A Coordinated Approach to Transactional Data Structures

Anonymous Author(s)

ABSTRACT

Transactional data structures (TDS) enable programmers to compose concurrent data structure operations into isolated transactions. TDS systems that seek generality typically detect conflicts by observing how transactions access the physical structure of the TDSs. Systems that seek to avoid false conflicts typically focus on abstract descriptions of operations and their commutativity conditions. No prior work has found a way to effectively unify the approaches.

In this paper, we introduce Harmony, a novel TDS system that bridges this divide by embedding semantic metadata in the nodes of a data structure. This enables TDS designers to *physically* materialize *semantic* conflicts. Harmony leverages fast multi-word atomic primitives to manage this metadata concurrently with the underlying data structure. It also employs a novel use of coroutines to efficiently defer structural modifications (such as safely unlinking nodes) until transaction commit or abort. The resulting system achieves a high degree of generality and best-in-class performance.

We demonstrate generality by implementing a broad range of data structures, including skip lists, resizable hash tables, arrays, and deques. We also show that these data structures support complex operations, like mutating and read-only range queries and scans. In evaluation across microbenchmarks and TPC-C workloads, Harmony consistently matches or outperforms the state of the art, demonstrating high resilience to contention and low overhead.

1 INTRODUCTION

Transactional data structures (TDS) are a special class of concurrent data structures (CDS) that support *composition*. That is, a programmer is able to create transactions that consist of invocations of several TDS methods. At the end of a transaction, its entire sequence of invocations will either appear to take effect all at once, without any interference from other threads' transactions, or none of those invocations will appear to have taken effect. This design is appealing because it promises better performance than general constructions like transactional memory (TM) [23, 41], without introducing significant challenges to programmer productivity.

TDS systems are an important research direction because modern parallel workloads increasingly rely on composing complex data structure operations. Examples include managing concurrent metadata for shared LLM K/V caches, hybrid OLTP/OLAP databases that modify a data store along with a set of indices, and processing a set of smart contracts within a blockchain. However, to date, TDS systems have not been robust enough to be applied in these domains. This is primarily due to a lack of generality. Some prior works are unable to handle arbitrary data types. Others are only able to handle set and map data structures, not queues, stacks, vectors, and heaps. Most significantly, we are not aware of prior TDS systems that support range queries and scan operations.

The main focus of prior TDS work has been on efficiently detecting conflicts among transactions. Suppose transaction X_A issued by thread T_A performs an operation O_D^A on data structure D. If that transaction is concurent with some operation $O_D'^B$ by some thread $B \neq A$, and O_D^A and $O_D'^B$ do not commute, then at least one of X_A and X_B must not be allowed to complete, and its effects on D must not be visible to any other transaction. This necessarily involves communication through shared memory, with two main approaches. In a line of research originating with *boosting* [20], each TDS consists of a CDS [21] and an out-of-band conflict table. The conflict table allows expressing operation-level *semantic* conflicts. For example, with boosting, a conflict between `remove(7)` and `lookup(7)` would occur on some abstract lock in the table, ensuring that at least one of those operations would become aware of the other. More recent systems that use semantic conflicts include Proust [9] and Conflict Abstractions [8].

The alternative, which originated with the Transactional Data Structure Library (TDSL) [44], is to use low-level structural information to express conflicts. This is more reminiscent of classic CDS designs. For example, in TDSL, if there was a list containing $4 \rightarrow 8$, then `lookup(7)` would record the fact that the successor pointer for the node holding `4` was the node holding `8`, and a conflict with `insert(7)` would occur on the pointer (or its associated synchronization metadata). Other TDS systems that use structural information include STO [25], LFTT [48], and Medley [6].

We observe that both conflict detection techniques have advantages. Semantic metadata enables precise conflict detection without false aborts. It allows a transaction to insert an element into a hash table while another transaction is resizing it, since these operations logically commute. Similarly, structural conflict detection enables complex operations and data types. For example, the fact that `2.718` $\rightarrow$ `3.14` concisely expresses that the infinite sequence of numbers between those values is not present in the data structure.

In this paper, we introduce a new TDS system called Harmony. Harmony employs a hybrid approach to conflict detection, which leverages the best ideas of the Semantic and Structural approaches. In Harmony, data structure designers embed a lightweight *semantic* metadata record (`HMeta`) directly into the nodes of a data structure. This metadata serves as a physical manifestation of a transaction's semantic footprint. It provides programmers with an efficient and precise mechanism for materializing the correctness guarantees of Conflict Abstractions in order to detect commutativity violations among concurrent transactions. Programmers control the granularity of this metadata to express conflicts

PL'18, January 01–03, 2018, New York, NY, USA
2018.

precisely, while also capturing the semantics of challenging data structures like deques, vectors and heaps, as well as complex operations like range queries and scan operations.

Safely updating a node's physical state along with these `HMeta` fields presents a synchronization challenge. Historically, CDSs relied on complex uses of a single compare-and-swap to achieve linearizability, making such updates difficult. Harmony leverages the fact that software simulations of multi-word atomic primitives have recently become fast and practical. By utilizing these primitives, Harmony is able to integrate and update semantic metadata atomically with the underlying structure during each CDS operation.

Harmony employs eager semantic conflict detection, but defers structural changes until commit or abort. Boosting introduced the idea of high-level *compensating actions* (e.g., remove ↔ insert) to run at cleanup, while structural approaches require the TDS system to mediate all data accesses via custom primitive types. Neither approach is a good fit for Harmony: high-level compensating actions fail to exploit efficiencies afforded by low-level semantic metadata, and custom types limit generality. Instead, Harmony implements each TDS operation as a coroutine that `yields` its result during transaction execution. Upon commit or abort, these coroutines are `resumed`. They then use their preserved local state and `HMeta` to perform precise, low-level cleanup (e.g., unlinking nodes). By relying on local stack frames rather than global undo logs or custom types, Harmony's deferred cleanup remains robust, efficient, and completely agnostic to the underlying program data.

In experimental evaluation, we show that Harmony matches or outperforms the state of the art across microbenchmarks and TPC-C workloads. It is also resilient to high levels of contention, and capable of executing workloads that prior systems could not (e.g., those with range queries).

2 RELATED WORK

Hardware and software TM were originally proposed as techniques for implementing CDSs [23, 41]. As STM became increasingly practical [13, 16, 22], attention turned to using it to compose multiple operations into a single coarse-grained transaction [17]. However, experiences using TM in benchmarks and real-world code indicated that TM, in general, was too conservative, treating every physical memory conflict as a true logical conflict [26, 31, 38, 46, 49].

Transactional Boosting [20] introduced semantic conflict detection. It manages transactional metadata (abstract locks) separate from underlying "black box" data structures. Boosting requires a semantic undo mechanism in case of abort (e.g., `remove`↔`insert`). Proust [9] extends boosting with a lazy mechanism where operations are executed on shadow copies of an object, and replayed on the shared copy at commit time. This helps to support data structures with tricky semantics, e.g., priority queues, but the reliance on high-level compensating actions increases overhead on abort.

TDS systems that use structural metadata build upon attempts to inject the notion of semantic conflicts into the TM frameworks themselves [22, 32, 37]. For example, early release [22] allows a transaction to remove memory locations from its read-set if it knows that those locations will no longer raise semantic conflicts. The Transactional Data Structure Library (TDSL) [44] greatly expands upon this idea. For example, its ordered map superimposes Fraser's skip list [13] with a TL2 [7] STM-based, early release-optimized singly linked list. Optimistic Transactional Boosting [18] is a similar technique, which integrates transactional metadata into "white box" data structures. Elastic transactions [12] took a different approach to normalizing and regularizing when entries enter a transaction's read set, so that an STM could avoid conflicts and support greater composability. Software Transactional Objects (STO) [25] defines the data types that a low-level transaction can use (e.g., `TBox<int>`), and then introduces predicates and other means for expressing semantic conflicts (or the lack thereof). For example, one STO type enhances an earlier idea from Ruan et al. [37] to allow increments and decrements of a shared counter to commute with each other, but not with reads of the counter value. The advantages of the structural technique come at the cost of tight coupling between the TDS system and program data, and high overhead for structural operations like resizing a hash table or rebalancing a tree.

The most recent TDS systems have focused on progress and persistence rather than generality. The Lock-Free Transactional Transform (LFTT) [48] carefully defines the semantics of operations, so that transactions can *publish* in-progress actions to facilitate helping. OneFile [36] translates TDSs to non-volatile byte-addressable memory. Most recently, Medley [6] achieved nonblocking progress and persistence by superimposing multi-word compare-and-swap [16] atop the linearization points [24] of nonblocking data structures.

An orthogonal line of research aims at enabling range queries in highly optimized concurrent data structures [1, 29, 33, 47]. However, none of these approaches allows range queries to be executed within a transactional context, and they also do not support bulk updates.

3 USER-FACING PROGRAMMING MODEL

In TDS systems, there are two distinct programming responsibilities: TDS *users* and TDS *designers*. A user crafts *transactions*, which interleave operations on local state with invocations of methods of the TDSs. To the user, each TDS is as a black box. However, unlike sequential data structures, TDS operations might *abort*, indicating that the method was unable to complete due to a conflict with a concurrent transaction.

Furthermore, it is possible that an in-flight transaction becomes logically invalid during execution because of concurrent updates to data it previously read. Early TDS systems provided strict opacity [14] to handle this transparently, ensuring transactions automatically aborted and retried if they would encounter inconsistent state. Cai et al. [6] questioned the necessity of strict opacity, observing that in most cases,

```
let self = create_harmony_transaction_context()
while true:
  self.begin()
  # after each operation, unwrap to detect WAW/RAW conflicts
  let res1 = self.unwrap(Tds1.Op1(self, ...))
  if res1 == abort: self.rollback(); continue
  let res2 = self.unwrap(Tds2.Op2(self, ...))
  if res2 == abort: self.rollback(); continue
  ...
  # example of manually checking for WAR conflicts
  if !self.validate():
    self.rollback(); continue
  ...
  let resN = self.unwrap(TdsN.OpN(self, ...))
  if resN == abort: self.rollback(); continue
  if self.try_commit(): break
```

Figure 1: An example of the user-facing API

```
# Low-level semantic metadata.  S is the data structure's
# synchronization mechanism
class HMeta<S>:
  # Fields related to semantic ownership
  owner: field<S, &HTx>
  semantic: field<S, u64>
  structural: field<S, u64>
  # Fields related to efficient cleanup
  num_coros: field<S, u32>
  removing: field<S, bool>
  undo_logged: field<S, bool>

# An optional global table of HMeta.semantic fields
# notexist_size is user-configurable
global notexist<S>: field<S, u64>[notexist_size]
```

Figure 2: Per-node metadata and global variables.

transactions can proceed to their commit point without continuous, implicit validation. Harmony adopts this model: When an operation's control flow requires strict consistency, the programmer can simply insert a manual validation check *between TDS method invocations*. A consequence of this design choice is that the TDS user becomes responsible for retrying transactions that abort.

The resulting API for the TDS user consists of five methods, whose use is depicted in Figure 1:

`begin()` initializes a transaction's per-thread state.

`unwrap()` evaluates the result of a TDS operation to decide if there is a valid return value, or if the transaction should abort. It also manages the underlying coroutine continuation.

`validate()` verifies that the reads of prior TDS operations remain consistent.

`rollback()` undoes the intermediate effects of the transaction and prepares for re-execution.

`try_commit()` finalizes the transaction's structural effects if and only if the transaction remains logically valid.

4 LOW-LEVEL SEMANTIC CONFLICT DETECTION

In Harmony, TDS designers embed semantic conflict detection metadata (`HMeta`) directly into data structures. This metadata allows programmers to express semantic conflicts through physical state. The `HMeta` object is presented in Figure 2. In this section, we first discuss the high-level correctness criteria that programmers achieve with this metadata. We then discuss how this low-level semantic metadata is used at the TDS and transaction levels.

4.1 Correctness Criteria

The correctness of Harmony TDSs stems from how programmers map a TDS's use of `HMeta` objects to the commutativity conditions defined by Conflict Abstractions [8]. We define $O = \{O_0 \dots O_n\}$ as the set of operations supported by the transactional data structures in a program. Each operation takes a single argument drawn from $A = \{A_0 \dots A_\infty\}$, a universe of compound types. Thus $O_i(a)$ represents an execution of O_i with argument $a \in A$. We further say that $O_i(a)$ returns a single value $r \in A$. The distinguished return value `conflict` indicates that an operation could not complete due to a commutativity violation. Commutativity violations can also be detected at any validation point, and at commit time.

When $O_i(a)$ reads and writes `HMeta` objects, it populates two abstract sets:

- $O_i^r(a)$ references `HMeta` fields read by $O_i(a)$
- $O_i^w(a)$ references `HMeta` fields written by $O_i(a)$

Given any two concurrent operations $O_i(a)$ and $O_j(a')$, the programmer must ensure that if $O_i(a)$ and $O_j(a')$ *do not commute*, and they are performed concurrently by transactions from different threads, then one of the following holds:

- $O_i^r(a) \bigcap O_j^w(a') \neq \emptyset$ (write-read conflict)
- $O_i^w(a) \bigcap O_j^r(a') \neq \emptyset$ (read-write conflict)
- $O_i^w(a) \bigcap O_j^w(a') \neq \emptyset$ (write-write conflict)

Note that $O_i(a)$ and $O_j(a')$ need not access and record the `HMeta` fields associated with every node that was accessed. They need only access enough fields to construct a sufficient set for conflict detection. As long as concurrent transactions can efficiently detect commutativity violations and employ aborting/cleanup to preserve isolation, the theory of Conflict Abstractions ensures that the resulting execution will guarantee isolation and serializability [10, 34]. We now turn our attention to the details of how Harmony achieves this goal.

4.2 Materializing Conflicts via `HMeta`

Figure 3 presents the per-thread metadata that is tracked by Harmony transactions, along with methods that a TDS designer uses to interact with that metadata. Following the argument by Cai et al. [6] that opacity [14] is unnecessary for individual data structures, a TDS designer does not, in general, worry about the `status` field. The main task is to interact with `HMeta` objects and to track some of those interactions in `HTx.reads` in order to detect commutativity violations. The mode of detection depends on how a conflict unfolds during the execution history.

Eager Detection of WAW and RAW:. If $O_i(a)$ will make changes to the logical state of a TDS, it will first `acquire` the `HMeta` object(s) associated with a by setting their `owner` fields to itself. This suffices to capture all write-after-write conflicts at the time that the second transaction attempts its write. In the common case, $O_i(a)$ will also call `inc_semantic` on those `HMeta` objects.

```
1  # Per-transaction state
2  class HTx:
3    status: Inactive | Active | Aborted
4    reads:  map<field_base<u64>*, u64>
5    coros:  vector<handle>
6    smr:    safe_memory_reclaimer
7  func log_semantic(self: &HTx, m: &HMeta<S>) -> bool
8    atomic(S):
9      let new_v = m.semantic;
10   let old_v = self.reads.get(&m.semantic)
11   if old_v: return old_v == new_v
12   self.reads.set(&m.semantic, new_v); return true
13 func inc_semantic(self: &HTx, m: &HMeta<S>) -> void
14   atomic(S):
15     ++m.semantic
16   let &old = self.reads.find(&m.semantic); if old: ++old.val
17 func log_struct(self: &HTx, m: &HMeta<S>) -> bool
18   # log_semantic(8-12), but s/m.semantic/m.structural/
19 func inc_struct(self: &HTx, m: &HMeta<S>) -> void
20   # inc_semantic(14-16), but s/m.semantic/m.structural/
21 func log_notexist<S>(self: &HTx, ki: u64) -> bool
22   let idx = hash(ki) % notexist_size
23   # log_semantic(8-12), but s/m.semantic/notexist[idx]/
24 func inc_notexist<S>(self: &HTx, ki: u64) -> void
25   let idx = hash(ki) % notexist_size
26   # inc_semantic(14-16), but s/m.semantic/notexist[idx]/
27 func check_owned<S>(self: &HTx, m: &HMeta<S>) -> bool
28   atomic(S): return m.owner == None || m.owner == self
29 func acquire<S>(self: &HTx, m: &HMeta<S>) -> bool
30   atomic(S):
31     if m.owner == None: m.owner = self; return true
32     else: return m.owner == self
```

Figure 3: Per-thread state and supporting methods

Validation to Detect WAR:. If $O_i(a)$ is dependent on the logical state of the TDS, but does not modify it, it uses `check_owned` to detect read-after-write conflicts. In the common case, it also calls `log_semantic`. The semantic version increases monotonically (`inc_semantic`), and thus read-after-write conflicts can be detected through validation of the `semantic` field's value. This validation can optionally be performed at any time by the user of the TDS. It is also performed before every commit.

4.3 Intra-Transaction Dependencies and Self-Aborts

Because control flows in transactions are dynamic, it is possible for a transaction to write a value after reading it. Doing so must not result in transaction mistakenly concluding that it conflicts with itself. Suppose transaction T_i reads a node (logging its `semantic` version v into O_i^r), and subsequently modifies that same node (incrementing its `semantic` version to $v+1$ and adding it to O_i^w). During the next `validate` phase, the logged version v will no longer match the physical version $v+1$. To resolve this without forcing transactions to search their write sets during every read, Harmony pushes this complexity into the modification phase. When an operation calls `inc_semantic` to increment a node's version, the system checks if the thread's local read set (O_i^r) contains an entry for this node. If it does, the logged version is updated.

Note that `inc_semantic` does not *overwrite* the value in `self.reads`, but increments it (line 15). This is a critical distinction. Note that neither `acquire` nor `inc_semantic` performs lookups in `self.reads`. Furthermore, recall that

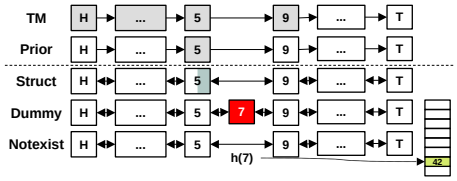


Figure 4: Unsuccessful `list::get(7)`. New techniques appear below the dashed line.

Harmony does not guarantee strict opacity. Thus it is possible that after T_i called `log_semantic`, but before it called `acquire`, some concurrent transaction T_j acquired the `HMeta`, incremented `HMeta.semantic`, and then committed, clearing `HMeta.owner`. In that case, T_i must abort. By incrementing instead of overwriting, T_i avoids self-aborts when there is no concurrent acquisition, while ensuring that validation (either manual or at commit-time) detects true WAR conflicts.

4.4 Materializing the Non-Existence of Data

Mapping abstract conflict sets to physical `HMeta` records is straightforward when operating on existing nodes. However, a transaction may attempt to lookup or remove an element that is not present. Figure 4 presents five different ways to represent that the value 7 is not present in a list where node `n5` contains the value 5 and node `n9` contains the value 9. The first technique, associated with TM, is to include the synchronization metadata for all nodes from the head through `n9`. Should any operation modify any of these nodes, a conflict will occur, causing the `get(7)` operation to detect a conflict on its next validation. Clearly this is an overapproximation.

Prior TDS techniques that used structural conflict detection, such as TDSL and Medley, use the second technique. The key idea is that the only structural information needed to detect a conflict is the successor pointer of `n5`, so the only metadata to log is the synchronization metadata protecting `n5`. This is much more precise than the TM approach, but still prone to false conflicts. In particular, the insertion of any value between `5` and `9` will be treated as a conflict. For sparse sets (e.g., if this list held floating point values), there remains a significant risk of conflicts. Furthermore, since this technique uses the synchronization metadata to detect conflicts, any change to `n5` will cause an abort, including nonconflicting `update` operations.

Below the dashed line we present three mechanisms provided by Harmony for materializing the absence of datum. These allow transactions to safely populate O_i^r and O_i^w while avoiding false positives.

Structural Versioning: The first option is similar to prior work, and involves tracking metadata associated with the pointer from `5` to `9`. The difference in this case is that Harmony can detect conflicts on `HMeta.structural`, not `HMeta.semantic` or synchronization metadata. In a TDS that uses this technique, `get` uses `log_struct` to add the

structural version of `n5` to its `self.reads`, and insertions/deletions perform `inc_struct` on the `HMeta` in `n5`. We note that this technique is particularly valuable in range queries and scans, where `HMeta.structural` records the nonexistence of all elements between `n5` and `n9`.

Dummy Nodes: When a Harmony TDS operation intends to remove a node, it typically sets the `HMeta.removing` flag. It does not physically unlink the node until commit time. We can leverage this mechanism to materialize the absence of data during failed lookups or range queries. In the "own" row of the figure, `get(7)` inserted `n7`, took ownership of it, and set its `HMeta.removing`. The operation must then remove the node upon commit or abort. By inserting a dummy node, the operation transforms a logical read of an absent value into a visible write. There is now an `HMeta` object whose `owner` field is visible to concurrent transactions. This transforms subsequent WAR conflicts (e.g., a concurrent insertion of 7) into WAW conflicts, which are eagerly detected.

`notexist` Table: The third alternative is to use an out-of-band table, `notexist`. Operations hash the missing value to an element in this table and `log_notexist` the element. Insertions must `inc_notexist` the same element. For hash function H and key k, this technique allows readers to place `notexist[`$H(k)$`]` into O_i^r, and writers to place it into O_i^w.

5 DEFERRED STRUCTURAL CLEANUP VIA COROUTINES

In Section 4.4, we briefly mentioned that Harmony encourages logical deletion of nodes, in which case the unlinking of a node is deferred until commit time. There are many other structural actions that might require undoing on abort, or completion upon commit. Upon abort, an inserted node may need to be removed. In TDS systems that explicitly manage program data, automatic structural cleanup can lead to performance anomalies and correctness challenges. For example, if an insertion triggers a hash table resize, and then the calling transaction aborts, the resize should not be undone. In a sophisticated TDS system, if a node is inserted into a red/black tree and then, before the inserter aborts, concurrent operations change the node's position and commit, then the steps to unlink the node will not be symmetric with the steps that inserted it. Harmony takes a novel approach to structural cleanup: it delegates this responsibility to the TDS designer, who can leverage low-level data structure knowledge to achieve efficiency.

A key enabler of this approach is the assumption that each TDS uses efficient multi-word atomics for synchronization. This allows designers of graph data structures (e.g., lists, trees) to safely and atomically maintain bidirectional edges (e.g., parent or predecessor pointers). By guaranteeing upward/backward traversability, a cleanup operation can dynamically compute its local neighborhood. Consequently, structural cleanup requires only two inputs: the local node and the transaction's commit/abort status. This eliminates the need to restart traversals from the root or rely on potentially outdated path logs.

```
class counter<S>:
  m: HMeta<S>
  c: field<S, int>
func read<S>(c: counter<S>, self: HTx) -> handle<int|abort>
  atomic(S):
    # If owned, fail
    if c.m.owner != None && c.m.owner != self
      self.status = Aborted; co_return abort
    # Be sure to log the semantic version!
    self.log_semantic(c.m)
    co_return c.c
func inc<S>(c: counter<S>, self: HTx) -> handle<int|abort>
  let seq = 0, old = None
  atomic(S):
    # Take ownership or fail
    if !c.m.owner:
      c.m.owner = self
      self.inc_semantic(c.m)
    if c.m.owner != self:
      self.status = Aborted; co_return abort
    # Undo log locally if self's first call to inc()
    if !c.m.undo_logged:
      old = c.c; c.m.undo_logged = true
    seq = ++c.m.num_coros
    co_yield ++c.c
  # Runs when the handle is resumed
  if self.status == Aborted:
    atomic(S):
      # reset value
      if old:
        c.c = old; c.m.undo_logged = false
      # structural cleanup would happen here
      # release semantic ownership
      if seq == 1:
        c.m.num_coros = 0; c.m.owner = None
  else:
    atomic(S):
      if seq == c.m.num_coros:
        # structural cleanup would happen here
        c.m.num_coros = 0; c.m.undo_logged = false
        c.m.owner = None
  co_return void
```

Figure 5: Cleaning up with coroutines

5.1 Coroutines as Structural Continuations

Harmony models each TDS operation as a coroutine. Operations that do not modify `HMeta` objects `co_return` their result. Operations that take ownership, increment fields of `HMeta`, or set `removing` `co_yield` their result in a resumable execution handle. The `HTx.unwrap` method places these handles into `HTx.coros`. The suspended coroutine frame implicitly captures the exact local state (pointers, traversal history, and local variables) needed to complete the physical structural change later. `HTx` requires no custom undo or redo logging structures, nor does it require any knowledge of the program data that was modified by an operation. It simply maintains an ordered list of lightweight coroutine handles. Upon commit, these handles are resumed in forward order, and on abort, they are resumed in reverse order. Figure 5 shows a complete implementation of a counter to illustrate ownership, interaction with `HMeta.semantic`, and coroutines.

5.2 Coordinating Intra-Transaction Continuations

Because a single transaction might execute multiple operations on the same node (e.g., an `insert` followed by an

update), Harmony must ensure that deferred physical cleanup is not duplicated. For example, a node must only be unlinked once, and its HMeta ownership released exactly once, or else there is a risk of "flicker" and races. Harmony coordinates these deferred operations using two fields in HMeta, which are evaluated locally by the coroutines upon resumption.

Releasing Ownership: Any operation that calls co_yield will increment HMeta.num_coros before doing so, and log the result to the coroutine stack frame. Upon commit, each coroutine is resumed in order. Using the logged value, each coroutine can determine if it was the last to run, in which case it assumes responsibility for releasing ownership. Symmetrically, upon abort, coroutines are resumed in reverse order, and thus the first coroutine to run on each node will assume responsibility for releasing ownership. In this manner, any cleanup will be performed *before* releasing ownership.

Logging Program Data: The HMeta.undo_logged field is used for logging and restoring data. A transaction's first operation that modifies a node's underlying program data will undo log the object to the coroutine stack frame and set HMeta.undo_logged. This may not be the first writing operation (e.g., if a key/value pair is removed and then reinserted from a map, remove simply takes ownership and sets HMeta.removing; the later call to insert clears HMeta.removing and must undo log the value before overwriting it). This approach avoids redundant logging (e.g., for a sequence of updates, only the first will produce an undo log). When a transaction aborts, coroutines are executed in reverse order, ensuring that the logged data is restored before the last coroutine on a node releases ownership.

Structural Finalization: The num_coros field also facilitates structural finalization. Each coroutine can track the specific changes that it made to a data structure, but lacks knowledge of the other coroutines run by the enclosing transaction. Since num_coros allows a coroutine to know if it was *last* (in the case of commits) or *first* (in the case of aborts), the coroutine has just enough information to know what structural changes it should perform, if any. This allows Harmony to guarantee that only the newest (or oldest) coroutine executes the physical structural change or data restoration, ensuring that nodes are never prematurely unlinked.

5.3 An Example

Figure 5 presents a complete example for a counter object. Since the read operation does not modify the counter, it uses co_return (line 11) to return without generating a continuation, even when it does not detect a conflict. The remaining instructions of the method manage ownership and logging. A key takeaway is that coroutines do not add any programmer burden for read-only operations.

The inc method does modify the counter, and must manage ownership, logging, and deferred execution. There are four important parts:

Undo Logging (Lines 22-23): If this is the transaction's first coroutine to modify the counter, it captures the original counter value to its local stack frame and sets undo_logged.

Sequencing and Yielding (Lines 24-25): The coroutine increments and logs num_coros to determine its relative age, increments the counter, then co_yields a result.

Recovery after Abort (Lines 30-31): When resuming after an abort, if this specific coroutine logged the original data, it restores the counter's value. (If the counter required structural cleanup, it would occur at line 32). On an abort, lines 34-35 ensure that the *first* coroutine releases ownership only after all data has been safely restored.

Finalization after Commit (Lines 38-41): When resuming after a commit, if the coroutine was *last* to access this node, it finalizes structural changes and releases ownership.

6 SYSTEM CONSIDERATIONS

Harmony seeks to achieve good performance and generality. One aspect of generality is the ability to allow each TDS to choose its synchronization mechanism. This results in some subtle synchronization and memory safety issues, which we discuss in this section and relate to the implementation of the user-facing Harmony API.

6.1 Multi-Word Synchronization

Our discussion thus far has relied on a parameter S, representing a TDS's synchronization mechanism. As discussed in Section 5, coroutines often require bidirectional graph traversal to manage deferred structural updates efficiently. In addition, interactions with fields of HMeta objects must be *atomic with* their associated TDS. In order to read and update multiple pointers and HMeta fields simultaneously, Harmony mandates that any compatible data structure synchronization mechanism, denoted S, must support in-place atomic multi-word updates. Several modern synchronization frameworks provide this support [2, 15, 43, 45]. In the pseudocode listings, the field<S, &HTx> template indicates that reads and writes of HMeta fields are mediated through an appropriate synchronization mechanism, and atomic(S) is a shorthand to indicate that a lexically-scoped region should be implemented according to the requirements of S.

There is one additional safety concern. The validate routine checks semantic, structural, and notexist fields outside of any TDS operation, and thus without the protection of an S. We address this by requiring the synchronization mechanism to support concurrent *reads* of fields through the C++ atomic_ref template. In most cases, this is already required by S, in order to support optimistic data structure traversal. The only exception we are aware of is when S is an eager STM with undo logging. In this case, accesses through atomic_ref may see a value that is about to be rolled back. Harmony is resilient to this issue: the speculatively read value will never be less than the true value, meaning that false positive conflicts are possible, but never false negatives.

```
1  func begin(self: &HTx) -> void
2    self.status = Active;
3    self.smr.enter()
4  func validate(self: &HTx) -> bool
5    if self.status == Aborted: return false
6    for [addr, val] in self.reads
7      if *addr != val: self.status = Aborted; return false
8    return true
9  func rollback(self: &HTx) -> void
10   self.coros.reverse().each(|c| c.resume())
11   self.status = Inactive;
12   self.smr.exit();
13   self.coros.clear(); self.reads.clear()
14 func try_commit(self: &HTx) -> bool
15   if !self.validate(): self.rollback(); return false
16   self.coros.each(|c| c.resume())
17   self.status = Inactive
18   self.smr.exit()
19   self.coros.clear(); self.reads.clear()
20   return true
21 func unwrap<V>(self: &HTx, ch: handle<V>) -> V
22   if ch.is_yield: self.coros.push(ch)
23   return ch.value()
```

Figure 6: Transaction-level management routines.

6.2 Global Safe Memory Reclamation

Since `S` is likely to use optimistic synchronization, it will typically require a safe memory reclamation (SMR) algorithm that ensures objects are not reclaimed while a TDS operation might still access them [3, 13, 42]. Harmony also directly reads `HMeta.semantic` and `HMeta.structural` fields *outside* of `S`-protected regions. These accesses must also be protected by SMR. We require every TDS's synchronization mechanism `S` to use the same SMR algorithm, and then we enclose an entire transaction attempt in a single SMR scope. This is represented by the `HTx.smr` field. A single SMR scope amortizes the cost of SMR boundary operations, enables the movement of data across TDSs that use different synchronization mechanisms, and ensures the safety of the Harmony optimistic commit protocol. Nodes unlinked in an `atomic(S)` block are transferred to the transaction's `HTx.smr`. The SMR ensures that physical reclamation is deferred globally until no active transaction could possibly hold a reference to the unlinked node's `HMeta`, ensuring safe TDS operations and safe validation for all concurrent transactions.

6.3 Implementing the User API

Figure 6 presents the user-facing portions of the Harmony API. As discussed in Section 3, there are five functions, which correspond to the lifecycle events of a transaction.

`begin` sets the transaction's status and opens an SMR scope. Harmony requires the programmer to call begin on each transaction attempt. Note that the status is private to a thread: there is no remote abort. It is set to `Aborted` in two situations: when a TDS operation detects a WAW or RAW conflict, or when a call to `validate` detects a WAR conflict.

`validate` checks the `semantic`, `structural`, and `notexist` values that were previously logged by the transaction. It can be called by the programmer, as a manual opacity check. It is also called by `try_commit`, as part of its standard optimistic commit protocol [27]. In this case, note that since TDS operations eagerly acquire ownership, as soon as `validate` returns `true` the transaction can commit.

The `rollback` method iterates through the ordered list of coroutine handles (`HTx.coros`) in reverse order, resuming each. These coroutines perform all clean-up, including restoring data, finalizing structural changes, and releasing ownership of `HMeta` fields. The coroutines capture the appropriate synchronization mechanism `S` as part of the continuation, and resume before the call to `smr.exit()`, ensuring proper synchronization and memory safety.

`try_commit` validates and, on success, resumes each coroutine in order. When this final validation fails, rollback is performed automatically, so the programmer can immediately retry the transaction. As with `rollback`, memory safety is ensured since coroutines resume before the call to `smr.exit()`.

Finally, `unwrap` takes a coroutine handle returned by a TDS operation and processes it. When the handle represents a `co_yield`, it indicates that the operation may need to perform cleanup, so the handle is stored in `HTx.coros`. After this step, the TDS operation's return value is extracted and returned. This simplifies cumbersome C++ coroutine syntax.

7 IMPLEMENTING TRANSACTIONAL DATA STRUCTURES

In this section, we briefly sketch the design of several fundamental data structures. The full Harmony implementation, including these and other data structures, is available at **(redacted for double-blind review)**.

7.1 Ordered Maps (List and Skip List)

Several of our data structures are built atop an ordered map, implemented as a doubly linked list (dlist). This dlist uses head and tail sentinel nodes [19], each of which has an `HMeta` object embedded in it. Each node in the list also has its own `HMeta` object, as well as space for a datum. Regarding keys that are not found, the dlist uses all three techniques from Section 4.4. That is, it uses the "Struct" technique for read-only range queries, the "Notexist" approach when an insert/lookup/remove/update does not find the target key, and the "Dummy" and "Struct" techniques for mutating range queries. This means that insertions and removals must increment `HMeta.structural` when modifying a forward link, and insertions must also update the `notexist` table. We also provide a doubly-linked skip list that extends the dlist's nodes with a variable-sized "tower" containing pointers to predecessors and successors.

By virtue of double-linking, the dlist and skip list can use coroutines for efficient clean-up. This defers unlinking a successful removal until after the operation commits. At that point, the resumed coroutine atomically unlinks the node from the dlist (or skiplist). In order to ensure asymptotic performance when a skip list is accessed several times by a single transaction, insertions eagerly link tall nodes at all appropriate levels. Insertion of a previously-removed node retains the old node's height.

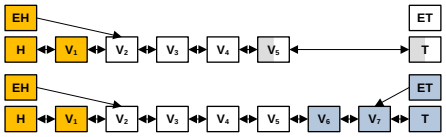


Figure 7: Deque. Pending `pop_front` / `back` operations (top); Pending `pop_front` / `push_back` (bottom).

7.2 Deque, Stack, and Queue

We also provide a deque based on the dlist, by adding two new fields, the effective head (EH) and the effective tail (ET). Figure 7 shows the resulting structure.

The deque associates one `HMeta` field with each node. Operations on the front and back can proceed concurrently, unless the deque is empty or only has one element in it. EH and ET are only used by the transaction that owns the head or tail, respectively, and they serve to identify the point at which the owner's next `push` or `pop` ought to take place in $O(1)$ time. A conflict can occur if different pending transactions own the head and tail, their combined operations logically empty the deque, and either attempts a `pop`. To detect this, we also require ownership of the individual nodes within the deque. In the figure, this ownership is illustrated by a `pop_front` owning both the head and node holding V_1, and a `push_back` owning the tail and the nodes holding V_6 and V_7. `pop` operations use the `removing` field; when a push follows a pop, it can re-use the node, and update EH or ET accordingly. Upon commit, EH (ET) is used to update `head` (`tail`) in constant time. Upon abort, n coroutines may be needed to restore or unlink one element each, in the case that n elements were pushed or popped, each in $O(1)$ time.

We further observe that the effective head technique also applies to priority queues. For example, a skiplist-based priority queue can be implemented with an effective head but no effective tail. This allows conflict-free concurrent insertions, while serializing all calls to `extract_min`. It also allows an insertion to detect, through the EH, when its value is smaller than the current EH, indicating a conflict with a concurrent `extract_min`. We have designed solutions that prioritize the insertion or the removal when these conflicts arise.

7.3 Vector (Resizable Array)

A key feature of Harmony is that it allows the TDS designer to determine the mapping between program data and `HMeta` objects. Our resizable vector leverages this feature so that it is able to remain compatible with the C++ requirement that elements are stored contiguously. Rather than colocate `HMeta` objects with array cells, we provide a separate table of `HMeta` objects for each vector instance. This is, in effect, an ownership record (orec) table from STM [7, 16, 39]. The programmer specifies the size of the table at construction time. We provide a variant where indices map to `HMeta` objects in a round-robin fashion, as well as one where sets of contiguous indices map to the same object. Figure 8 depicts

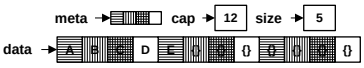


Figure 8: Vector. Shading correlates indices to `HMeta`.

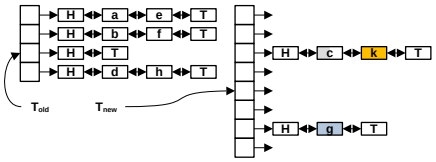


Figure 9: Hash table. The third bucket resized without aborting `insert(k)`, `remove(g)`, or `get(c)`.

the transactional vector, which consists of an array of elements, a table of `HMeta` objects, and size and capacity fields. For simplicity, this discussion only considers expansion.

An access to entry `i` should not conflict with a `push_back` or `pop_back` when the size is larger than `i`. To this end, `push_back` and `pop_back` take ownership of `size`. In the common case, accessing an element of the vector *reads* the `size` field without checking if it is owned, or recording its semantic version. If `i` is a valid position, then the operation will check if the orec corresponding to `data[i]` is owned. If `data[i]` has actually been pushed or popped by a concurrent transaction, a conflict will manifest on the `HMeta` object, not on the size. If `i` is out of range, the transaction will check if `size` is owned, and abort if it is.

When `push_back` discovers that `size` equals `cap`, resizing is needed. Resizing is a structural operation, but must be coordinated with concurrent accesses to the array elements. Those accesses may be by the same transaction, or conflicting transactions, and any of these transactions can subsequently abort and undo. Our solution is to require any coroutine that will run on account of an abort to record the *index* of the position it modified, not the *address*. Thus an operation can resize the table (and move *owned* data) without taking ownership of anything other than the `cap` field itself. Furthermore, if resize is due to `push_back`, it need not undo the resize upon abort: it can even leave the `cap` at its new value.

7.4 Unordered Set/Map (Hash Table)

We also provide a closed addressing hash table, suitable for sets and maps. Figure 9 depicts the data structure design, which is based on [28]. The data structure is an array of dlists, where all inserts are at the head of the list. Elemental operations manage ownership of individual nodes in the appropriate list. `Foreach` operations subscribe to each list's head sentinel's structural version, so they can detect concurrent insertions. Read-only `foreach` also subscribe to the semantic version of each encountered node, while mutating `foreach` take ownership of each encountered node.

By carefully managing structural versioning, resizing does not cause self-aborts with preceding `foreach` operations. Inasmuch as resizing is an internal operation, caused by an `insert`, resizing is free to conflict with concurrent `foreach` operations

in other transactions. Furthermore, since resizing is structural, it can move nodes that are owned, or whose `semantic` version is being watched, without causing aborts. Note that upon abort, we need not undo the resize.

8 PERFORMANCE EVALUATION

In this section, we evaluate Harmony's performance. The current state of the art is Medley, which has the best performance among published TDS systems [6]. We also include results for LFTT [48], OneFile [36], and TDSL [44]. Our test machine has four Intel Xeon Platinum 8160 CPUs at 2.10GHz (96 cores/192 threads) and 754GB of RAM. It runs Ubuntu 22.04.2 LTS with Linux kernel 5.19.0-43-generic. All code was compiled using clang++ 18.1.8 with O3 optimizations, and all data points are the average of five trials, using the jemalloc allocator [11]. Timed experiments ran for 10 seconds each. We did not observe significant variance for any of the experiments.

We use the resizable hash table from Section 7.4 and the skip list from Section 7.1 as our unordered and ordered maps. Medley's unordered map is based on Michael's non-resizable lock-free hash table [30], and uses the recommended settings (1M buckets). Medley's ordered map is derived from Fraser's nonblocking skip list [13], which does not support range queries. Since we are not concerned with persistent memory, we switched Medley from the ralloc persistent allocator [5] to jemalloc, which delivered a 10% performance boost.

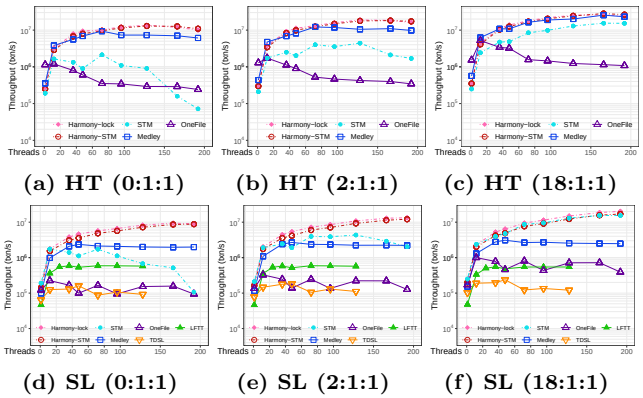
TDSs in Harmony are parameterized on a synchronization mechanism `S`. In "Harmony-lock" experiments, all TDSs use a general-purpose optimistic locking framework similar to [15, 43]. In "Harmony-STM" experiments, the TDSs use an eager STM with undo logging. We also evaluate a version of the workloads where each top-level transaction uses STM (using the same STM as Harmony-STM).

Since prior work only supports ordered/unordered maps, there is no baseline for evaluating the other data structures from Section 7. We developed rigorous unit tests to verify the correctness of these other implementations, even though they are not evaluated in this paper.

8.1 Microbenchmark Performance

We first look at performance in targeted microbenchmarks. To perform these experiments, we integrated our system into Medley's microbenchmark framework. The throughput test preloads the structure with 0.5M key-value pairs, drawn from a key space of 1M keys. Keys and values are 8-byte integers. In the benchmarking phase, each thread composes and executes transactions of 1 to 10 operations each, with each transaction size equally likely. The operation mix among get/insert/remove is specified as 0:1:1, 2:1:1, or 18:1:1, which ensures the data structure stays at a steady size. Keys are chosen with uniform probability. Results for the hash table and skip list appear in Figure 10.

Note that since the Medley implementation of TDSL and LFTT did not include a hash table, we were not able to perform such a comparison. Nonetheless, the relationship between Medley, OneFile, LFTT, and TDSL is roughly the same as reported by Cai et al [6]. In particular, Medley shows the best scalability among prior work. It also outperforms STM in most cases, since writing transactions often must validate at commit time, increasing latency. However, the overheads of STM are much lower than TDSL, confirming recent findings [4, 35, 43] that modern STM is competitive with nonblocking data structures (such as those in Medley).



(a) HT (0:1:1) (b) HT (2:1:1) (c) HT (18:1:1)

(d) SL (0:1:1) (e) SL (2:1:1) (f) SL (18:1:1)

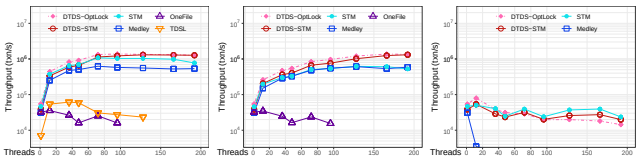
Figure 10: Hash table (HT) and skip list (SL) microbenchmark with varied get:insert:remove ratios.

Harmony matches or exceeds Medley in all experiments. There are a number of sources of improvement. First, our lock-based implementations have low latency, particularly since Medley must superimpose a multiword compare-and-swap on top of data structure operations. Second, because Harmony decouples physical structure from logical metadata, concurrent operations can safely traverse past nodes that are currently *semantically owned* by other transactions. This structural bypassing prevents the false bottlenecks inherent in structural TDS systems, explaining Harmony's superior scaling for write-heavy workloads.

8.2 TPC-C Performance

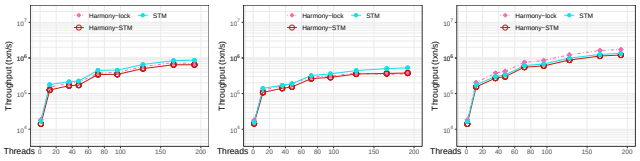
Next, we consider performance on the TPC-C benchmark suite. TPC-C supports five transaction types. Unfortunately, Medley (and indeed all prior work that has been ported to Medley's benchmark system) does not yet support range operations, so they can only execute two transaction types: NewOrder and Payment. We split our evaluation into two parts: First, we consider only these two transactions, so that we can compare against prior work. Then we look at how Harmony performs for the other transaction types.

Figure 11 uses the configuration from Medley to run an equal mix of NewOrder and Payment transactions. For low contention, there are as many warehouses as threads, and a static thread:warehouse assignment. For medium contention, we keep the same number of warehouses, but threads pick a warehouse at random for each transaction. We also added a "high contention" workload, where all threads contend on a single warehouse. In this pathological configuration,



(a) Low (b) Medium (c) High

Figure 11: TPC-C, 1:1 mix of NewOrder and Payment transactions, parameterized by contention level.



(a) Low contention (b) Medium contention (c) High contention

Figure 12: TPC-C, read-only and range transactions.

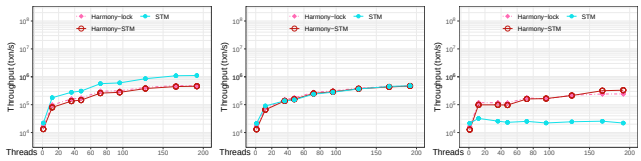
scalability is not possible and livelock is likely, as all Payment transactions serialize on a single row.

TDSL and OneFile have high latency and reach a scalability bottleneck quickly. At higher contention levels, they frequently timed out, so we omit results in those cases, and focus our discussion on the performance of Medley, STM, and Harmony. Under low and medium contention, Harmony and STM outperform Medley by a consistent margin. In the absence of contention, their overheads are similar: STM has more logging overhead, but Harmony incurs boundary instrumentation on each TDS operation, whereas STM only incurs this overhead at user-level transaction boundaries.

At high contention, Medley livelocks: While the individual TDS operations in Medley are lock-free, the composition is only obstruction-free. In contrast, Harmony and STM degrade more gracefully. This difference is despite the fact that all three approaches use eager conflict detection. The key issue is that once a Harmony or STM transaction takes ownership of a hot `HMeta` object, it blocks others transactions, which thus are less likely to cause it to abort. However, the main conclusion of this result is simply that all three systems would benefit from more nuanced contention management [40].

While Medley's skip list does not support range operations, we had no trouble implementing mutating and read-only ranges for our skip list. Figure 12 shows the performance of STM and Harmony for an equal mix of OrderStatus and StockLevel transactions, both of which require read-only range operations. Both systems show good scaling. We see a slight advantage for STM, because read-only operations in STM do not need to validate at commit time. However, as contention increases, this benefit diminishes, and at high contention, the systems perform equally well.

Finally, Figure 13 shows the performance of a full TPC-C run, using the recommended workload mix of 45% NewOrder, 43% Payment, 4% OrderStatus, 4% Delivery, and 4% StockLevel transactions. Under low contention, both systems scale



(a) Low (b) Medium (c) High

Figure 13: TPC-C, default workload mix, all transaction types, parameterized by contention level.

well. As contention increases, STM performance degrades more steeply than Harmony. This workload highlights the fundamental weakness of structure-based conflict detection: The STM baseline conservatively treats every physical memory read as part of the transaction's conflict set, leading to false aborts. In contrast, Harmony uses `HMeta` to filter these out, tracking only true semantic conflicts. This allows Harmony to sustain high throughput even under high contention, confirming the impact of a design that separates semantic conflicts from synchronization.

9 CONCLUSIONS

In this paper, we presented Harmony, a novel transactional data structure system that bridges the divide between semantic and structural conflict detection. Harmony embeds metadata in data structure nodes, and uses it to physically materialize semantic conflicts among transactions. This approach decouples logical conflict detection from physical synchronization, achieving the precision of semantic conflict detection and the generality of structural techniques. In addition, Harmony employs new techniques for managing the non-existence of data, and introduces a novel coroutines-based mechanism that elegantly defers structural cleanup. Together, these techniques allow it to remain completely agnostic to the underlying program data.

This unified design yields exceptional performance, characterized by low latency and best-in-class scalability. Across standard microbenchmarks and complex TPC-C workloads, Harmony consistently matched or outperformed the state of the art, while remaining resilient to contention. In addition to performance, Harmony achieves a high degree of programmability and expressiveness. It is the first TDS system we are aware of that can implement diverse data structures, including sets, maps, vectors, and deques, and also implement complex operations, including range, scan, and concurrent hash table resizing.

We see two exciting directions for future work. First, given that high-contention workloads fundamentally bottleneck on logical conflicts rather than physical locks, there is a clear need to revisit the study of contention management within this new low-level semantic context. Second, we believe the broad generality of our design opens new opportunities to begin deploying transactional data structures in emerging applications that require strict isolation across multiple, highly complex concurrent data structures.

REFERENCES

[1] Maya Arbel-Raviv and Trevor Brown. 2018. Harnessing Epoch-Based Reclamation for Efficient Range Queries. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Vienna, Austria.

[2] Naama Ben-David, Guy Blelloch, and Yuanhao Wei. 2022. Lock-Free Locks Revisited. In *Proceedings of the 27th ACM Symposium on Principles and Practice of Parallel Programming*. Seoul, South Korea.

[3] Trevor Brown. 2015. Reclaiming Memory for Lock-Free Data Structures: There has to be a Better Way. In *Proceedings of the 34th ACM Symposium on Principles of Distributed Computing*. Portland, OR.

[4] Trevor Brown, William Sigouin, and Dan Alistarh. 2022. PathCAS: An Efficient Middle Ground for Concurrent Search Data Structures. In *Proceedings of the 27th ACM Symposium on Principles and Practice of Parallel Programming*. Seoul, South Korea.

[5] Wentao Cai, Haosen Wen, H. Alan Beadle, Chris Kjellqvist, Mohammad Hedayati, and Michael L. Scott. 2020. Understanding and Optimizing Persistent Memory Allocation. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management* (London, UK).

[6] Wentao Cai, Haosen Wen, and Michael L. Scott. 2023. Transactional Composition of Nonblocking Data Structures. In *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures*. Orlando, FL.

[7] Dave Dice, Ori Shalev, and Nir Shavit. 2006. Transactional Locking II. In *Proceedings of the 20th International Symposium on Distributed Computing*. Stockholm, Sweden.

[8] Thomas Dickerson, Eric Koskinen, Paul Gazzillo, and Maurice Herlihy. 2019. Conflict Abstractions and Shadow Speculation for Optimistic Transactional Objects. In *Proceedings of the Asian Symposium on Programming Languages and Systems*. Bali, Indonesia.

[9] Thomas D. Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. 2017. Brief Announcement: Proust: A Design Space for Highly-Concurrent Transactional Data Structures. In *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017*. ACM, 251–253.

[10] Kapali P. Eswaran, Jim Gray, Raymond A. Lorie, and Irving L. Traiger. 1976. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM* 19, 11 (1976), 624–633.

[11] Jason Evans. 2017. jemalloc memory allocator. http://http://jemalloc.net/.

[12] Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. 2017. Elastic Transactions. *J. Parallel and Distrib. Comput.* 100 (Feb. 2017), 103–127. Issue C.

[13] Keir Fraser. 2003. *Practical Lock-Freedom*. Ph. D. Dissertation. King's College, University of Cambridge.

[14] Rachid Guerraoui and Michal Kapalka. 2008. On the Correctness of Transactional Memory. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*. Salt Lake City, UT.

[15] Rachid Guerraoui and Vasileios Trigonakis. 2016. Optimistic Concurrency with OPTIK. In *Proceedings of the 21st ACM Symposium on Principles and Practice of Parallel Programming* (Barcelona, Spain).

[16] Tim Harris and Keir Fraser. 2003. Language Support for Lightweight Transactions. In *Proceedings of the 18th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*.

[17] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. 2005. Composable Memory Transactions. In *Proceedings of the 10th ACM Symposium on Principles and Practice of Parallel Programming*. Chicago, IL.

[18] Ahmed Hassan, Roberto Palmieri, and Binoy Ravindran. 2014. Optimistic Transactional Boosting. In *Proceedings of the 19th ACM Symposium on Principles and Practice of Parallel Programming*. Orlando, FL.

[19] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William Scherer, and Nir Shavit. 2006. A Lazy Concurrent List-Based Set Algorithm. In *Proceedings of the 9th international conference on Principles of Distributed Systems*. Pisa, Italy.

[20] Maurice Herlihy and Eric Koskinen. 2008. Transactional Boosting: A Methodology for Highly-Concurrent Transactional Objects. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*. Salt Lake City, UT.

[21] Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. 2021. *The Art of Multiprocessor Programming, 2nd edition*. Morgan Kaufmann.

[22] Maurice P. Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. 2003. Software Transactional Memory for Dynamic-sized Data Structures. In *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing*. Boston, MA.

[23] Maurice P. Herlihy and J. Eliot B. Moss. 1993. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th International Symposium on Computer Architecture*. San Diego, CA.

[24] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: a Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems* 12, 3 (1990), 463–492.

[25] Nathaniel Herman, Jeevana Priya Inala, Yihe Huang, Lillian Tsai, Eddie Kohler, Barbara Liskov, and Liuba Shrira. 2016. Type-aware transactions for faster concurrent code. In *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*. ACM, 31:1–31:16.

[26] Tomas Karnagel, Roman Dementiev, Ravi Rajwar, Konrad Lai, Thomas Legler, Benjamin Schlegel, and Wolfgang Lehner. 2014. Improving In-Memory Database Index Performance with Intel Transactional Synchronization Extensions. In *Proceedings of the 20th International Symposium on High-Performance Computer Architecture*. Orlando, FL.

[27] H. T. Kung and John T. Robinson. 1981. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems* 6, 2 (1981), 213–226.

[28] Yujie Liu, Kunlong Zhang, and Michael Spear. 2014. Dynamic-Sized Nonblocking Hash Tables. In *Proceedings of the 33rd ACM Symposium on Principles of Distributed Computing*. Paris, France.

[29] Alexander Matveev, Nir Shavit, Pascal Felber, and Patrick Marlier. 2015. Read-log-update: a lightweight synchronization mechanism for concurrent programming. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 168–183.

[30] Maged Michael. 2002. High Performance Dynamic Lock-Free Hash Tables and List-Based Sets. In *Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures*. Winnipeg, Manitoba, Canada.

[31] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. 2008. STAMP: Stanford Transactional Applications for Multi-processing. In *Proceedings of the IEEE International Symposium on Workload Characterization*. Seattle, WA.

[32] Eliot Moss. 2006. Open Nested Transactions: Semantics and Support. In *Proceedings of the 4th Workshop on Memory Performance Issues*. Austin, TX.

[33] Jacob Nelson, Ahmed Hassan, and Roberto Palmieri. 2022. Bundling Linked Data Structures for Linearizable Range Queries. In *Proceedings of the 27th ACM Symposium on Principles and Practice of Parallel Programming*. Seoul, South Korea.

[34] Christos H. Papadimitriou. 1979. The Serializability of Concurrent Database Updates. *Journal of the JACM* 26, 4 (1979), 631–653.

[35] Pedro Ramalhete and Andreia Correia. 2024. Scaling Up Transactions with Slower Clocks. In *Proceedings of the 29th ACM Symposium on Principles and Practice of Parallel Programming*. Edinburgh, UK.

[36] Pedro Ramalhete, Andreia Correia, Pascal Felber, and Nachshon Cohen. 2019. OneFile: A Wait-Free Persistent Transactional Memory. In *Proceedings of the 49th International Conference on Dependable Systems and Networks*. Portland, OR.

[37] Wenjia Ruan, Yujie Liu, and Michael Spear. 2013. Boosting Timestamp-based Transactional Memory by Exploiting Hardware Cycle Counters. *ACM Transactions on Architecture and Code Optimization* 10, 4 (Dec. 2013), 40:1–40:21.

[38] Wenjia Ruan, Trilok Vyas, Yujie Liu, and Michael Spear. 2014. Transactionalizing Legacy Code: An Experience Report Using GCC and Memcached. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. Salt Lake City, UT.

[39] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. 2006. McRT-STM:

A High Performance Software Transactional Memory System For A Multi-Core Runtime. In *Proceedings of the 11th ACM Symposium on Principles and Practice of Parallel Programming*. New York, NY.

[40] William N. Scherer III and Michael Scott. 2005. Advanced Contention Management for Dynamic Software Transactional Memory. In *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing*. Las Vegas, NV.

[41] Nir Shavit and Dan Touitou. 1995. Software Transactional Memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*. Ottawa, ON, Canada.

[42] Gali Sheffi, Maurice Herlihy, and Erez Petrank. 2021. VBR: Version Based Reclamation. In *35th International Symposium on Distributed Computing, DISC 2021, Freiburg, Germany (Virtual Conference), October 4-8, 2021 (LIPIcs)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 35:1–35:18.

[43] Yaodong Sheng, Ahmed Hassan, and Michael Spear. 2023. Separating Mechanism from Policy in STM. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. Vienna, Austria.

[44] Alexander Spiegelman, Guy Golan-Gueta, and Idit Keidar. 2016. Transactional Data Structure Libraries. In *Proceedings of the 37th ACM Conference on Programming Language Design and Implementation*. Santa Barbara, CA.

[45] Shahar Timnat, Maurice Herlihy, and Erez Petrank. 2015. A Practical Transactional Memory Interface. In *Proceedings of the 2015 Euro-Par Conference*. Vienna, Austria.

[46] Qingsen Wang, Pengfei Su, Milind Chabbi, and Xu Liu. 2019. Lightweight Hardware Transactional Memory Profiling. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. Washington, DC.

[47] Yuanhao Wei, Naama Ben-David, Guy Blelloch, Panagiota Fatourou, Eric Ruppert, and Yihan Sun. 2021. Constant-time Snapshots with Applications to Concurrent Data Structures. In *Proceedings of the 26th ACM Symposium on Principles and Practice of Parallel Programming*. Seoul, South Korea.

[48] Deli Zhang, Pierre Laborde, Lance Lebanoff, and Damian Dechev. 2018. Lock-Free Transactional Transformation for Linked Data Structures. *ACM Transactions on Parallel Computing* 5, 1 (June 2018).

[49] Ferad Zyulkyarov, Vladimir Gajinov, Osman Unsal, Adrian Cristal, Eduard Ayguade, Tim Harris, and Mateo Valero. 2009. Atomic Quake: Using Transactional Memory in an Interactive Multiplayer Game Server. In *Proceedings of the 14th ACM Symposium on Principles and Practice of Parallel Programming*. Raleigh, NC.