**Transactional Data Structures Libraries**

**Multi-Core Architecture and Systems**

**Ofir Yakovian**

Supervised by Prof. Adam Morrison.

# **Introduction**

**Concurrent Data Structures** are everywhere in the software stack, from core low-level kernels to user applications. These data structures, when inefficient, can cripple the performance of the system. Concurrent Data Structure permit many threads to operate on common data using a high-level interface. When a data structure is accessed concurrently by many threads, its semantics are typically defined by a property called “**Linearizability**”, which provides strong consistency.

Linearizability requires that each operation appear to take effect instantly at some point between its invocation and response. [Linearizability](http://cs.brown.edu/~mph/HerlihyW90/p463-herlihy.pdf) is a guarantee about single operations on single objects, and it provides a real-time guarantee on the behavior of a set of single operations (often reads and writes) on a single object (such as a distributed register).

Concurrent algorithms with strong linearizability are quite complex to design. Currently, each new Concurrent Data Structure requires its own paradigm, and the community has spent decades writing papers and developing algorithms for all kinds of structures (e.g., Skiplists, Priority Queues, Hash Tables). Unsurprisingly, key major challenges in designing Concurrent Data Structures are dealing with contention penalties, which occurs when operations often affect the output of other ones. **Operation Contention** is challenging because operations must observe each other across cores (For example, a write of a new value affects a subsequent read), while handling complex sharing and overlays.

More complications arise with “**Composition**”. A composition of Concurrent Data Structures is a powerful approach to combine simple structures to create more complex ones, and it works as a building block for many advanced useful data structures.

Our work explores the challenges in combining Concurrent Data Structures with respect to ensuring the consistency as well as the lock-correctness of the composed data structure. We also implement an algorithm and experimentally evaluate its performance. We study the aforementioned by using both theoretical and practical capabilities of **Transactional Data Structures**, allowing us to design efficient transactionalalgorithms, reaching the speed of their concurrent counterpart.

# **Transactional Data Structures**

When threads are executed concurrently on different processors, they subject to Operating System scheduling decisions, page faults, interrupts, etc. This significantly complicates the task of designing correct Concurrent Data Structures. Designing Concurrent Data Structures for multiprocessor systems also provides numerous challenges with respect to performance and scalability. Furthermore, the issues of correctness and performance are closely tied to each other - algorithmic enhancements that seek to improve performance, often make it more difficult to design a certified data structure implementation.

**Concurrent Data Structure Libraries** (CDSLs) formally expose Concurrent Data Structures available in various interfaces. If numerous operations on multiple data structures are required, library consumers might violate the concurrency model. To achieve a correct shared data structure behavior, consumers often synchronize manually. That often comes with a price, as locking introduces a host of problems related to both performance and software deadlocks.

With the emergence of a “transactional software” approach, several CDLS have tried to adopt their inner implementation to **Software Transactional Memory** (STM), a novel design that supports flexible transactional programming of synchronization operations in software. One can use STM to dynamically resolve inconsistencies and deadlocks, by rolling back transactions using a memory access instrumentation. This solution has not been adopted due to poor performance, that incurs high overhead in practice.

Our goal is to provide transaction semantics for CDSLs without sacrificing performance. We leverage the concept of a **Transactional Data Structure Library** (TDSL), which unlike STM, encloses transactional operations on TDSL solely, whereas other memory accesses are not protected (thus potentially avoiding the performance overheads). Furthermore, when considering stand-alone operations (denoted “**Singletons**”), we might schedule transactions wisely to improve the actual abort rate and reduce the overall management dealt by state-of-the-art STM counterpart libraries.

We attain the former objectives by focusing on transactional access to a well-defined set of data structure operations, and implementing a native library supporting transactions on any number of sets (implemented as Skiplists), which heavily uses the algorithmic approach appears in [Transactional Data Structure Libraries](https://iditkeidar.com/wp-content/uploads/files/ftp/TransactionalLibrariesPLDI16.pdf) (**TDSL** of Alexander Spiegelman, Guy Golan-Gueta and Idit Keidar).

The practical side of the paper is developed using the Java programming language. While being a natural programming choice for dealing with concurrency problems due to strong built-in primitives, it incurs a performance penalty from garbage collection, which can cause unpredictable stalls. As for now, it remains unclear whether TDSL is surpassing best of bread CDSLs implementations, which are mostly provided in native programming languages designed as an abstracted form of assembler with optimizing compilers.

Ultimately, we introduce transactions into a library of Concurrent Data Structures, denoted “**TiXeL**”, that enables custom-tailored benchmark tests, providing a robust assessment that no performance degradation has been made compared to existing CDSL implementations.

# **The TiXeL Library**

TiXeL is an open-source, low-level, cross-platform library, designed to provide a portable, robust, fast, and extensive **Transactional Data Structures**, allowing the composition of any number of well-defined concurrent operations.

As a north-star approach, TiXeL introduces simple (yet useful) API sets, that produce robust code for the long term, and can be easily integrated into existing benchmark suites used to evaluate synchronization techniques on various data structures. It also facilities the porting process of legacy CDSL code, by allowing operations outside of transactions (treated as singletons). Singletons cannot abort, and so legacy code can continue to use the original thread-safe library operations.

The semantics of singletons relative to other transactions is preserved. In other words, each TiXeL run has a straightforward linearization, encompassing all of its transactions and singletons.

## **Software Design Guidlines**

TiXeL was structured to accommodate change, based on the following principles:

* **Compatibility**: TiXeL can operate with other products that are designed for interoperability with existing CDSL libraries.
* **Robustness**: TiXeL is designed with resilience to heavy transactional benchmarks on multi-core platforms.
* **Portability**: TiXeL is portable and platform-neutral. It works on different OSes while being agnostic to the underlying processor architecture.
* **Performance:** TiXeL reuses shared resources across transactions, paying for the transaction set-up and tear-down only once, without making transactions dependent on each other.
* **Maintainability -** TiXeL liberates benchmark writers from housekeeping chores, and let them focus on the benchmark content. It automatically keeps track of all workloads defined, and doesn't require their enumeration.
* **Extensibility:** TiXeL can support new Transactional Data Structure of different characteristics without major changes to the underlying architecture
* **Fault-Tolerance**: TiXeL is resistant to and able to recover from transactional aborts. Transactions that are deemed to abort don’t see inconsistent states of the data structure.
* **Modularity:** TiXeL comprises well-defined and independent layering model, which leads to better maintainability. Different components can be easily implemented and tested in isolation.

## **Modular Engineering Components**

TiXeL decomposes into smaller programs with standardized interfaces:

**Text

Description automatically generated**

* **TiXeL Infrastructure (TixelInfra)**: Houses the baseline for the C++ programming ecosystem. It formalizes the exception model, logging facilities, and safe synchronization capabilities.
* **TiXeL Abstract Data Types (TixelAdt)**: Introduces the **IList<K,V>** Abstract Data Type (ADT), whose behavior is defined by a set of well-defined operations encapsulated with that matching interface. These definitions only mention what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations.

Furthermore, it introduces an **Index** (backed by a Skiplist), that would later be used to shorten the access time of the Linked List. Each **Index Node** extends a **Version Lock** (implemented in **TixelInfra**). In addition to versions, nodes are associated with locks, to allow a particular transaction changing all the nodes it affects atomically.

* **TiXeL Transactional Layer** **(TixelTxn)**: Introduces the following fundamental components:
  + Global Version Clock (**GVC**): exposes two operations: **READ**, which returns the current global version, and **ADD-AND-FETCH**, that atomically increases the global version and returns its new value. To ensure **Opacity**, this global clock is read at the beginning of each transaction (denoted as transaction’s read version), used to detect conflicts among concurrent transactions.
  + TXN List Transaction: Implements the fundamental phases of a general transaction: **Begin**, **Commit**, **Rollback** (sometimes referred to as **Abort**). To preserve strong encapsulation, and to prevent improper library usages, these building blocks are not exposed to library consumers, whatsoever.

Each transaction prologue and/or epilogue operates on a **TXN Local Storage** (which is actually a **Thread Local Storage**). This storage encapsulates all transaction-local data, including locks owned along the transaction processing.

Text

Description automatically generated

This storage encapsulates all transaction-local data, including locks owned along the transaction processing, transaction’s **Read Set** & **Write Set**, and transaction’s **Index Add** & **Index Remove**. The later encapsulate the add and remove operations it needs to perform on the **Index**, which is only lazily updated outside the scope of the transaction (since an added node cannot be removed from the **Index** by an older remove).

* TXN Dispatcher: Guides the lifecycle of a TXN List Transaction, and executes a given TXN routine that typically operates on the underlying data structure. Library operations invoked between a TX **Begin** and the following TX **Commit** are associated to the same transaction.

The library may abort a transaction during any of the operations, issuing a **Transaction Exception**. In case of a **Rollback**, none of the transaction’s operations are reflected in the data structure. Library consumers catch this exception, at which point they typically restart the transaction.

Text

Description automatically generated

* TXN List: Introduces the logical (algorithmic) layer of the actual **Transactional Data Structure**. Each node in the Linked List is tagged with a version, which is updated by transactions that create or update the node.

A transaction validates its reads by checking that their versions have not been increased (by other transactions), since the transaction had begun. Transactions re-validate all their reads at commit time. In case of conflicts (manifested as newer versions), the subjected transaction aborts.

Each TXN List consist of two TXN List Operators, to optimize singleton handling. Since singletons do not increment the **GVC**, transactions cannot rely on versions alone to detect conflicts with singletons. Hence, the **Version Lock** maintains an additional singleton bit.

Each ADT operation routes to the corresponding TXN List Operator, depending on whether an in-flight transaction is already being proceed by the currently dispatched thread.

Text

Description automatically generated

## **Formal Test Suits**

Several quality measures of TiXeL aim to reveal failures severe enough to reject a prospective release, and certify our algorithmic performance. Our tests cover the most important functionality of our system, used to aid assessment of whether main functions of the software appear to work correctly. The following suits are currently in action:

* **TiXeL Unit Tests** **(TixelUnitTests)**: Tests the smallest TiXeL units to certify the library correctness, using the **Google Test** (gtest) unit-testing library for C++.
* **TiXeL Console (TixelConsole)**: Provides the **Command-Line Interface** (CLI) of TiXeL, to facilitate future test automations.
* **TiXeL Benchmark** **(TixelBench)**: Introduces the benchmark performance testing, the metric against which TDSL can be compared to assess our quality measures. We enable benchmark evaluations using **SynchroBench**.

SynchroBench is a micro-benchmark suite used to evaluate synchronization techniques on data structures. SynchroBench is written in C/C++ and Java and was enhanced to include transactional operations for the course of the project.