Software Transactional Memory for Dynamic-Sized Data Structures

[Review]

Aivar Annamaa
Institute of Computer Science
University of Tartu
aivarannamaa@hotmail.com

Abstract

Software transactional memory (STM) is a programming abstraction for organizing access to shared mutable state in concurrent programs by dividing work into atomic units, called transactions. Dynamic STM builds on this idea by allowing extra runtime decisions about transactions and memory usage, therefore it is useful for working with dynamic-sized data structures.

This review first introduces STM in general and then describes first influential approach to dynamic STM (DSTM) as given in [1]. Distinctive features of this system are obstruction free progress and pluggable contention management. We give brief overview of the system and evaluate its usability and published quantitative results.

1 Introduction

Most prevalent technique for managing concurrent access to mutable state is locking – i.e. disallowing simultaneous access to certain data by blocking one thread’s flow until another thread has released the data. While locks are conceptually simple and easy to implement, their usage in larger programs causes several difficulties: coarse-grained locks don’t scale well as threads may spend lot of time for waiting; on the other hand, fine-grained locks are hard to get right because it is difficult to reason about heavily interleaved modifications.

Software transactional memory [2] provides an alternative to lock-based access control. With STM, programmer organizes concurrent tasks (which may interfere with each-other) into units of work called transactions, and STM runtime guarantees that each transaction is performed atomically, i.e. it either succeeds wholly or its (partial) effects are wholly undone (aborted). When STM detects that two concurrent transactions try to modify same data, it aborts one of the contending transactions. Therefore STM is similar to optimistic locking – extra synchronization measures are used only after conflict is actually detected.

An important aspect of transactional memory is that partial effects performed during the transaction are not visible outside this transaction until the transaction is committed. This means that one transaction can’t interfere with another transaction running in another thread and therefore programmer can use simpler sequential reasoning for understanding statements inside transactions.

First STM systems required both the memory usage and the transactions to be statically known. DSTM allows creating new transactions and transactional objects according to information available at runtime.

Isolating transactions from each-other must satisfy correctness requirement: the exact interleaving of steps done in different threads does not affect the result (the program doesn’t do wrong thing). Equally important question to every concurrent system are its progress properties (it should finally do the right thing). DSTM’s synchronization mechanism is obstruction-free, meaning each thread can achieve its goal if other threads don’t work against it.

Another progress issue is how to deal with conflicts. DSTM basically sidesteps this topic by allowing user to provide custom contention managers suitable for particular situation.

In next sections we explain most important features of the DSTM system according to Java implementation developed by the authors. We conclude by evaluating the system both by subjective and objective measures.

2 Overview

Central notion in DSTM is transactional object of class TMObject, which acts as a special container for a regular Java object, intended to be accessed concurrently. In following example, a mutable Java object (counter) is wrapped into a transactional object:

```java
Counter counter = new Counter(0);
TMObject tmObject = new TMObject(counter);
```

Transactional object is meant to be shared between threads, but in order to access its content, thread must explicitly open the container in the context of a transaction. Opening returns a Java object, called version, which corresponds to original wrapped object but is visible only to given thread, therefore no further synchronization is required for manipulating the version. After thread has finished manipulating objects opened in this transaction, it tries to commit the effects produced. If, in the meanwhile, other threads have opened any of those transactional objects, then commit fails, modified versions are abandoned and thread has opportunity to try again same work in new transaction.

```java
while (true) {
    thread.beginTransaction();
    // get private version of counter
    Counter cl = (Counter)tmObject.open(WRITE);
    cl.increment();
    if (thread.commitTransaction()) {
        break;
    }
}
```
As seen from the example, open method takes the opening mode as the argument. When thread only needs to read a value, then it is useful to open the container in READ mode, as concurrent read operations from different threads are not considered to be in conflict. As efficiency measure, DSTM also allows releasing objects opened in read mode before the end of transaction, allowing other threads to start modifying them without conflict. This feature is unfortunately unsound – it may break correctness property when used improperly.

3 Implementation

One of the main challenges in implementing STM is maintaining the impression that commit or abort updates “instantly” all transactional object opened in given transaction (i.e. other threads must not see old versions of some objects and new versions of others). In DSTM this is solved by using 2 levels of indirection.

Figure 1 describes the structure of transactional object. Each such object has three logical fields which are combined into a separate object (Locator), so that changes to those fields can be published atomically by updating respective pointer (start). The current version of transactional object is determined by the start pointer and the status of transaction object in transaction field of the respective Locator. If the transaction is committed, then current version is in new object otherwise it’s in old object.

When a transaction opens a transactional object (let’s assume previous transaction which accessed this object has committed), then it first creates new version of Locator, where it sets transaction field to itself and old object field to old Locator’s new object field. Field new object is linked to a clone of old Locator’s new object. The clone is returned as result of open operation and it’s visible only to this transaction. When transaction is done manipulating all it’s private clones, it initiates commit by linking start pointers of its opened objects to new Locators. But according to rules, the current version will be still old object until transaction’s status is uncommitted. When all links are updated, then transaction’s status is set to committed and this atomically makes all new versions to be current versions.

A transaction is allowed to request abortion of another transaction that uses same transactional objects. This feature gives DSTM obstruction freedom property – thread can eventually succeed when other threads don’t counteract. It’s weaker property than lock freedom, but easier to implement and reason about. Conflict resolution is delegated to user-provided contention managers. As an example, authors describe “agressive” contention manager that always allows aborting other threads and “polite” that denies first attempts of abortions.

4 Results

For evaluating the solution empirically, authors implemented a linked-list based integer set using DSTM and for comparison, another based on simple locking. They measured how many insert and delete operations per second those data structures allow, configured with varying number of threads, with different contention managers and with/without early releases. They also tried a red-black tree instead of linked-list. Measurements were conducted on a server with 72 1050MHz processors. Results are shown in figure 2. Measurement for 1-thread lock-based case was mentioned only in the text – 768 operations/millisecond.

With single thread, simple lock-based solution is clearly most efficient. Performance of DSTM solutions is mostly affected by the amount of contention (early release reduces number of conflicts considerably). “Aggressive” contention management works better only with lot of contention, but “Polite” is more efficient in situations with reduced contention. Results also show that extra complexity of red-black tree pays off in better performance.

5 Conclusion

According to simple linked-list example given in the original paper, using DSTM is not more complex than programming with locks. Unfortunately authors don’t describe their experience with implementing red-black tree, it would have been interesting to know eg. how big was its source code.

Performance measurements hint that DSTM pays off only with large number of processors – lock based implementation running in single thread beats even 10 processors with DSTM. It’s clearly necessary to reduce overhead of proposed system (possibly with some extra static program-analysis).

6 References
