

Supporting Existing Code in Transactional Memory Systems

Nathaniel Nystrom and Mukund Raghavachari

IBM T. J. Watson Research Center
{nystrom,raghvac}@us.ibm.com

Transactional memory (TM) [3] is a simple abstraction for concurrent programming that affords several advantages over lock-based synchronization: deadlock-freedom, ease-of-use, and composability. Transactional memory systems need to interact with existing code: transactional code like databases, and non-transactional code that performs messaging, synchronization, and I/O. In the DALI (Data Access Language Integration) Project, we are investigating programming languages features for integrating existing code into a software transactional memory system. software transactional memory system. We focus in particular on interacting with existing transactional models.

Challenges

Accessing existing libraries presents a number of challenges that can complicate the simple transactional programming model provided by TM or can constrain the TM implementation.

I/O. When one transaction conflicts with another, it may be aborted and rolled back; however, some operations, such as I/O, cannot be undone or cannot redone. Common solutions to this problem are to delay I/O until the transaction commits or to disallow I/O within a transaction altogether. However, these solutions are not always feasible or even possible. For instance, a library function might be implemented as a remote procedure call or might access a database.

Abstraction violations and inconsistencies. Library state may become inconsistent with state managed by the TM. This can happen when there are accesses to the library not mediated by the TM. For example, a persistent store may become inconsistent with the transient in-memory state managed by the TM if another process updates the store. A programmer may invoke library functions that do not fit into the transactional model. For example,

A mechanism is needed to enforce consistency guarantees between the library state and the transactional memory. Developers should be able to specify their desired consistency guarantees.

Other transactional models. Some libraries—JDBC, for example—support their own notion of transactions. Integration of disparate transaction models into the same framework enables a simpler programming model. Issues such as isolation levels and different transaction implementations make this integration particularly challenging.

Synchronization. Non-transactional libraries may perform their own concurrency control, leading to deadlock, priority inversion, and other problems [1]. In general, transactional languages should allow programs to use both TM—for expressiveness—and explicit locking—for performance and interaction with existing libraries.

The interaction of explicit locking and TM needs to be investigated further. This interaction is strongly dependent on both the library and the TM implementation. The library may not be thread-safe. With optimistic TM implementations, conflicting library calls may be made and may go undetected at transaction violation.

TM implementation. Access to existing code can constrain the TM implementation. For example, TM can be implemented using a *write-back* approach—writes are buffered until commit, when the heap is updated—or a *write-through* approach—writes update the heap immediately, but are logged so they can be rolled back if the transaction rolls back. With the write-back approach, references passed into a library need to be prevented from reading stale values. With a write-through approach, library code that performs a write to TM-managed locations must do the appropriate logging, or the library call itself must be logged for undo. In our work, we assume a write-through policy.

Using open nested transactions to support I/O imposes further restrictions on the TM implementation.

DALI

In the DALI Project, we address the above issues by introducing new programming languages features that allow programmers to adapt existing code to a transactional setting.

Open (trans)actions and compensations. To support interacting with non-transactional code, we provide `open` and `open atomic` statements.

The statement `open S` executes `S` without transaction management: no locking or logging is performed. The statement `open atomic S` executes `S` as an open nested transaction.

An optional `undo` clause, or *compensation* [5, 2], allows the programmer to specify how a the corresponding `open` statement should be rolled back. As the transaction executes, the compensations are appended to the log. If the enclosing transaction is aborted, the clauses are executed in reverse order, returning the system to its original state (or an equivalent state).

```
open {
  c.executeUpdate(
    "INSERT into orders VALUES (" + oid ... + ")");
} undo {
  c.executeUpdate(
    "DELETE from orders WHERE oid = " + oid);
}
```

Compensating expanders. To make compensations easier to use, we use *expanders* [4] to wrap classes in the library with compensating code.

```
expander UndoableList of List {
  public void add(Object x) {
    open { inner.add(x); }
    undo { inner.remove(x); }
  }
  ...
}
```

An expander extends and overrides the type it expands. When a compensating expander for a type T is in scope, calls to methods of T invoke the expander instead.

Transaction event listeners. To adapt existing transaction models to the TM implementation and to make existing non-transactional code transactional, we introduce *transaction event listeners* to register code to be invoked when an event such as transaction begin, commit, or rollback occurs.

In DALI, we have used transaction listeners to integrate JDBC transactions into a transactional memory system; the listener records savepoints for open JDBC connections on transaction begin and rolls back to these savepoints when the TM rolls back.

Status

We have implemented these features in an extension of AtomJava that supports experimentation with composition of transactional and non-transactional code with AtomJava's software transactional memory system. Advanced developers can use our system to develop custom transactional semantics; we have used the language to develop a transactional SQL library. We are currently working through the semantic foundations of the language.

References

- [1] Brian D. Carlstrom, JaeWoong Chung, Hassan Chafi, Austen McDonald, Chi Cao Minh, Lance Hammond, Christos Kozyrakis, and Kunle Olukotun. Transactional execution of Java programs. In *OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOOL)*, October 2005.
- [2] Hector Garcia-Molina and Kenneth Salem. Sagas. In *Proceedings of the 1987 ACM SIGMOD International Conference on the Management of Data*, pages 249–259. ACM Press, May 1987.
- [3] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the 14th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 204–213, August 1995.
- [4] Alessandro Warth, Milan Stanojević, and Todd Millstein. Statically scoped object adaptation with expanders. In *Proceedings of the 21st ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOSPLA)*, Portland, OR, October 2006.
- [5] Westley Weimer and George Necula. Finding and preventing runtime error handling mistakes. In *Proceedings of the 19th ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOSPLA)*, October 2004.