl-Tabatabai, A.R., Hudson, R.L., Minh, C.C., Hertzberg, B.: Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming. pp. 187–197 (2006)

23. Shavit, N., Touitou, D.: Software transactional memory. In: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing. pp. 204–213 (1995)
24. Spiegelman, A., Golan-Gueta, G., Keidar, I.: Transactional data structure libraries. ACM SIGPLAN Notices **51**(6), 682–696 (2016)
25. Volos, H., Welc, A., Adl-Tabatabai, A.R., Shpeisman, T., Tian, X., Narayanaswamy, R.: Nepaltm: design and implementation of nested parallelism for transactional memory systems. In: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming. pp. 291–292 (2009)

Appendix

A Correctness Proof:

A.1 CP-CNO correctness criteria

A schedule S is classified as belonging to the **CP-CNO** class if there exists a serial schedule SS meeting the following criteria:

1. *Event Equivalence:* The operations performed in S and SS are identical.
2. *Schedule Partial order Equivalence:* If two nodes nY and nz are peers in the transaction tree represented by S, with nY occurring before nZ in S, then the same order is maintained in SS.
3. *OptConf Equivalence:* If two memory operations in S are in optConf relation, indicating that they could have executed in either order without affecting correctness, then the same relationship holds in SS.

In the CP-CNO context, the transaction set $dset(T_i)$ encompasses T_i itself, all its descendants, and all write operations conducted by both T_i and its descendants. Additionally, any transaction whose read operation does not originate from its local buffer is designated as the *external read* operation for that particular transaction.
The *Optconf* relation is established between write operations present in the *dset* of T_i and the external read operation of T_j, capturing all three types of conflicts between transactions:

1. *w-r* optconf: If $p_i(x)$ is the write operation in T_i's *dset* and $q_j(x)$ is the external read of T_j.

2. *r-w* optconf: If $p_i(x)$ is the external read of T_i and $q_i(x)$ is the write operation in T_j's *dset*.

3. *w-w* optconf: If $p_i(x)$ is the write operation in T_i's *dset* and $q_j(x)$ is the write operation in T_j's *dset*.

OptGraph, a polynomial-time algorithm, is devised to efficiently determine membership in the CP-CNO class. This algorithm constructs a conflict graph based on the Optconf relations, facilitating the subsequent acyclicity check. The optGraph construction proceeds as follows:

1. Vertices: Each node in the transaction tree, including those representing aborted transactions, is represented as a vertex in the graph. Nodes in the transaction tree encapsulate both transactions and memory operations.

2. Edges: Each transaction t_X starting from t_R is considered. For every pair of children n_P and n_Q (excluding t_{init} and t_f) in $S.children(t_X)$, edges are added from vertex v_P to v_Q as follows:
1. Completion edges: If n_P precedes n_Q in the schedule S.
2. Conflict edges: For any two memory operations m_Y and m_Z, where m_Y is in the $dSet$ of n_P and m_Z is in the $dSet$ of n_Q, an edge is added if $S.isOptConf(m_Y, m_Z)$ evaluates to true.

Theorem. *The history generated by the NestedBTO algorithm satisfies the* ***Conflict preserving close nested opacity(CP-CNO)****[19]*

Proof: Nested BTO incorporates two kinds of edges during schedule generation: conflict edges and timestamp order edges. Timestamp order edges preserve the schedule's partial order equivalence, ensuring that if one node precedes another in real time, it also precedes it in the timestamp order. The operations in the resultant serial schedule generated by Nested BTO match those given as input. Therefore, Nested BTO fulfills the initial two conditions of CP-CNO.
Consider two transactions, t_i and t_j, where $timestamp(t_i) < timestamp(t_j)$, establishing a timestamp ordering edge between t_i and t_j. Now, let us explore the following scenarios:

1. If $t_j.dset(w(x))$ executes before $t_i.extread(x)$, they are in a tryC-read conflict. Consequently, our underlying algorithm aborts t_i because $max - w(x)$ contains the identifier of a later transaction. 2. If $t_j.extread(x)$ executes before $t_i.dset(w(x))$ in the schedule, indicating read-TryC conflicts, t_i is aborted because $max - r(x)$ contains the identifier of a later transaction. 3. If $t_j.dset(w(x))$ executes before $t_i.dset(w(x))$ in the schedule, indicating Try-TryC conflicts, t_i is aborted because $max - w(x)$ contains the identifier of a later transaction.

In all three cases, the graph produced by the Optgraph remains acyclic. Consequently, the resulting schedule produced by our NestedBTO algorithm satisfies the CP-CNO correctness condition for closed nested transactions.

B Analysis:

1. **Yada in STAMP:** Because of its effective timestamp-based conflict detection, NestedBTO outperforms NesTM by 6.1% in this benchmark's normalized performance. While NesTM's commit-lock mechanism creates bottlenecks under high thread concurrency, NestedBTO's lack of global locking reduces contention during frequent sub-transaction retries.
2. **Hashtable Benchmark Evaluation: NestedBTO vs. NesTM**
   Using a concurrent hashtable modeled after STAMP, set up with 4096 buckets and a workload that is primarily read (12.5% inserts, 87.5% lookups), we compare NestedBTO to NesTM. With an emphasis on throughput, abort rates, tail latency, and memory overhead, performance is evaluated by adjusting thread counts (8–64), nesting depths (2–5), and transaction sizes (4–32 operations).
   (a) *Throughput:* Because of its lock-free, timestamp-based commit protocol, NestedBTO routinely outperforms NesTM, reaching up to 1.29× higher throughput at 64 threads. As nesting depth increases, NestedBTO shows linear degradation, whereas NesTM experiences a steeper quadratic drop.
   (b) *Abort rates:* : NestedBTO dramatically lowers aborts, with a 9% rate compared to 21% in NesTM at high contention. This improvement is due to deterministic timestamp ordering against NesTM's lock-induced stalls.

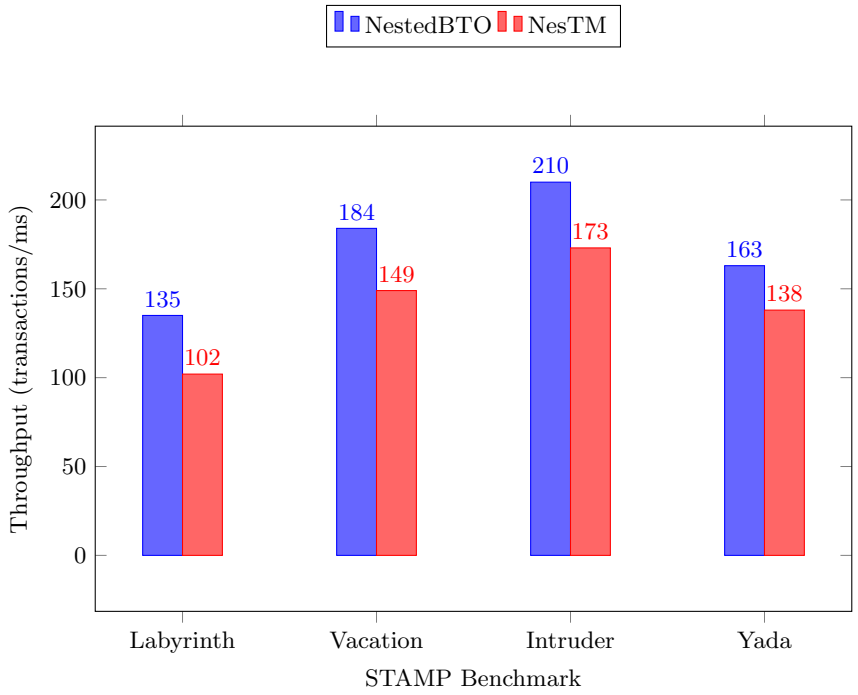


Fig. 4: Throughput Comparison of NestedBTO vs. NesTM on STAMP Benchmarks

(c) *Latency:* The latencies are 30-40% lower in NestedBTO across nesting depth. Significant latency spikes are caused by NesTM's nested locking, particularly when the hashtable is being resized.

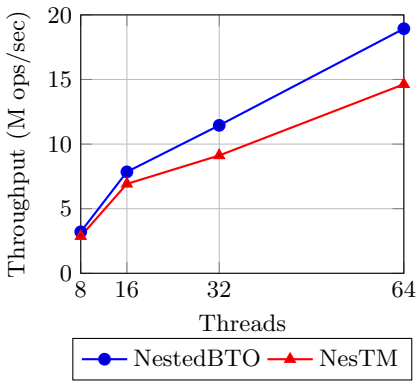


Fig. 5: Throughput comparison

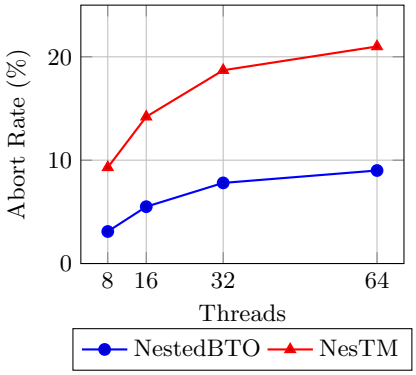


Fig. 6: Abort rate comparison

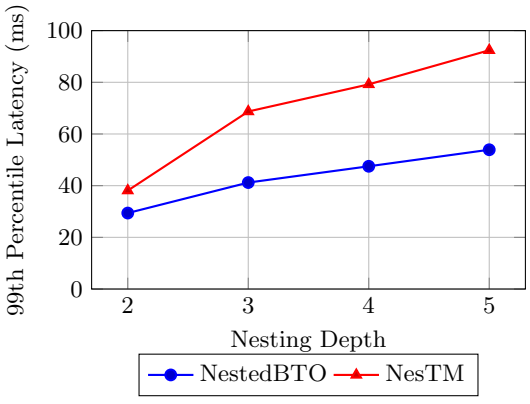


Fig. 7: Tail latency vs nesting depth

NestedBTO Protocol (continued)

Operation traverse_up$_i$(x)
create a transactional stack *tstack*
$t_j \leftarrow t_i.parent$
tstack.push(t_i)
while t_j is not NULL **do**
  if x is present in t_j.writelist **then**
    lock obj(x)
    if $t_j.writelist[x].max\_r > t_i.id$ **then**
      unlock obj(x)
      $t_k \leftarrow tstack.top()$
      Rec_abort(t_k)
      return **FAILURE**
    end if
    $val \leftarrow t_j.writelist[x].val$
    set $t_j.writelist[x].max\_r$ *to higher timestamp*
    break
  else if x is present in t_j.readlist **then**
    lock obj(x)
    $val \leftarrow t_j.readlist[x].val$
    set $t_j.writelist[x].max\_r$ *to* $t_i.id$
    unlock obj(x)
    break
  else
    tstack.push(t_j)
    $t_j \leftarrow t_j.parent$
  end if
end while
traverse_down(tstack,val,$t_i.id$*,x)*
return *val*
Operation traverse_down (tstack,val,t$_i$.id,x)
while tstack is not NULL **do**
  $t_k \leftarrow tstack.top()$
  if x is present in the readlist of t_k **then**
    lock *obj(x)*
    update $t_k.readlist[x].val \leftarrow x.val$
    update $t_k.readlist[x].max\_r \leftarrow t_i.id$
    unlock *obj(x)*
  else
    create t-obj x in the readlist
    $x.val \leftarrow val$
    $x.max\_r \leftarrow t_i.id$
  end if*tstack.pop()*
end while
Operation Try_Abort$_i$()
if $t_i.status \leftarrow$ **ABORT then**
  return
end if
lock $trans_i$
$t_i.status \leftarrow$ **ABORT**
$t_i.parent.termcount{+}{+}$
unlock $trans_i$
delete t_i
return
Operation Rec_abort$_i$()
if $t_i.status \leftarrow$ **ABORT then**
  return
end if
for all k in $t_i.child\_list()$ **do**
  Rec_abort(k)
end for
$t_i.status \leftarrow$ **ABORT**
$t_i.parent.termcount{+}{+}$
return