AsyncTx: Software Transactional Memory Primitives for Rust

William Zhang

May 2022

1 Abstract

This report describes an implementation of software transactional memory (STM) primitives in Rust on top of the language's `Future` abstraction. Data protected by the STM runtime are accessed within transactions, where all actions within the transaction are visible to other threads as if performed by a single instruction. My runtime *AsyncTx* supports the construction of a wide range of transactionally protected data types. I demonstrate how *AsyncTx* transparently isolates transactions from unprotected state via Rust features such as lifetimes and macros and allows for the construction of complex data structures such as a log-backed key-value. I provide the results of microbenchmarks against programs protected via standard synchronization primitives such as locks, demonstrating the trade-offs between the two approaches.

2 Introduction

Rust is a low-level systems programming language that has been rapidly growing in popularity. At its core is its motto of *Fearless Concurrency*, where relatively small amounts of annotation provided by the programmer provides safety guarantees such as freedom from data races, buffer overflows, and memory leaks. Rust however does not provide total protection against unsafe operations such as deadlocks. This paper implements a software transactional memory (STM) runtime that aids the programmer from avoiding deadlocks and subtler errors such as priority inversion. STM runtimes provide a set of guarantees for a set of operations grouped into a single *transaction*. The most stringent set of guarantees follows the ACID paradigm, where the operations are committed *atomically* as if they were performed by a single instruction, the operations don't violate system *consistency* invariants, the intermediate values generated by the transaction are *isolated* and opaque to all other transactions until commit, and committed values remain *durably* persistent for all future transactions until they are overwritten. Transactional systems sometimes weaken certain guarantees usually for performance.

2.1 Future

I built my runtime on top of Rust's asynchronous support. Rust's standard library provides a trait `Future` which provides an interface for user-level yieldable tasks designed to handle asynchronous operations. The `Future` trait exposes a single interface function, `poll`:

```
pub enum Poll<T> {
    Ready(T),
```

```
    Pending,
}

pub trait Future {
    type Output;
    fn poll(self: Pin<&mut Self>,
                cx: &mut Context<'_>) ->
                Poll<Self::Output>;
}
```

A third-party executor invokes the `poll` function to make progress on computing the unresolved return value, providing the future with callbacks notifying the executor when it is ready to be polled again through the `cx` argument. The future notifies the executor by returning `Poll::Ready(T)` when the result has been computed or `Poll::Pending` if it requires to be yielded for asynchronous IO or some other task. A sample implementation of single-threaded and multi-threaded executors is provided in the library under the `runtime` module.

The nature of this interface necessitate a state-machine oriented implementation for all user-level tasks. To conform to its mostly iterative programming model, Rust provides `colored` functions and code blocks that are automatically compiled into a state machine. This allows for the composition of primitive implementations of `Future` such as an asynchronous IO implementation into a higher-level future, where transitions between states in the state machine are coordinated via the invocation of the `await` keyword on the encapsulated futures. As an example, the following code block shows `colored` async block.

```
let arr: [usize; 2] = [1, 2];
let sum = async {
    let first = &arr[0];
    let second = &arr[2];
    sleep(Duration::from_secs(1)).await;
    let sum = *first + *second;
    sum
}.await;
```

This block would be compiled into an opaque state machine consisting of two states: the state prior to polling on the future storing the reference to the array outside of the block and the state prior to polling on `sleep` where the two references `first` and `second` are stored. The inner `await` would be compiled to an executor of that future where notifications from and to the higher-level executor are interposed.

2.2 Lifetimes

As all Rust types are annotated with lifetimes, this includes the opaque structures generated by the compiler. Rust uses its type system to help annotate lifetimes, where lifetimes that can be verifiably ordered against one another are treated in a parent-child relationship. The programmer can also explicitly label the ordering of lifetimes. These explicit and implicit annotations are verified throughout the program, and violations that would ever lead to a use-after-free are prohibited at compile time. A special lifetime `'static` is ordered above all other lifetimes, representing static values that are never freed until the program terminates or when their reference counter drops to zero when using the special reference counter wrappers (`Arc`, `Rc`). Types adopt the lifetimes of all the references contained within them. In the previous example, this would make the opaque future adopt `arr`'s lifetime.

3 Implementation

3.1 Access Control

I implement transactions as an `AsyncTx` struct implementing the `Future` trait. The `AsyncTx` struct encapsulates another future and interposes on all of its yields. By requiring that the future have a static lifetime, `AsyncTx` prevents any transaction that uses references outside of its scope from being constructed, with the exception of static variables. Rust, through special rules provided by the marker traits `Send` and `Sync` furthermore guarantees that if a reference to a static variable is provided, it must be immutable. These restrictions enable *isolation* of the transaction, as it is unable to use references to indirectly snoop into another transaction.

Transactionally protected data is encapsulated in a special `TxData` wrapper, which provides explicit, asynchronous read, write, and set accessor functions to the local data via handlers. Handlers promote from *unresolved* to *shared* when `read` is invoked and from *unresolved* or *shared* to *owned* when `write` or `set` is invoked. Invoking `read` on an unresolved handler first acquires the lock onto the associated data, then copies a read-only reference-counted versioned copy of the protected data, registers the data in a statically-stored, thread-local *transaction context*, and checks if all registered data in the context are up-to-date. Should any data currently opened for reading be determined to be stale, the transaction is aborted by propagating an error to the parent `AsyncTx` future through registering the abort in the context and polling on an unresolvable future. Invoking `write` performs the same actions as `read` due to implicitly being the same as copying the old value while in addition labelling the value as requiring a commit lock. Invoking `set` performs no action.

3.2 Two Phase Commit

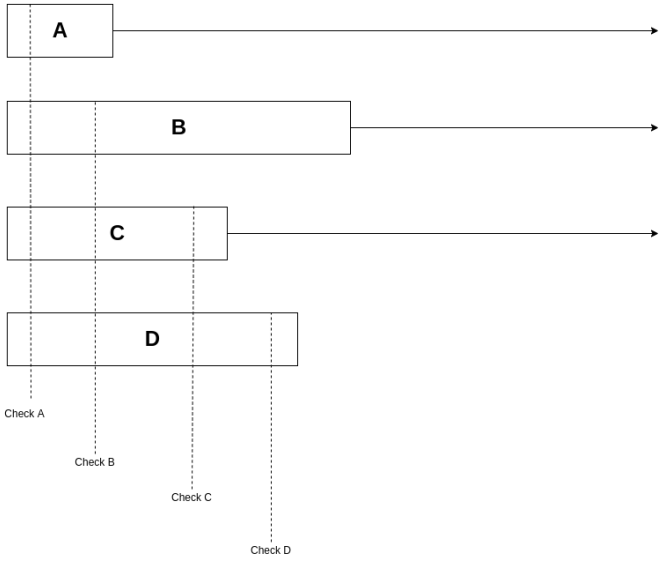

Figure 1: Here the boxes describe when the versioned data a transaction is checking is not stale. Even if a value becomes stale after the check, we can guarantee that it wasn't stale during the first check.

Two phase commit is performed by first sorting the addresses of then locking all the data accessed via an *owned* handler. A total order of the locks prevents deadlocks from occurring between two transactions that commit at the same time. After the locks are acquired, all data whose first access was not `set` are checked for whether they are stale. If so, to guarantee that all transactions appear sequentially ordered, the transaction aborts. Otherwise, the transaction commits. Figure 1 demonstrates why not locking the readers when checking for staleness is safe. Linearizability can be guaranteed by choosing the first check as the point that the commit occurred. Locks on the writers guarantee that no other transaction is able to have an inconsistent view of the committed values.

3.3 Condition Variable Semantics

Inspired by Haskell's internal STM implementation [1], transactions can signal the runtime to immediately abort and block the thread until a value becomes updated by another transaction. I provide this functionality through a `TxWaiter` future, which signals the transaction context to check if the data to wait on has updated. The future will poll until a committing transaction wakes it up where it will then invoke an abort.

4 Microbenchmarks

I implemented three benchmarks compared against a synchronous implementation using mutexes to evaluate the runtime: a multi-producer, multi-consumer (MPMC) array-backed queue with a buffer size of 32 inspired by [1], a hashmap with 100 key value pairs with entry-level locking granularity that is uniformly accessed, and a similar hashmap except accessed via a hot-cold pattern where the first 10% of keys are accessed 90% of the time. The MPMC queue test involves an equal number of producers and consumers enqueuing and dequeuing 13440 elements split equally. My AsyncTx implementation uses wait semantics to enforce blocking when the queue is empty or full. The hashmap tests involve performing 13440 transactions acting on up to 5 entries at a time sampled with replacement as described above. The MPMC queue tests heavy write contention due to necessitating updating the producer and consumer pointers, the uniformly accessed hashmap tests read-dominated, nonconflicting workloads, while the hot-cold pattern tests read-dominated, conflicting workloads. Testing occurred on a 16-core AMD Ryzen 7 5800H laptop running the Linux distribution Pop!_OS 22.04 LTS. Code was compiled on stable Rust 1.60.0. Statistics were generated via the Criterion crate, which runs benchmarked functions at over 100 iterations and computes a 90% confidence interval using bootstrap resampling.

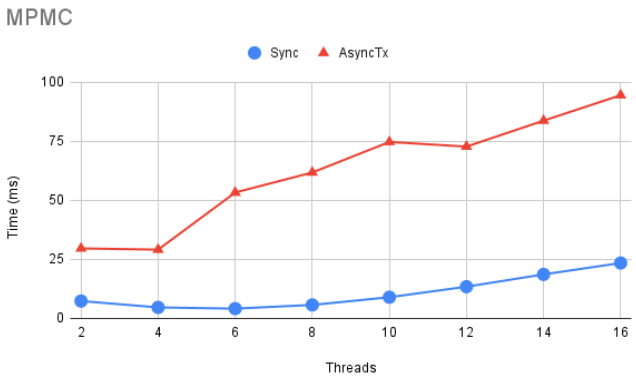

Figure 2: Multiple Producer Multiple Consumer Queue

Shown in Figure 2 are the results of the MPMC benchmarks. Error bars for this figure and subsequent ones are omitted due to them never overlapping and usually close to not visible, though 90% confidence interval endpoints are provided in the raw data. *AsyncTx* performs poorly when writes are heavily contended. I expect this to be as a result of my implementation's lazy conflict detection leading to a high cost in aborts. Surprisingly, the implementation still maintains within 5x slowdown compared to the synchronous implementation. More information is needed on whether real workflows ever exhibit highly-conflicting write behavior, but regardless of rarity, addressing this bottleneck is essential as a future direction.

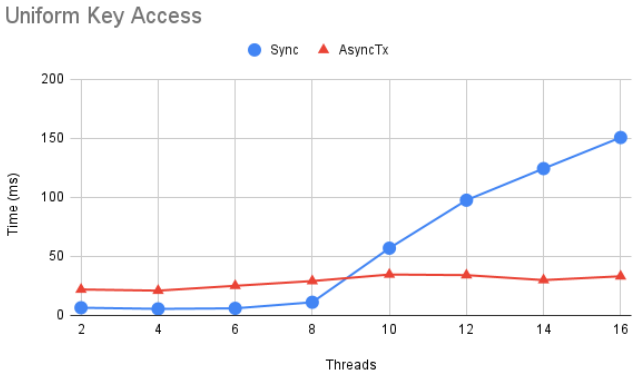

Figure 3: Uniform Key Access

Shown in Figure 3 are the results of the uniform hashmap access benchmark. Surprisingly, *AsyncTx* seems to almost be unaffected by the increased probability of conflicts. Due to my implementation's lazy conflict detection, this leads to my implementation behaving closer to a read-write lock, allowing for concurrency between conflicting readers. The synchronous implementation, however, requires that the locks for all associated operations be held during the entire duration, increasing the probability of conflicts.

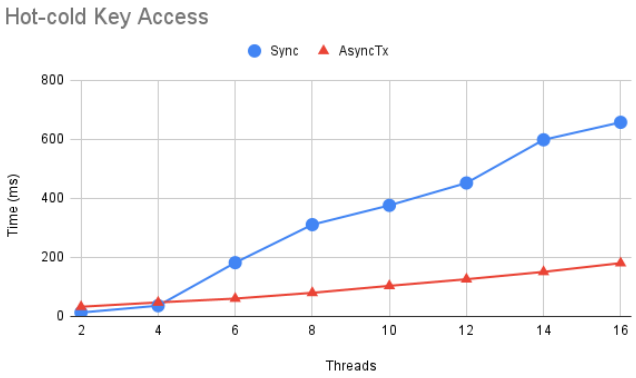

Figure 4: Hot-cold Key Access

Shown in Figure 4 are the results of the hot-cold hashmap access benchmark. Very surprisingly, *AsyncTx* does not seem to exhibit the same behavior as in the MPMC test case despite writes being more likely to conflict. However, there is a cost to conflicting transactions present in the figure; slightly higher costs in managing conflicts are exhibited.

5 Applications

I demonstrate the versatility of my API through implementing an API for supporting log-based data structures. This entails wrapping the head of the log in a *TxData* wrapper. To allow an implementer to transparently provide their data structures to users, a trait `TxLogView` is provided with the following specifications:

```
pub trait TxLogView: Default + Into<TxLogEntry<Self::Record>> {
    type Record;
    fn consume_prev(
        &mut self,
        entry: &Arc<TxLogEntry<Self::Record>>
    ) -> bool;
}
```

The `consume_prev` interface function is used to iteratively read through the log, returning false when reading is not needed. Requiring implementors to provide a `Default` implementation allows construction of the view by a special `TxLogStructureHandle`. Into allows the modifications to the view to be persisted into an entry in the log. Persistence is carefully only executed at commit time to allow for expensive operations to be performed without worry of an abort, though at an expense of a longer commit window and therefore higher likelihood of priority inversion.

6 Future Directions

Partially implemented is support for blocking `TxData` wrappers. These acquire locks immediately to limit the cost of aborts. I implemented a technique for handling deadlocks by implicitly constructing a waits-for graph. This consists of the owning transaction of the lock posting a *deed*, a sign of ownership, on all locks that they own. When blocking, the transaction creates an

edge between its deed to the deed stored in the lock. A single pass iteration can detect a cycle. On detection, the edge is removed and the transaction proceeds, ignoring the lock. The tortoise and the hare algorithm is not required under the assumption that another transaction will break the cycle should one be stuck in an infinite loop. Unfortunately, determining how unfair this policy is and supporting read-write semantics proved too big of a challenge to complete before the deadline.

References

[1] Tim Harris et al. “Lock Free Data Structures using STMs in Haskell”. In: *FLOPS ’06: Proceedings of the Eighth International Symposium on Functional and Logic Programming, to appear*. Apr. 2006. URL: https://www.microsoft.com/en-us/research/publication/lock-free-data-structures-using-stms-in-haskell/.