Actions in the Twilight: Concurrent Irrevocable Transactions and Inconsistency Repair (Extended Version)

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

Software Transactional Memory (STM) allows grouping of accesses to shared memory in concurrent transactions, thus ensuring their atomicity, consistency and isolation.

The Twilight STM introduced in this paper 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 read 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.

1 Introduction

Transactional Memory (TM) is a promising alternative to lock-based synchronization in multi-threaded programs. It improves productivity by offering fine-grained mutual exclusion while providing modularity, scalability, and composability [25]. High-performance implementations of TM are being developed in hardware and software that are competitive with systems implementing other concurrency paradigms.

However, even new code in multi-threaded applications written from scratch cannot rely solely on TM because the code has to interact with synchronization protocols buried in system calls, libraries, and legacy code. These code bases rely on handshaking and 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 [28, 33]. Similar to earlier proposals [32], they have severe limitations because they serialize transactions which have non-reversible side-effects with a global lock. The serialization requirement rules out handshaking-based synchronization protocols and locking inside of transactional code as further elaborated in Section 2.

This paper introduces the Twilight STM, 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([16, 19, 30]). 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* like 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, consistency, and isolation for transactional code and the Twilight API, it cannot prevent misuse of the twilight code. That is, the programmer is 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. Hence, twilight code can implement highly sophisticated contention management strategies, which may depend on the outcome of external operations.

We have developed two implementations of Twilight STM, a Java implementation and a C implementation that is based on the TL2 algorithm. We have run benchmarks on the C implementation. The results show that twilight code does not introduce extra overhead on top of TL2. On the contrary, utilization of the repair mechanism reduces the abort rates and improves run-time performance.

Contributions. The unique feature of Twilight STM is its facility to augment a transaction with (twilight) code that can influence the outcome of the transaction by attempting to repair inconsistencies. It is able to interact with globally visible side-effecting operations and traditional concurrency control mechanisms.

1. We describe our design of the Twilight STM API, which separates the commit operation of a transaction in two phases, commit preparation and commit finalization. It provides the means to run twilight code in between with a number of unique features.

- The twilight code of a transaction can execute concurrently with any other kind of thread: outside of transactions, inside of transactions, or the twilight code of another transaction.
- Standard concurrency control mechanisms, as well as I/O and system calls are available in the twilight code.
- After a commit preparation, the twilight code can access and modify the state of its pending transaction. It can turn a failing transaction into a successful one by resolving inconsistencies detected during a commit preparation.

2. We define the semantics of a transaction with twilight code, which partially relaxes the isolation property. We prove the transactional properties.
3. We provide two implementations of the Twilight STM. The code is available on the web[1].
4. We present experimental results that show how twilight code can be used to advantage for application specific contention management.

Outline. After discussing related work in Section 2, we motivate the need for the Twilight STM with examples in Section 3. Section 4 introduces Twilight's API concentrating on the operations available to twilight code and Section 5 a detailed explanation of the algorithm. Section 6 defines a semantics for Twilight STM with proofs of transactional properties. Section 8 presents the implementations and evaluates some benchmarks before concluding in Section 9.

2 Related Work

TM has a huge design space which is investigated by numerous researchers.

One key issue is the choice between an implementation in hardware (HTM) [1, 8, 18, 21], an implementation in software (STM) [5, 10, 11, 13, 23, 26]. , or a hybrid approach [4, 14].

HTM Approaches. Although our work is a software-only approach, there are quite a few related HTM-based efforts that suggest mechanisms to execute irrevocable operations inside a transaction.

Blundell et al. [2] simulate a hardware TM design that supports I/O and system calls within transactions. They allow only one unrestricted transaction, which can perform I/O and system calls, running concurrently with multiple restricted transactions, which cannot perform I/O. This choice limits concurrency.

Moravan et al. [19] simulate a nested TM in hardware. Their transactions may contain "escape actions" that execute code outside the transactional scope. To retain the transactional appearance, escape actions can register actions to run on a successful commit or on restart (compensating actions). Such commit and compensation actions are also needed in their implementation of open nesting of transactions.

McDonald and coworkers [16] define an instruction set architecture for HTM. Their architecture includes commit handlers and violation handlers that can continue a transaction in an arbitrary way after a conflict. However, only the validated outcome of a transaction is visible. There is no provision to compare previously read values with the current ones.

Volos et al. [30] analyze the problems of using locking operations with TM. They implement transaction-safe locks on top of a HTM. Their design includes commit actions to obtain locks and compensating actions to release them. Further tasks can be scheduled at conflicts and escape actions are also supported.

[1] `http://proglang.informatik.uni-freiburg.de/projects/syncstm/`

Conflict Avoidance. Harris and Stipić [12] implement the concept of an abstract nested transaction (ANT) in a software TM. Failure of an ANT does not cause failure of the enclosing transaction, instead the ANTs are retried when the enclosing transaction is ready to commit. Side effects like I/O and system calls are disallowed inside ANTs. ANTs can be implemented with the Twilight API by performing potential retries in the twilight code.

Ramadan and colleagues [22] enable conflicting transactions to commit. They introduce the notion of dependency-aware TM where a value written by one transaction is forwarded to another transaction reading the same variable before the actual commit, thus avoiding a conflict between the transactions and achieving significant speedups. Twilight STM can also deal with this kind of conflicts by first letting the writing transaction run to completion and then relying on twilight code to fix the conflicting read in the other transaction.

STM and Irrevocability. Most closely related to our proposal are extensions of STM with irrevocability and inevitability, respectively.

Welc et al. [33] propose an irrevocability mechanism to support flexible contention management and the execution of non-reversible actions (I/O, system calls, precompiled libraries, ...) within transactions. To ensure safety, they use a protocol with single-owner read locks. A transaction becomes irrevocable by executing a special statement that tries to acquire locks on the read set so far and all upcoming reads. This approach enforces that only a single irrevocable transaction can run at a time to avoid deadlocks. The system is implemented as an extension of the Java-based McRT-STM and uses dynamic method dispatch to enforce the correct usage and interaction with other language constructs.

Similarly, Spear and coworkers [28] compare five mechanisms which all require that at most one irrevocable transaction runs at a time and that these transactions do not abort (they use the term inevitability instead of irrevocability).

Twilight STM introduces a notion of irrevocability, too. The external operations performed in twilight code cannot be restarted: if the corresponding transaction has to be restarted, the programmer must first compensate for such operations explicitly. The main difference is that Twilight allows arbitrarily many transactions with possibly irrevocable actions to run concurrently.

Ziarek et al. [34] propose a framework where Java's synchronization monitors can be freely mixed and combined with atomic blocks. Their system optimistically tries to execute all concurrency primitives transactionally. In situations where such an execution is infeasible, the implementation irrevocably switches the execution of the concerned critical region to monitors as specified in the original program and to a global locking approach for user-defined transactions. Twilight STM does not allow an arbitrary mixing of synchronized and atomic blocks. It restricts the use of non-transactional synchronization primitives to the twilight code and never reverts to global locking. The default case is transactional execution.

Welc and coworkers [32] propose a dual-mode implementation of monitors for Java which switches at run time between a lock-based implementation and a transactional one. The switch is based on the level of contention where high contention triggers the use of the transactional implementation. To integrate irrevocable actions, they rely on the same mechanism as in the previously described paper [33]. Here, mode switching is handled transparently to the programmer. Twilight does not switch modes but offers integration with lock-based code in the twilight code. It further reduces the conflict potential of transactions by allowing to inspect and repair read conflicts.

Harris [9] proposes a mechanism to integrate exceptions and side effects with transactions. As a precursor to ANTs, the proposal relies on a transaction being able to register external actions that execute at commit-time of the transaction.

Smaragdakis and coworkers [27] integrate side effects in a transaction by essentially committing before and resuming the transaction afterwards. Volos and coworkers [31] propose a system call interface for use

```
volatile int counter;

void threadWorker() {
  stm_begin();                                         ///// *** transaction body begins ***
  // complex computation involving STM operations omitted

  stm_enter_region(COUNTER);
    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.

inside transactions. They perform error checking as early as possible and defer the main task of the system call to commit time. They rely on sentinels to manage concurrent access to resources. Programs have to be rewritten to use the new interface.

3 The Twilight STM at Work

A range of applications can benefit from the features of the Twilight STM as demonstrated in this section. The first example combines several twilight features. It uses the twilight code to attach I/O operations to transactions and it shows how to turn a failing transaction into a successfully committing one. The second example considers a more realistic setup that combines transactions with I/O operations and locking. The last subsection surveys further scenarios where twilight code can be used to integrate I/O operations in a semantics preserving safe manner, to debug applications written in the transactional memory paradigm, or to increase the overall performance.

All code fragments are presented in a C-like pseudo code.

3.1 Counting Committed Transactions

Operations that perform non-transactional side effects (e.g., I/O) do not mingle well with STM, despite some partial successes [6, 7, 24]. The problem is that it is impossible to revoke and restart I/O operations executed inside a transaction if that transaction fails to commit. For the same reason, it is not possible to run other concurrency control protocols like locking inside a transaction. A common approach is to decouple I/O operations and STM operations completely and to defer I/O operations after the successful commit.

The Twilight STM allows I/O operations as well as locking-based protocols to be placed in twilight code. The code in 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 to speed up a large computation. Each worker thread runs a separate transaction that performs some 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).

In this situation, twilight code comes to the rescue. 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. This condition is easy to check and it holds for the example.

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. Moving the output statement out of the transaction, either explicitly or by means of commit hooks, does not guarantee this property.

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 synchronization primitives.

The example program also makes good use of *consistency regions*, another facility of the Twilight API. This concept allows the easy correlation of inconsistencies with (groups of) variables. At each time during the execution of a transaction, there is an active region marker, which is administered in a stack-like manner (cf. `stm_enter_region` and `stm_leave_region` in Fig. 1). Every access to a variable is tagged with the currently active region marker during the transaction's execution. In the twilight code, inconsistencies can be checked conveniently by region as demonstrated by `stm_only_inconsistent`.

3.2 External Locking Protocols

Consider a banking application where transactions are running concurrently to transfer money from one client's account to some other accounts. A transaction may contain multiple transfers and multiple transactions may be running simultaneously on behalf of a single client. For performance reasons, all transactions operate on a fast in-memory model. For durability, a summary of the outcome of each transaction is also logged to permanent storage using one dedicated file per client. After a crash, the system should be able to reconstruct its state from these files.

Additionally, the bank runs a background task that regularly compares the state on disk with the state in memory to detect data corruption caused by faulty hardware or programming errors. As the transfer information in memory contains the file pointer to its summary on disk, a validation is possible in both directions. The background task concurrently picks a client `c`, reads `c`'s data from memory, scans `c`'s file, and checks for mutual consistency.

The major challenge in this scenario is that reading the memory and the file must form one atomic action, while other transactions can still execute concurrently and without deadlocking. This otherwise complex problem has a fairly straightforward solution in Twilight STM.

Figure 2 contains the code for a transaction that performs some money transfers. After transferring the money by reading and writing to the memory, it writes a summary to the file, obtaining location information that is also written to the in-memory model. There are two subtleties involved in this code. The `stm_lock` operation may involve waiting for other transactions to complete their twilight code. As other transactions may have committed between `stm_prepare` in the transaction body and the `stm_lock`, the latter may discover new inconsistencies. The return value of the `stm_lock` operation indicates the presence of such inconsistencies, and the programmer has to deal with them. The second point is that storing the location information must be prepared with a dummy location in the transaction body because the twilight code must not write to new global variables. This dummy value is never visible outside the transaction. Section 4

```
void transactMoney(client c, t_info[] transfers) {
  stm_begin();

  // change in-memory model
  for (t_info t: transfers) {
    transfer(c, t.recipient, t.amount);
    add_to_summary(t.recipient, t.amount);
  }
  store_location_info (c, 0); // dummy value

  stm_prepare();

  // obtain a lock on the client's file
  bool consistent = stm_lock(file_lck);

  if (!consistent) {
    stm_unlock(file_lck);
    stm_restart();
  }

  int loc = write_summary_to_file(c);
  stm_unlock(file_lck);

  // update in-memory model
  store_location_info(c, loc);
  // complete the transaction
  stm_finalize();
}
```

```
void transactVerify(client c) {
  stm_begin();
  read_client_data(c);
  stm_prepare();

  bool consistent = stm_lock(file_lck);

  if (!consistent) {
    stm_unlock(file_lck);
    stm_restart();
  }
  assert_consistent_with_file(c);
  stm_unlock(file_lck);

  stm_finalize();
}
```

Figure 2: Banking transactions.

explains these special Twilight API functions in detail.

The code for background verification task follows a similar pattern. The file locks taken in the twilight zone ensure that the entire transaction forms an atomic with the file access. Hence, the client's data in memory matches the contents of the file. This code thus fulfills the application's requirement.

3.3 Further Scenarios

Complex applications Distributed systems, like multi-player online games, require a multitude of different concurrent tasks. To ensure scalability, every participating node is responsible for modeling a fragment of the world consisting of a number of entities (e.g., monsters) that act autonomously in the virtual world. Furthermore, every node drives a GUI and processes commands entered by the player. Finally, the node receives status update messages from neighboring nodes.

Sweeney [29] suggests that each of these tasks can be assigned to its own thread that modifies the internal state of the node, which is shared between the threads. However, each thread also sends information about the state of its monster to the neighboring nodes. A mutual exclusion protocol with semaphores guarantees safe multiplexing of different TCP connections. The communication should not happen outside the atomic region to prevent the transmission of inconsistent state information.

In this scenario it is essential that multiple threads can execute the communication phase in the twilight code at the same time, as a game model may contain 10,000 active game-play object and maintain connections to 10-20 neighbors. The Twilight STM may be employed to obtain a high degree of concurrency.

Debugging As the Twilight STM guarantees the atomic execution of the I/O actions in combination with the corresponding memory operations, it is particularly suited to debugging STM applications. For example, we used it in our benchmark programs to determine the reasons for restart for memory locations with high contention.

Lock-based protocols Although contrary to the spirit of STM, it is sometimes advantageous to provide some means of direct communication between threads. Handshake protocols that are locked-based can easily be integrated into the twilight code. In a similar vein, legacy code that is unaware of transactions can also be integrated with transactional code without compromising safe execution of the transactions.

Contention management In a situation with high contention, a restart gives rise to more even more contention. Twilight STM can be used to employ a sophisticated, application specific contention management to prevent the application from spin-locking in this situation.

4 The Twilight API

Many operations of the Twilight API (`stm_begin`, `stm_read`, `stm_write`, etc.) behave as in conventional STMs with weak atomicity. Thus, this section concentrates on the operations of the Twilight API that are new or changed with respect to other STMs. While the correct working of the discussed API relies on programming conventions, the appendix A sketches a language extension which rules out incorrect uses of the API.

We use the following terminology in this paper: A *(global) variable* is a location in global, shared memory. The *read set* (*write set*) of a transaction is the set of variables read from (written to) during the transaction. We maintain additional meta data about variables in the read set, including a timestamp, region marker, and the latest value read. The write set additionally records the last value written to each variable. To precisely specify the semantics of the operations, a notion of time is needed, which is further elaborated in Sec. 6. For some time t, a read set is *t-consistent* if all variables in it are t-consistent. A variable is t-consistent in read set R if and only if its value in R is equal to its value in the global memory at time t.

Twilight STM performs write operations lazily. Inside the transaction it only records the new values locally in the write set. These write operations get published to the shared memory on completion of the transaction.

4.1 Demarcating regions

Regions can be used to split the read set into non-overlapping groups of variables. They are denoted by integer markers, and are valid for the extent of one transaction. Regions are not semantically nested: they are handled in a stack-like manner with the topmost marker serving as the current region marker. `stm_enter_region`(*i*) puts the marker *i* on the stack, `stm_leave_region`() pops it from this stack. Each variable has a unique region marker, which is determined by the region of its first read. If the marker stack is empty, variables are placed in a default region.

In the twilight code, the programmer can use the regions to determine which variables were marked as inconsistent and compensate for them as applicable (cf. Sec. 3.1).

4.2 Preparing and finalizing the commit

Reminiscent of a two-phase commit in the database world, the Twilight STM splits the commit operation into operations `stm_prepare` and `stm_finalize` as indicated in Fig. 3. The return value of `stm_prepare` indicates if the transaction's read set is inconsistent because of other transactions committing since the transaction's start. However, the decision on the outcome of the transaction is left to the following twilight code.

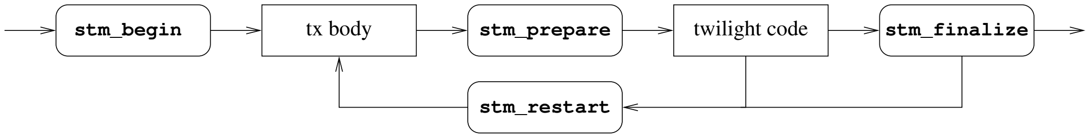

Figure 3: Work flow of a Twilight transaction.

The twilight code can observe and modify the state of the transaction with *twilight operations*. Subsequently, the code can either restart the transaction (`stm_restart`), or it can fix the inconsistencies and finalize the transaction (`stm_finalize`). In the latter case, the values in the write set are published to shared memory.

4.3 Twilight operations

The following operations must not be used outside of twilight code.

- `stm_reload`(): Atomically reloads a consistent snapshot of the read set. Afterwards, the programmer is obliged to update the write set to reflect the new values in the read set or to restart the transaction. This operation is always needed if the twilight code is entered with an inconsistent read set. Otherwise, the transaction cannot commit.
- `stm_inconsistent`(region): Indicates whether the region contains at least one inconsistent variable.
- `stm_only_inconsistent`(region): Indicates whether the `region` contain at least one inconsistent variable, and all other regions do not contain inconsistencies.
- `stm_inconsistencies`(): Returns all inconsistent regions.
- `stm_lock`(lck): Serves as wrapper for lock acquisition method of an external lock. To prevent deadlocking with other twilight zones, the locks on the write set entries are temporarily released such that other transactions that have entered their twilight zones and need to validate their read set can proceed. After the acquisition of the lock, the read set gets reloaded atomically. The return value of `stm_lock` indicates if the read set has become inconsistent.
- `stm_unlock`(lck): Serves as wrapper for the release operation of an external lock.

Further, the read and write operations behave differently in the twilight zone:

- `stm_read`(var): Returns the value of variable `var` as it is recorded in the read set. It is an error if the variable is still inconsistent or if the variable is not in the read set. Extending the read set is allowed via `stm_extend_readset`. It checks whether the variable is currently updated by another transaction or has been updated since the begin of the transaction. If so, its return value indicates that the transactions state is not consistent any more.
- `stm_write`(var,val): Updates the recorded value of variable `var` with value `val`. It is an error if the variable is not in the write set. Extending the write set is not allowed.

5 Algorithm

The basic STM algorithm resembles closely TL2. Figure 4 contains pseudo code for the most important operations of the Twilight API. It relies on a global counter/timer `T`[2]. Each shared variable is associated

[2]Incorporating techniques as used in SkySTM [15] reduce the contention on `T`. For the sake of clarity, we refrain from doing so in this paper.

```
stm_begin(){
  t_init      ← sample T
  read set   ←∅
  write set  ←∅
  region     ← stack(0) // default region
  inTwiZone  ← false
  consistent ← true
}

stm_enter_region(r){
  push(region,r)
}

stm_leave_region(){
  pop(region)
}

stm_read(var){
  if contains(write set,var)
    return lookup(write set,var)
  if consistent
    if contains(read set,var) or inTwiZone
      return lookup(read set,var)
    (val,t_var) ← load var
    if var locked or t_init < t_var
      stm_restart()
    add(read set,var,(val,t_var,top(region)))
    return val
  else error
}

stm_write(var,val){
  if inTwiZone
    update(write set,var,val)
  else
    add(write set,var,val)
}

stm_prepare(){
  succeeded ← false
  while !succeeded
    wait till S ≤ 0 and decrement S
    succeeded ← try lock write set
    increment S
  validate_readset()
  inTwiZone ← true
  return consistent
}

validate_readset(){
  consistent ← true
  for (var,(_,t_var,r)) in read set
    (val,t'_var) ← load var
    if t_var ≠ t'_var or
     (!contains(write set,var) and var locked)
        consistent ← false
}

stm_finalize(){
  if !consistent
    stm_restart()

  t_commit ← sample T
  publish(write set,t_commit)
  unlock write set
}
```

```
stm_extend_readset(var){
  (val,t_var) ← load var
  add(read set, var,(val,t_var,top(region)))
  if var locked by other transaction or t_init < t_var
    consistent ← false
    return false
  else
    return true
}

stm_lock(lck){
  stm_exposed( {acquire_lock(lck)} )
}

stm_unlock(lck){
  release_lock(lck)
}

stm_try_reload(){
  for (var,(val,t_var,r)) in read set
     (val',t'_var) ← load var
    add(read set, var,(val',t'_var,r))
  return validate_readset()
}

stm_reload(){
  stm_exposed( {} )
}

stm_exposed(delegate){
  wait till S ≥ 0 and increment S
  unlock write set

  delegate()

  validate_readset()
  if consistent
    lock write set
    decrement S
    return true
  else
    lock read and write set
    decrement S

    reread the read set
    unlock read set
    return false
}

stm_inconsistencies(){
  regionset ←∅
  for (var,(_,t_var,r)) in read set
    if (t_init < t_var)
      add(regionset,r)
  return regionset
}

stm_inconsistent(region){
  return contains(stm_inconsistencies(),region)
}

stm_only_inconsistent(regions){
  return subset(stm_inconsistencies(),regions)
}
```

Figure 4: Pseudo code of operations from the Twilight API.

with a version number — the time of its last modification. The first (transactional) read of a variable creates an entry in the read set comprising the variable, its value, and its version number at the time of the read operation. Write operations to shared variables are performed lazily. They are first recorded locally in the transaction's write set.

At its beginning, a transaction reads the timer to obtain t_{init}. To prevent zombie transactions, Twilight restarts a transaction if it tries to read from a variable with a time stamp more recent than t_{init}.

When the transaction attempts to commit, it invokes the method `stm_prepare`. The protocol associates with each global variable `v` a lock which grants exclusive access to `v`. A semaphore `S` acts as a sentinel that manages the permission to start executing twilight code. Initially, `S` is `0`. As an unusual feature, `S` can assume *integer* values including negative ones. If `S` is not zero, some transaction is in the process of acquiring the locks. In particular, if $\mathtt{S} < 0$ then at least one transaction executes `stm_prepare` and if $\mathtt{S} > 0$, then at least one transaction executes `stm_expose` (either from `stm_reload` or from `stm_lock`). Before entering its twilight code, a transaction waits until $\mathtt{S} \leq 0$. Similarly, it may only proceed with `stm_reload` or `stm_lock` if $\mathtt{S} \geq 0$.

After acquiring exclusive access to the variables in the write set, the transaction validates the read set by checking that the current version numbers of the read set entries are still less than or equal to t_{init}. The validation collects the inconsistent entries with a version number more recent than t_{init}.

In this state, the transaction enters its twilight code, which may contain the twilight operations from Sec. 4 as well as other operations. If the twilight code wants to correct inconsistent reads, it can only do so after obtaining a consistent read set with `stm_reload` or `stm_lock` and then performing some `stm_write` operations. In the end, the transaction is either restarted, or the commit is completed by calling `stm_finalize`. The latter operation first checks that the read set is consistent. It restarts if this is not the case. Otherwise, it publishes the write set by writing the new values to the shared variables and setting their version numbers to the current time t_{commit}. In any case, the transaction releases the exclusive access to the shared variables.

The operation `stm_reload` updates the read set atomically. To improve performance, the reload of the read set only happens if the current read set is inconsistent.

The operation `stm_reload` is guaranteed to succeed, but it can be quite expensive because it obtains exclusive access to the read set, drops the locks on the write set's variables, and uses the semaphore `S` to ensure a successful outcome. To avoid this expense, the API also includes a lock-free variant `stm_try_reload` that does not perform these safeguards and simply tries to reload the read set's variables. On the upside, `stm_try_reload` may run concurrently with `stm_prepare`, but on the downside, it may not always produce a consistent read set (e.g., when there is high contention). It fails when it encounters a locked variable as this variable is then (likely) locked by another transaction for modification.

The Twilight API operations rely on a locking protocol that maintains the following invariants:

1. Transactions that execute twilight code concurrently have disjoint write sets.
2. A transaction has exclusive access to the variables in its write set from `stm_prepare` to `stm_finalize`.
3. The Twilight API is deadlock-free.

The locking protocol facilitates a form of collaboration between transactions that has a low overhead at the expense of concurrency. No transactions can pass through `stm_prepare` while another transaction is obtaining its write locks through `stm_reload`. However, in practice, transactions only need to use `stm_reload` in the rare cases when `stm_try_reload` fails.

6 Properties of Twilight STM

Twilight STM provides operations to the programmer which relax the isolation of transactions in a controlled way. As Twilight supports concurrent execution of twilight code, single global lock semantics [17] is not appropriate. Subsection 6.1 specifies suitably relaxed properties and Subsection 6.2 proves that the Twilight transactions fulfill them.

6.1 Properties of Twilight Transactions

Twilight's semantics is captured by the following properties:

Weak Atomicity. Transactional writes and reads appear to take place atomically at a single global time, t_{commit}.

Consistency. The values in the read set are consistent with respect to t_{init} during the execution of the transaction and with respect to t_{commit} in the twilight code.

Progress. All Twilight API operations terminate if the execution of twilight code does not involve waiting for other transactions for non-transactional synchronization.

Irrevocability. A transaction executing its twilight code can always finish successfully.

Independence. Transactions executing their twilight code concurrently have disjoint write sets.

The first two properties specify the transactional behavior. The consistency property guarantees that out-dated values are consistently re-read. Results from computations on them can only be out-dated, but not wrong as the algorithm prevents inconsistent memory snapshots. Furthermore, the transaction can also observe the current values and has a chance to adjust its results.

The last three properties are related to the twilight code. Irrevocability ensures that twilight code can perform I/O and other irrevocable actions safely, and independence enables it to adjust the outcome of the transaction without further re-validation. Progress implies deadlock freedom for the correct use of the Twilight API operations.

Twilight transactions implement weak atomicity. However, unique to Twilight STM is that twilight code can access transactional and non-transactional memory locations in the same place. Therefore, demanding that transactional data is only accessed by means of transactions is not a strong restriction. The transaction is then made atomic with respect to other operations by combining `stm_reload` operation (which determines the point in time the transaction's actions place) with another form of synchronization (i.e. invoking `stm_reload` after exclusive access to some resource has been obtained). Using `stm_reload` creates an ordering among transactions and `stm_lock` ensures that this order does not conflict with external forms of synchronization used in the twilight code.

6.2 Proofs

This section contains the formal statements and proofs of the properties from Section 6.1.

6.2.1 Weak Atomicity

Theorem 6.1. *Transactional writes and reads appear to take place atomically at a single global time,* t_{commit}.

Proof. Assume that the transaction succeeded and finished with `stm_finalize`, and consider the associated time stamp t_{commit}. Because the transaction does not read any updates after `stm_prepare` and possibly `stm_reload`, it operates on a consistent memory snapshot without observing interferences from other transactions. Hence, the transaction's writes are published atomically with respect to a consistent read set. □

Although the memory updates appear to happen atomically, there is no global order on the commits. If t_{commit} of transaction A is more recent than t_{commit} of transaction B, then there is not necessarily a read-write dependency between these transactions even though transaction B may read variables which transaction A is writing to.

6.2.2 Consistency

The entries in the read set consist of a local copy of the value of each read variable with its version number and a flag which marks its consistency with respect to the memory snapshot taken so far. A value can only be read from the read set (through `stm_read`) when it is marked as consistent, and each read operation checks for consistency.

Theorem 6.2. *The values in the read set are consistent with respect to t_{init} during the execution of the transaction and to t_{commit} in the twilight code.*

Proof. Before entering `stm_prepare`, values are stored in the read set only if they have a version number smaller than or equal to t_{init} (to prevent zombie transactions). Otherwise the transaction is restarted. Therefore, the values in the read set are consistent with respect to t_{init}.

After `stm_prepare`, during execution of the twilight code, the only way to change entries in the read set is by means of `stm_reload` or `stm_try_reload`. In the case of `stm_reload`, all entries in the read set are reread atomically such that they are consistent. As no intermediate updates can be seen till the call of `stm_finalize`, the read set maintains its consistency until t_{commit}. Similarly, for `stm_try_reload`, it holds that either a value has been updated successfully, or some of the variables were locked. In the latter case, the entries are marked as inconsistent and the values are thus not accessible (i.e., `stm_read` fails on these variables). □

6.2.3 Progress

The progress property essentially states deadlock freedom and absence of starvation. The underlying assumption is that the sentinel `S` is biased towards `stm_safe_exposed` and that there is a fair scheduling of processes.

Theorem 6.3. *All Twilight API operations terminate, assuming that the execution of all twilight code either does not involve waiting on other transactions (due to non-transactional blocking synchronization), or invokes* `stm_safe_exposed` *before doing so.*

Proof. There are only three operations that may involve waiting: `stm_prepare`, `stm_safe_exposed` and `stm_reload`. All other operations have their execution time bounded by a polynomial in the size of read and write set. Each of these three operations may need to wait for another transaction that either executes `stm_prepare`, `stm_safe_exposed`, `stm_reload`, or some non-Twilight API code. We examine the relevant combinations.

`stm_prepare` needs to acquire shared access from `S` and exclusive access to the variables in its write set. This exclusive access is obtained in a fixed order, which rules out deadlocks with another transactions' `stm_prepare` or `stm_reload`. A transaction executing twilight code is never dependent on a transaction which is executing its transaction body. Therefore, while some `stm_prepare` is waiting, the transaction executing twilight code that prevents the `stm_prepare` from obtaining a lock, will eventually release the lock. No other transaction can enter its twilight code prior to that release, but other transactions are free to leave their twilight code.

`stm_safe_exposed` may have to wait for an `stm_prepare`. Because `stm_safe_exposed` drops the locks on the write set variables, this `stm_prepare` never has to wait for the transaction of `stm_safe_exposed`. Therefore, such an `stm_prepare` can continue and eventually reaches the point where it releases the entrance permission, so that `stm_safe_exposed` can obtain it.

`stm_reload` never needs to wait for a transaction executing `stm_safe_exposed` because `stm_safe_exposed` has dropped the locks on the write set's variables. Transactions executing non-twilight API code that requires waiting for some external blocking source are therefore required to have executed `stm_safe_exposed`. Also, `stm_reload` only waits a finite time for an `stm_prepare`, for similar reasons as `stm_safe_exposed` only needs to wait a finite time. Because `stm_reload` re-acquires all locks in predetermined order, it never deadlocks with another `stm_reload`. One of the transactions can then continue when the other transaction finishes or when the other transaction executes a `stm_safe_exposed` or `stm_reload` again. □

6.2.4 Irrevocability

After `stm_prepare` a transaction is in principle committable. The reason is that a transaction has exclusive access to the variables in its write set. No other transaction can therefore interfere in the transaction's commit process.

Theorem 6.4. *A transaction executing its twilight code can always finish successfully.*

Proof. To finish successfully, the transaction has to invoke in the end `stm_finalize` while having a consistent read set. If the read set was discovered to be inconsistent during `stm_prepare`, the transaction can execute `stm_reload` to obtain consistency. Due to the progress property, `stm_reload` terminates successfully in finite time, so it can be invoked prior to `stm_finalize`. □

As a side note, invoking `stm_finalize` directly after `stm_reload` ignores the new values that were written before starting the twilight code, but after their variables were read by the transaction. Essentially, it confirms that the values in the write set (computed from the old values) are good enough, no matter what the new values are. In practice, such ignorance is the exception. Instead, a programmer wants to iterate over the inconsistencies fixed by `stm_reload` and recompute the values in the write set or even restart. Twilight empowers the programmer to choose the most appropriate outcome.

6.2.5 Independence

Theorem 6.5. *Transactions executing their twilight code have disjoint write sets.*

Proof. The operation `stm_prepare` ensures that a transaction entering its twilight code has exclusive access to the variables in its write set. This exclusive access is obtained while having entrance permission from the sentinel $\mathtt{S}$. Hence, all successful concurrent executions of `stm_prepare` must yield disjoint write sets. As further $\mathtt{S} > 0$ while obtaining the write locks, there cannot be another transaction with temporarily dropped access rights: that would require this other transaction to have executed `stm_safe_exposed` and not yet be finished with `stm_reload`. However, this other transaction would have established $\mathtt{S} < 0$ in `stm_safe_exposed`, which contradicts the assumption that $\mathtt{S} > 0$. Thus, the write sets are always disjoint. □

7 Open-Nested Transactions

A twilight transaction may only be a top-level or open-nested transaction. Closed-nested transactions are not allowed to use twilight-functionality.

Following the argumentation and proposal of Moss [20], open nested child transactions do not observe their parents' local state. This restriction prevents them from unintentionally publishing intermediate states. Child transactions have their own read and write sets, separate from their parents' read and write sets. To prevent parents from restarting due to conflicts induced by the children, we update the parents' values with the corresponding values as published by the children. This write-through leads to a inversion of control flow as a child transaction gets executed independently before the parent. Although this choice seems startling at first glance, it inhibits surprising behavior when nesting with twilight code.

8 Evaluation

To evaluate Twilight's usability and performance, we provide two implementations. The first one is a conservative extension of the TL2 algorithm [5] written in C. This implementation is library-based and it can be used with any platform supporting pthreads. Its correctness hinges on programming conventions, which are neither checked nor statically enforced.

A program not using twilight code has the same guarantees for execution inside a transaction as before. Like TL2, Twilight requires exclusive access to the transaction's variables during the commit. Twilight's functionality can be integrated into other STMs that have the same requirement.

There is also an unoptimized reference implementation in Java, which is object-based and which provides a more user-friendly API: a transaction body is represented as an object implementing two methods, `transactional`() and `twilight`(). Executing such a transaction body means to open a transaction, run `transactional`(), prepare to commit (and restart if needed), and run the `twilight`() method, which leads to a commit or a restart. The body object also serves as a container to communicate values between the transactional code and the twilight code.

8.1 Benchmarks

Twilight STM is intended for situations where a transaction is not allowed to restart, in particular when I/O operations are involved. This feature is powerful, but it cannot be measured meaningfully. Another typical use is contention management based on insight into the system's state dynamics. Thus, the experiments for evaluating the Twilight STM concentrate on the contention management aspect. To show the competitiveness of Twilight STM, we compare the performance of Twilight STM (the C implementation, both with and without twilight code) with the TL2 reference implementation [5].

For this purpose, we added twilight code to the vacation application from the STAMP benchmark [3]. The application implements a multi-threaded enterprise server. Each thread simulates a client that makes reservations for hotel rooms, cars, or flights. These bookings lead to updates of the available items, which are represented with hash maps. Access to the data in these collections is wrapped into transactions.

We assigned group identifiers to each STM operation to be able to recover from the following cases of inconsistency during a commit:

- An internal restructuring of a hash map caused by deletion or addition of entries may cause some inconsistency on the path to a data item, but it does not invalidate the data item itself. We ignore these benign inconsistencies as they do not affect the application logic.
- If some other client's booking interrupts our booking and there are still enough items available, we recalculate the new available amount and update it directly instead of performing a restart.

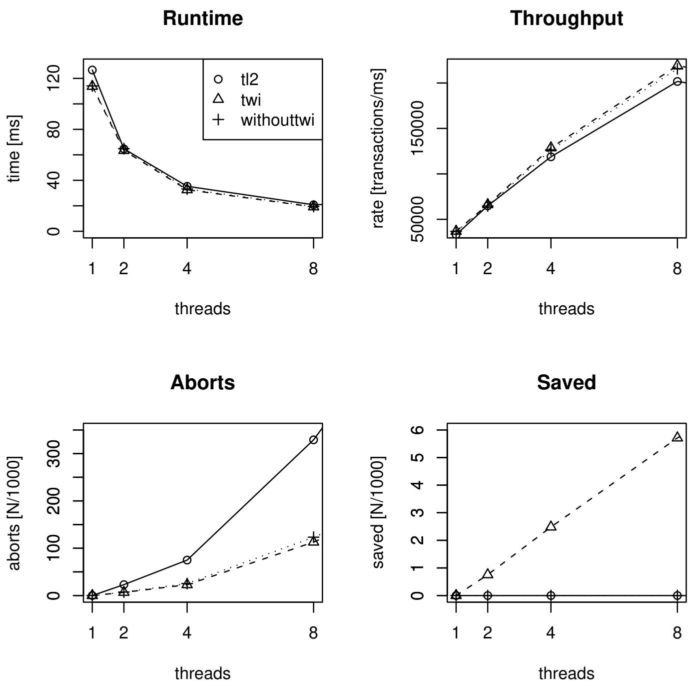

Figure 5: Vacation benchmark with parameters -n4 -q60 -u90 -r1048576 -t4194304.

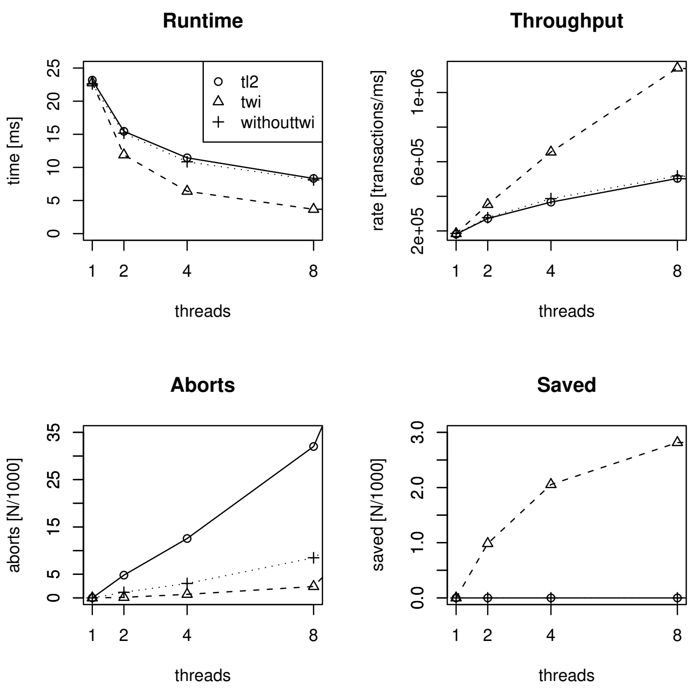

Figure 6: Vacation benchmark with parameters -n100 -q90 -u90 -r200 -t4194.

The benchmark machine is powered by a Quad-Core AMD Opteron at 2.3 GHz with caches 4 x L1 64 KB, 4 x L2 512 KB, and L3 2048 KB shared. It runs OpenSuSE 10.2 and gcc 4.2.2.

Figure 5 displays graphs where each entry is the average taken from 10 runs of the vacation application using the parameters for high contention [3]. The "Aborts" graph shows the number of aborted transactions (in 1000Tx; the highest mark indicates more than 300,000 aborts) and the "Saved" graph shows the number of transactions saved by twilight code (in 1000Tx; highest mark >5,000). The proportion of saved to aborted transactions shows that the twilight code can only rescue a transaction in a small number of cases. The ratio of aborts to saves is around 100:11 for 2 threads, 100:10 for 4 threads, 100:5 for 8 threads). The Twilight STM both with and without twilight code show similar performance. Apparently, already this small number of transactions is sufficient to pay for the extra overhead of performing the twilight code's computations. Furthermore, Twilight performs slightly better than TL2. TL2 aborts in its preparation phase as soon as it discovers that a variable that it reads from is locked. This strategy can lead to multiple restarts. 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 significantly fewer aborts and leads to marginally better performance in case of high contention.

For Figure 6 we increased the contention even more by performing a higher number of actions per transaction and acting on a lower number of different items to see if the contention management is effective. Here, the effectiveness of the contention management is revealed. By increasing the contention, the opportunity to save transactions is also increased. The ratio of aborts to saves is approximately 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 performance. The execution time of TL2 and Twilight STM without twilight code is comparable. However, Twilight STM with twilight code is significantly faster.

9 Conclusion and Future Work

Twilight STM is an implementation of software transactional memory that extends the commit phase with twilight code. This code may perform irrevocable actions that execute I/O operations, run lock-based protocols with other peers and concurrently executing transactions, or implement sophisticated contention management. Twilight STM guarantees a seamless integration of these potentially unsafe actions into a transactional setting. It preserves the transactional semantics while imposing only mild restrictions on the twilight code. The additional flexibility offered by Twilight STM gives the programmer fine-grained control over the application's concurrent behavior. Experimental results show that the Twilight implementation is competitive with a state-of-the-art STM implementation.

There are several directions for future work:

- Abstract nested transactions [12] are a highly useful abstraction which should be easily implementable with Twilight STM.
- Static enforcement of API constraints give improved language support for creating a safe STM API such that operations cannot be easily misused as shown in Appendix A.

References

[1] C. Blundell, J. Devietti, E. C. Lewis, and M. M. K. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. In D. M. Tullsen and B. Calder, editors, *34th International Symposium on Computer Architecture (ISCA 2007)*, pages 24–34, San Diego, California, USA, 2007. ACM.

[2] C. Blundell, C. Lewis, and M. Martin. Unrestricted transactional memory: Supporting I/O and system calls within transactions. Technical Report TR-CIS-06-09, University of Pennsylvania, Philadelphia, PA, USA, May 2006.

[3] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. Stamp: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of the IEEE International Symposium on Workload Characterization*, pages 35–46, September 2008.

[4] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ASPLOS-XII: Proceedings of the 12th International Conference on Architectural support for programming languages and operating systems*, pages 336–346, San Jose, California, USA, 2006. ACM.

[5] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proceedings of the 20th International Symposium on Distributed Computing, DISC 2006*, LNCS 4167, pages 194–208. Springer, 2006.

[6] K. Donnelly and M. Fluet. Transactional events. In *ICFP '06: Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, pages 124–135, Portland, Oregon, USA, 2006. ACM.

[7] L. Effinger-Dean, M. Kehrt, and D. Grossman. Transactional events for ML. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN International Conference on Functional Programming*, pages 103–114, Victoria, BC, Canada, 2008. ACM.

[8] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. *SIGARCH Comput. Archit. News*, 32(2):102, 2004.

[9] T. Harris. Exceptions and side-effects in atomic blocks. *Science of Computer Programming*, 58(3):325–343, 2005.

[10] T. Harris and K. Fraser. Language support for lightweight transactions. In *Proc. 18th ACM Conf. OOPSLA*, pages 388–402, Anaheim, CA, USA, 2003. ACM Press, New York.

[11] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable memory transactions. In *Sixteenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 48–60, Chicago, IL, USA, June 2005. ACM Press.

[12] T. Harris and S. Stipić. Abstract nested transactions. In *Second ACM SIGPLAN Workshop on Transactional Computing, TRANSACT'07*, Portland, OR, USA, Aug. 2007.

[13] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *Proc. 21th ACM Conf. OOPSLA*, pages 253–262, Portland, OR, USA, 2006. ACM, New York, NY, USA.

[14] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In J. Torrellas and S. Chatterjee, editors, *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP 2006)*, pages 209–220, New York, New York, USA, 2006. ACM.

[15] Y. Lev, V. Luchangco, V. Marathe, M. Moir, D. Nussbaum, and M. Olszewski. Anatomy of a scalable software transactional memory. In *4th ACM SIGPLAN Workshop on Transactional Computing, TRANSACT'09, Raleigh, USA*, 2009.

[16] A. McDonald, J. Chung, B. D. Carlstrom, C. C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural semantics for practical transactional memory. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 53–65, Washington, DC, USA, 2006. IEEE Computer Society.

[17] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. L. Hudson, B. Saha, and A. Welc. Single global lock semantics in a weakly atomic STM. *SIGPLAN Not.*, 43(5):15–26, 2008.

[18] K. Moore, J. Bobba, M. Moravan, M. Hill, and D. Wood. LogTM: log-based transactional memory. *The Twelfth International Symposium on High-Performance Computer Architecture, 2006.*, pages 254–265, Feb. 2006.

[19] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood. Supporting nested transactional memory in LogTM. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 359–370, San Jose, California, USA, 2006. ACM Press, New York, NY, USA.

[20] J. E. B. Moss. Open nested transactions: Semantics and support. Poster presented at Workshop on Memory Performance Issues (WMPI 2006), Austin, TX, Feb. 2006.

[21] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. In *ASPLOS X: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 5–17. ACM Press, 2002.

[22] H. E. Ramadan, I. Roy, M. Herlihy, and E. Witchel. Committing conflicting transactions in an STM. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 163–172, Raleigh, NC, USA, 2009. ACM.

[23] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *20th International Symposium on Distributed Computing, DISC 2006, Stockholm, Sweden*, pages 284–298, 2006.

[24] M. F. Ringenburg and D. Grossman. AtomCaml: First-class atomicity via rollback. In B. C. Pierce, editor, *Proc. ICFP 2005*, pages 92–104, Tallinn, Estonia, Sept. 2005. ACM Press, New York.

[25] C. J. Rossbach, O. S. Hofmann, and E. Witchel. Is transactional memory programming actually easier? In *Proceedings of the 8th Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD), Austin, Texas*, 2009.

[26] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP'06: Proceedings of the 11th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 187–197, New York, NY, USA, 2006.

[27] Y. Smaragdakis, A. Kay, R. Behrends, and M. Young. Transactions with isolation and cooperation. In *Proc. 22nd ACM Conf. OOPSLA*, pages 191–210, Montreal, QC, CA, 2007. ACM Press, New York.

[28] M. F. Spear, M. M. Michael, and M. L. Scott. Inevitability mechanisms for software transactional memory. In *3rd ACM SIGPLAN Workshop on Transactional Computing, TRANSACT'08, Salt Lake City, UT.*, 2008.

[29] T. Sweeney. The next mainstream programming language: A game developer's perspective. In S. Peyton Jones, editor, *Proc. 33rd ACM Symp. POPL*, page 269, Charleston, South Carolina, USA, Jan. 2006. ACM Press.

[30] H. Volos, N. Goyal, and M. M. Swift. Pathological interaction of locks with transactional memory. In *Third ACM SIGPLAN Workshop on Transactional Computing, TRANSACT 2008*, Salt Lake City, UT, USA, Feb. 2008.

[31] H. Volos, A. J. Tack, N. Goyal, M. M. Swift, and A. Welc. xCalls: Safe I/O in memory transactions. In *EuroSys '09: Proceedings of the fourth ACM European Conference on Computer systems*, pages 247–260, Nuremberg, Germany, 2009. ACM.

[32] A. Welc, A. L. Hosking, and S. Jagannathan. Transparently reconciling transactions with locking for Java synchronization. In *20th ECOOP*, volume 4067 of *LNCS*, pages 148–173, Nantes, France, July 2006. Springer.

[33] A. Welc, B. Saha, and A.-R. Adl-Tabatabai. Irrevocable transactions and their applications. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 285–296, New York, NY, USA, 2008. ACM.

[34] L. Ziarek, A. Welc, A.-R. Adl-Tabatabai, V. Menon, T. Shpeisman, and S. Jagannathan. A uniform transactional execution environment for Java. In J. Vitek, editor, *22nd ECOOP*, volume 5142 of *LNCS*, pages 129–154, Paphos, Cyprus, 2008. Springer.

A Higher-level syntax

The Twilight API given in Section 4 provides the bare essentials for fixing inconsistencies before committing. However, more expressiveness is needed when Twilight operations are used in a practical setting. For standard transactional data structures (e.g., lists and trees), there should be a library of easy-to-use recovery functionality. Recovery should be specified in a local and composable way, without corrupting the structure of the program. To this end, an abstraction layer has to be built on top of the bare essentials. This layer can be obtained by using a language extension that enforces programming conventions and increases usability.

```

atomic {
  region (r) {
    /* ... stm statements */
  }
  recovery(r) {
    /* ... recompute */
  }
}

twilight {
  acquire {
    /* ... acquire/lock external resources */
  }
  if (main.isConsistent()
      && !r.isConsistent()) {
    r.applyRecovery();
    /* ... more recovery */
  }
  /* ... perform irrevocable I/O */
  /* ... release external resources */
}
```

Figure 7: Example of higher-level syntax for Twilight.

Figure 7 sketches such a language extension. Some of the ideas were already used for the twilight code in the vacation benchmark (Section 8). For example, the grouping feature was valuable to quickly discover frequently occurring inconsistencies and to easily pinpoint recoverable situations later on. Future work addresses the proper design of a suitable language extension.

A twilight transaction is defined using the standard `atomic` syntax, with a `twilight` block attached. Twilight transactions may be nested with open-nested semantics (Sec. 7). A nested transaction without a `twilight` block becomes a close-nested transaction. An atomic-block may occur inside the twilight-block. Twilight-blocks, however, may not be nested.

A `region` block attaches a region object to a number of STM statements. `region` blocks may be nested, in which case the innermost group is attached to the statements. Also, closures of the `recover` blocks are registered with the region. Inconsistencies in a region can be recovered by checking the condition of a `recover` block and applying it if possible. This style enables a local (and composable) definition of recovery, while orchestrating the whole recovery process in the `twilight` block.

A `atomic` block in the twilight code requires as precondition a consistent read set. If this is not the case, the `recover` blocks are consulted to recover the read set if possible. If this cannot be done, the `else` block is executed, and otherwise execution continues in the `atomic` block. An atomic-block inside the twilight code can be regarded a close-nested transaction with some restrictions. Writes may only range over variables in the write set, and reads over variables not in the read set may require a restart of the `atomic` block. Under the hood, this involves re-establishing the precondition by running `stm_reload` again and applying recovery if possible.

A `recover` block may also occur without a `atomic` block attached. In that case, an empty `atomic` block is assumed to be prefixed.

The syntax allows for a number of static checks. For example, a warning could be issued when there exists an execution path on which an `expose` block is not followed up by a `atomic` block or a `recover` block. A flow analysis could warn about assignments to global variables not in the write set or reads from variables which are not recorded in the read set outside an `atomic` block.