Brief Announcement: Actions in the Twilight
Concurrent Irrevocable Transactions and Inconsistency Repair

Annette Bieniusa
University of Freiburg, Germany
bieniusa@informatik.uni-freiburg.de

Arie Middelkoop
Universiteit Utrecht, The Netherlands
ariem@cs.uu.nl

Peter Thiemann
University of Freiburg, Germany
thiemann@informatik.uni-freiburg.de

ABSTRACT
Twilight STM enhances a transaction with twilight code that executes between the preparation to commit the transaction and its actual commit or abort. Twilight code runs irrevocably and concurrently with the rest of the program. It can detect and repair potential inconsistencies in the state of its transaction and may thus turn a failing transaction into a successful one. Moreover, twilight code can safely use I/O operations while modifying the transactionally managed memory.

The Twilight STM keeps a pending transaction committable while running the twilight code, but without blocking all other transactions, including pending transactions that execute their twilight code at the same time. Benchmark results show that Twilight performs competitively with state-of-the-art systems like TL2.

Categories and Subject Descriptors
D.1.3 [Programming Techniques]: Concurrent Programming

General Terms
Algorithms, Performance

Keywords
software transactional memory, irrevocable transactions, consistency

1. INTRODUCTION
Even newly developed multi-threaded applications written from scratch cannot rely solely on Transactional Memory primitives as the code has to interact with synchronization protocols buried in system calls, libraries, and legacy code. These legacy code bases rely on locking to implement concurrent data structures or to safely perform I/O and other system services. As neither locking nor non-reversible operations are compatible with transactions, a flexible mechanism is needed that makes these worlds safely coexist. Recent proposals in this direction introduce the notions of irrevocable and inevitable transactions [4, 6]. Similar to earlier proposals [5], they have severe limitations because they serialize transactions which have non-reversible side-effects with a global lock.

This paper introduces the Twilight STM [1], an STM implementation that splits the commit operation of a transaction in two phases, a prepare to commit, which checks the preconditions for a successful commit and isolates the transaction from other concurrently running transactions, and a finalize commit, which makes the outcome of the transaction globally visible. As its main novel feature, Twilight allows the programmer to augment a transaction with twilight code that runs between the commit preparation and the actual commit.

Twilight code is substantially different from the commit and abort hooks of TM systems implemented in hardware. While these hooks run in a fixed order after the STM implementation has decided on the transaction’s fate (commit or abort), the twilight code executes before that decision is taken and can affect its outcome. To this end, the Twilight API has operations to detect and repair read inconsistencies (a feature unique to the Twilight STM). The finalize commit operation only finishes the transaction successfully if the twilight code resolved all inconsistencies, otherwise it restarts the transaction.

Furthermore, the twilight code is free to perform external operations such as invoking system services or using handshaking- and lock-based protocols (e.g., synchronization barriers). These external operations are immediately and irrevocably globally visible. To maximize applicability, twilight code may run concurrently with other transactions including their twilight code. While the Twilight STM guarantees atomicity, twilight code may run concurrently with the Twilight API, the programmer is still obliged to ensure freedom of deadlock and races for the external operations. However, the API guarantees that the twilight code does not affect the transactional properties of the transaction bodies.

2. THE TWILIGHT STM AT WORK
Fig. 1 illustrates the correct use of the Twilight API to attach an I/O operation to a transaction. It is a fragment of a program that spawns a number of worker threads that perform operations on shared memory using transactional reads and writes. To trace the application’s execution and to measure its progress, a global variable counter assigns each transaction a number indicating at what position in the commit order the transaction succeeded. As every transaction reads and updates the counter, its presence causes read inconsistencies in many transactions, each of which would have to abort and restart (i.e., spin-locking).

Instead of blindly aborting the transaction, the twilight code first updates the shared variables to a consistent state (stm_reload) and checks if the counter was the only problem while validating the read set (and restarts otherwise: stm_restart). Any operation dominated by stm_reload and from which no stm_restart is reachable, is sure to be part of a successful commit and hence safe for performing I/O.

After the stm_reload, the program repeats the update of the counter state in the transaction’s write set. As the transaction cannot fail anymore, it is safe to output a log message. The resulting output trace contains the log messages in their commit order.
volatile int counter;

void threadWorker()
{
    // *** transaction body begins ***
    stm_begin();
    // *** transaction body begins ***
    int pos = stm_read(counter) + 1; // update counter
    stm_write(counter, pos);
    stm_leave_region();
    bool succeeded = stm_prepare();
    // *** transaction body ends, twilight code begins ***
    if (succeeded) |
        stm_reload();
        // update variables to consistent state
        if (stm_only_inconsistent(COUNTER)) |
            pos = stm_read(counter) + 1; // update counter
        stm_write(counter, pos);
        else |
            stm_restart();
    |
    printf("TX %i finished at %i,\n", tid, pos);
    stm_finalize();
    // *** twilight code ends ***
}

Figure 1: I/O and locking in twilight code.

This is not guaranteed when moving the output statement out of
the transaction.
Alternatively, the counter could be handled as a shared variable
outside the control of the transaction using mutexes or other kinds
of locking for guaranteeing exclusive access. Such code could also
be placed in the twilight code and would yield the same results. Of
course, in this case, it is the programmer’s responsibility to avoid
data races, deadlocks, and other problems related to these synchro-
nization primitives.
The example program also makes good use of consistency re-

ions, another facility of the Twilight API. This concept permits
the easy correlation of inconsistencies with (groups of) variables.
At each time during the execution of a transaction, there is an ac-
tive region marker, which is administered in a stack-like manner.
Every access to a variable is tagged with the currently active region
marker during the transaction’s execution. In the twilight code,

consistencies can be checked conveniently by region as demon-

strated with $\text{stm}$ only inconsistent. Furthermore, it can give

 insight in congestion points of the application.

2.1 Properties
Twilight STM provides operations to the programmer which re-
lax the isolation of transactions in a controlled way. The API opera-
tions rely on a locking protocol that maintains the following invari-
ants:

1. Transactions that execute twilight code concurrently have dis-
   joint write sets.
2. A transaction has exclusive access to the variables in its write
   set from $\text{stm}$ prepare to $\text{stm}$ finalize.
3. The Twilight API is deadlock-free.

The protocol further guarantees that out-dated values are con-

sistently re-read. Results from computations on them can only be
out-dated, but not wrong. Furthermore, the transaction can also ob-
serve at commit the current values and has a chance to adjust its
results.

3. EVALUATION
To show the competitiveness of Twilight STM, we compare the
performance of a C implementation of Twilight STM (both with
and without twilight code) with the TL2 reference implementa-
tion [3]. We added twilight code to the vacation application from
the STAMP benchmark [2] to ignore benign restructuring of hash
maps and to recalculate the amount of available items instead of
restarting the conflicting transactions. We further increased the
contention by performing a higher number of actions per transac-
tion and acting on a lower number of different items.

Fig. 2 displays the average taken from 10 runs of the bench-
mark. TL2 aborts in its preparation phase as soon as it discov-
ers that a variable to be read is locked. In contrast, Twilight STM
waits until the variable’s lock is released as it might fix the arising
inconsistency in the twilight code. This strategy results in signif-

icantly fewer aborts and leads to marginally better performance in
case of high contention. The ratio of aborts to saves is approxi-

mately 100:830 for 2 threads, 100:270 for 4 threads and 100:115
for 8 threads. Furthermore, the transactions perform more work,
so that restarting a transaction introduces a substantial loss in per-
formance. The execution time of TL2 and Twilight STM without
twilight code is comparable. However, Twilight STM with twilight
code is significantly faster.

4. REFERENCES
Concurrent irrevocable transactions and inconsistency repair (extended version).
the 20th International Symposium on Distributed Computing, DISC 2006,
software transactional memory. In TRANSACT ’08, 2008.
transactions with locking for Java synchronization. In 20th ECOOP, volume
applications. In SPAA ‘08: Proc. Twentieth Annual Symposium on Parallelism in
ACM.