The Paper Formerly Known As "ASTM Paper"

Bryce Adelstein-Lelbach*, Steve Brandt*
*Center for Computation and Technology, Louisiana State University
blelbach@cct.lsu.edu, sbrandt@cct.lsu.edu

***Abstract*—In recent years, Software Transactional Memory has become an increasing popular solution for managing concurrency. STM has a number of advantages over traditional approaches (such as locking), but it also has the disadvantage of placing rigid regulations on code contained in critical sections. Many STM models lack mechanisms for integrating side-effecting code into applications. These restrictions limit the usefulness of STM in software that frequently interact with hardware. This paper describes a model for an STM system which supports the asynchronous invocation of side-effecting code from within a transaction. We present the details of the system and examine two use cases for our design.**

I. Introduction

Software Transaction Memory is a powerful alternative to traditonal lock-based programming. STM offers both performance benefits and stronger safety gurantees when compared to lock-based concurrency. Modern hardware has reached the physical limitations of CPU clock speeds, a watershed event that has shifted the concerns of performance-driven applications. Communication, not processing power, is now the limiting factor in many parallel applications. STM is optimistic and speculative, which is an ideal fit for modern platforms where cycles are cheap and communication is expensive. Locking can be prone to priority inversion, while STM is not. Additionally, transactional models offer stronger safety gurantees than lock-based programming. When programming with locks, application programmers must remember to acquire and release exclusive access for each relevant shared object every time they write a critical section. Programmers must be able to identify operations that will access the same shared data concurrently. Programmers who fail to properly protect critical sections will introduce subtle race conditions which are hard to detect. Because locks are pessimistic and non-speculative, programmers who are too cautious will introduce unnecessary critical sections, impeding application performance. Those who are not cautious enough may fail to protect overlapping operations. Lock-based programming requires the adoption of an application-wide protocol for obtaining locks, to prevent deadlock and livelock (the lock ordering problem). Transactions are easy to program with as they intercept and manage data access for the programmer, preventing the possibility of deadlocks/livelocks due to user error. Because of the optimistic nature of transactions, the performance penalty for overuse is less severe.

Transactional systems place restrictions on the types of operations which can be performed within a transaction. STM systems can only function properly if all mutating operations within transactions can be intercepted by the STM system. This is necessary because an STM system must be able to roll-back the effects of a failed transaction. If a transaction does not commit, it should have no effect on data outside of the internal state of the transaction. Typically, STM implementations are capable of intercepting mutating operations that operate on first-class entities of the application programming language (entities that can be stored in variables and copied). Reads and writes to variables which represent ordinary application data can be handled in a straightforward fashion as long as the there is a well-defined copy operation for the object represented by the variable. A copy operation allows the STM system to locally duplicate the data within a transaction, modify it, and then copy it back upon successful commit.

However, this restriction can prevent the use of STM in applications where copying shared data is not desirable. Software which reads from hardware typically encounter these issue. Such software frequently contains data structures which directly represent the state of hardware: e.g. hardware queues, network sockets, device buffers, file streams. All of these objects have **mutating read operations** which makes them impossible to copy without affecting them in an irreversible fashion. However, it may be possible to buffer an application specific update operation. For example, in the case of a stream object representing a file, it is not feasible to define a copy operator, but we can buffer an operation which will write to the stream.

The side-effecting restrictions of STM also makes it difficult to use when writing reusable software. Generic algorithms (implemented with templates or polymorphism) must either pass these restrictions on to their client code, or they must assume that operations on user-provided types have side-effects. For example, it may be feasible for a generic container to require side-effect free comparison operator but allow side-effecting constructors which are executed at the end of a successful commit. The insertion algorithm for the container would execute in the transaction, but the side-effecting constructor would be executed during a successful commit.

Additionally, certain classes of applications may wish to avoid the speculative execution of certain routines within a transaction. Some code may have side-effects that would not prevent its usage within an transactions, but those side effects could stress system resources. Memory allocation is the best example of this; clearly, allocating memory has side effects (possibly even at the kernel level, if the allocator needs to request additional pages). For small allocations, it is feasible to allow them to occur within transactions, provided that reference counting or memory management is used to clean up the allocation in the event of transaction failure. However, for larger memory structures it may be preferable to defer their allocation until the transaction commits successfully.

For these classes of software, we propose a feature which allows developers to express side-effecting actions which will be executed upon the successful commit of a transaction if they wish to use an STM system. We will call these actions

escaped functions. Escaped functions are written written within a transaction, and may reference the scope where they are written. Their environment is captured in a closure, and execution of these functions is deferred until the commit succeeds. Our system features a mechanism for expressing the data dependencies of escaped functions, to ensure that overlapping operations are executed safely. We present an implementation of this mechanism using C++ futures, built on top of the HPX parallel runtime system.

A. HPX

ASTM is implemented using HPX, a general purpose parallel runtime system for parallel applications of any scale. HPX exposes a homogeneous programming model which unifies the execution of remote and local operations. The runtime system has been developed for conventional architectures. Currently supported are SMP nodes, large Non Uniform Memory Access (NUMA) machines and heterogeneous systems such as clusters equipped with Xeon Phi accelarators. Strict adherence to Standard C++ [**?**] and the utilization of the Boost C++ Libraries [**?**] makes HPX both portable and highly optimized. The source code is published under the Boost Software License cite making it accessible to everyone as Open Source Software. It is modular, feature-complete and designed for best possible performance. HPX's design focuses on overcoming conventional limitations such as (implicit and explicit) global barriers, poor latency hiding, static-only resource allocation, and lack of support for medium- to fine-grain parallelism.

II. Technical Approach

The implementation of our system is written in C++, and uses an object-oriented model. Our system consists of three basic classes. **Transaction managers** represent a single transaction. A transaction object provides storage for transaction-local data, handles reads and writes from non-local values and manages commit attempts. The variables which are accessible in transactions are represented by **transaction variables** objects. These objects are containers for a single object of a user-provided type. Transaction variables expose transaction-aware interfaces for accessing the underlying object they represent. Finally, **transaction futures** represent escaped functions which have been invoked asynchronously from inside a transaction.

A. Transaction Manager

The transaction manager class is both a representation of a single transaction, and an implementation of the core services of ASTM. ASTM has no global state. Each transaction manager can be thought of as an independent **STM runtime**. Transaction managers only interact with each other when they both acquire mutual access to the same transaction variable. This interaction occurs via an underlying **concurrency runtime**, such as HPX or the C++ standard concurrency library. By using this design, we completely encapsulate the state of each transaction from all other transactions. If the internal state of one transaction instance becomes corrupted, no other transaction will be affected. In our system, a transaction block is formed in native C++ using a do-while loop and a transaction object:

```
transaction t
do {
    // Transaction block.
} while (!t.commit());
```

The transaction manager has two roles in ASTM:

- **Provides transaction-local storage**: during transaction attempts, the shared variables which are accessed by the transaction need to be recorded. The initial value of read-variables needs to be stored to check for transaction failure during the commit, and updates to write-variables need to be buffered. Additionally, any continuations launched within the transaction need to be buffered so that they can be launched on a successful commit.
- **Implement the four basic operations which occur in transactions**: **read** (copy the value of a transaction variable into local storage), **write** (buffer a write to a transaction variable), **async** (buffer an escaped function), and **commit** (attempt to commit the transaction; if successful, copy local writes to transaction variables and asynchronously evoke escaped functions). These methods are intended to be used directly by application code. Instead, transaction variables and transaction futures provide a high-level interface which is built upon these basic operations.

The transaction manager uses four data structures to implement transaction-local storage: the **variable map** which contains the local state of transaction variables and three structures which store the information needed to process a commit attempt (the **read list**, **write set** and **continuation list**).

During a transaction, all reads to transaction variables are cached and all writes are buffered, so the transaction manager must maintain a local copy of all transaction variables accessed during its' transaction. These local copies are stored in an ordered map which is indexed by the memory address of the original transaction variable associated with each local copy. The ordering of the map is important because it ensures a uniform locking order will be used by all transaction managers. Upon insertion into the map, transaction variables are deep-copied (or potentially moved in the case of a write) Actually implement move-on-write.

Whenever a transaction variable is read, the value which is read is added to a read list. These values will be compared to the transaction variables that they were read from during a commit attempt. The elements of the read list have the same key-value structure of the variable map. Likewise, write operations will add entries to a write set to keep track of which entries in the variable map will need to be written externally upon commit.

The read operation requests access to a local copy of a transaction variable. If the variable is not present in the variable map, a thread-safe read of the actual variable value will occur. This value will be added to the variable map and the read list. If the variable has been written prior to the read in the transaction, no external read is performed. Instead, the buffered

value from the write will be read from the variable map and the map entry will be added to the read list.

The write operation is mostly symmetric to the read operation. A local write to a variable that has not been previously accessed will create a new entry in the variable map, but no external read will be made. A write operation will add the variable's memory reference to the write set.

The async operation asynchronously executes a function. This operation will buffer a continuation, which will be launched asynchronously upon the successful commit of the transaction. The async operation takes two arguments; a bound function to be invoked, and a reference to the future object which the operation will wait on (if this reference is null, the function will not have any dependencies). The future object can be thought of as a queue. When a transaction commits, escaped functions are attached to their futures as continuations. When the action associated with the future becomes ready, the continuation will begin immediately.

The commit operation attempts to complete the transaction. First, the elements of the variables map are iterated, and exclusive access to the transaction variable is obtained. Then, the read list is iterated and each entry of the list is compared to the value of the corresponding transaction variable. If one of these comparisons fails, the transaction will fail. In the case of failure, the internal state is purged and the commit routine returns immediately. If no discrepancies are found, the commit succeeds. The transaction manager will update every transaction variable in the write set with its local value. Next, the continuations which have been buffered by the async operation are enqueued. Finally, exclusive access will be released and the internal state of the transaction manager will be cleared.

B. Transaction Variables

Transaction variables can be thought of as "wrapped" instance of of their underlying type (e.g. an "is-a" relationship with the underlying type). Transaction variables have the following functions in our system:

- Safely type-erases objects that are used in transactions. Transaction variables is part of a type erasure system which enables a type-agnostic implementation of the transaction manager.
- Provides an interface to the user for accessing and updating variables inside and outside of transactions.
- Enforces the invarants regarding variable access in our system.

Transaction variables expose most of their functionality through **local variable** objects. A local variable is a proxy object that holds a memory reference to a transaction variable and a memory reference to a transaction. Local variables are representations of the value of a transaction variable within a given transaction. These local variables use a smart pointer interface. Dereferencing a local variable will produce an object that can be assigned to; such an assignment is implemented using the write operation of the transaction manager. The structure dereference operator can be used to invoke non-mutating methods of the underlying type, and the indirection operator can be used to produce a constant reference to an object of the underlying type. Both of these operations are implemented using the read operation of the transaction manager. A syntax table for local variables is provided below.

Syntax	Effect
`auto A_ = A.local(t)`	Create a local variable `A_` which represents the state of the transaction variable `A` (with underlying type `X`) in transaction `t`
`A_->f()`	Invoke the `const` member function `f` of the underlying object represented by `A_`
`*A_ = b`	Perform a write operation (using transaction manager `t`) assigning `b` to `A_`
`X x = *A_`	Perform a read operation (using transaction manager `t`) on `A_`

C. Transaction Futures

A transaction future representing the result of a computation which has been "escaped" from the transaction. They are created with the async operation. Transaction futures can be thought of as a representation of a value that may not be known yet because its computation has not completed execution. Transaction futures are used to integrate side-effecting and computationally expensive code into transactions. A function invoked via the async operation will have the following properties:

- Execution of the function will only occur if the transaction commits successfully.
- Execution will only occur after all other effects of the transaction have been written back to shared storage.
- Local variables that are passed to or referenced by the function will be copied when the function is passed to the async operation; e.g. the function will execute in a closure obtained from within the transaction.
- The function will not execute inside of a transaction.

Escaped functions have no restrictions on what operations they may contain, as long as they do not violate the invariants regarding transaction variable access (e.g. transaction variables may only be accessed concurrently from within a transaction). Escaped functions may contain operations that have side-effects or operations which are considered too time consuming to be executed within a critical section.

However, the async operation introduces potential concurrency issues which must be addressed. If two transactions which are running concurrently each launch an escaped function which accesses the same non-transaction, non-thread-safe variables (e.g. instances of regular C++ data structures not synchronized with ASTM), it is clear that race conditions may arise. This is problematic because escaped functions are most useful if you can seperate your data into two categories: objects accessed only through transactions (represented by transaction variables) and objects that are accessed only through escaped functions.

To solve this problem, we take advantage of the composability of futures. C++ futures allow the construction of dependency chains. Instead of waiting for the value of a future to become ready, application code can pass a callback function

to the future which will be invoked when the data is ready. The operation that registers such a callback is called `then()`, and it returns another future (representing the execution of the callback), which enables chaining. Another operation, `when_all()`, allow the composition of multiple futures into one future, which can then have a callback attached to it in the same fashion.

The async operation in ASTM makes use of the `then()` method to launch escaped functions. Application code provides a future as an argument to the transaction managers async operation. If this future is null, then the escaped function does not have any dependencies, and after the transaction commits successfully it will be scheduled without constraints. If the future is not null, then the escaped function will be attached to it as a callback. In either case, the async operation will create a new future during a successful commit; this new future will be assigned to the argument passed to async within the transaction (the operation is similar to the effect of compound assignment operators such as +=).

This functionality allows escaped functions to access shared data safely. As mentioned, applications using escaped functions will divide their data into two categories: objects accessed only from within transactions, and objects accessed only from escaped functions. The shared objects accessed through escaped functions are represented by future objects in the application codes. When an escaped function needs to access one of these objects, it will be attached to the object as a callback function. As long as all shared data accessed by escaped functions is passed by futures, concurrency will be guranteed.

III. Applications/Results

A. *Binary Tree with Side-Effecting Constructors*

IV. Future Work/Conclusions

Acknowledgment