Towards Software Transactional Memory for Real-World Programs

Benjamin Mellor
Supervisor: Zoltan Somogyi

Honours Report 2008-2009

Abstract

Parallel programming is becoming more and more important as commonly available machines begin to grow in their capability for parallel execution. Parallel streams of execution that operate on shared data to achieve a common task need to be carefully synchronised. The standard way of achieving this synchronisation is to use locks, but lock-based programming is difficult and error-prone, making parallel programs costly and unreliable.

Another method of providing synchronised access to shared data is the transactional memory model. Software Transactional Memory (STM), implementing this model purely in software and thus compatible with today's hardware, has been an area of much research in the last decade.

This report describes the implementation of a sophisticated STM system using recently developed techniques as well as novel innovations. This implementation is for the logic programming language Mercury, building on an earlier and simpler STM implementation.

Contents

List of Algorithms

1 Introduction

It seems obligatory to introduce a discussion of software transactional memory with the note that the rapid growth in processor speeds, which the computing world has been accustomed to for decades, has all but halted. Progress in the design of CPUs is instead resulting in increasing the number of tasks that can be executed in parallel. Already basic entry-level desktop computers near-universally have dual-core processors, and quad-core processors are readily available. With this development comes an increasing need for parallelism to be utilised within single applications for acceptable performance. Therefore, there is an increasing need for programmers to produce applications in which multiple processes execute in parallel and coordinate to achieve some shared task.

Parallel programming introduces the complexity of reasoning about simultaneous threads of execution acting on the same shared data. The standard approach to managing these difficulties is to synchronise the parallel processes using *locks* to ensure that certain sections of code are not executed in parallel with each other. Lock-based parallel programming, however, is widely acknowledged as a difficult and error-prone task. Several classes of bugs that can result from mistakes in this area cannot be found by static analysis, and can sometimes be difficult to find even with the most rigorous testing, as they are only triggered under precise timing conditions while executing tasks in parallel. All of this increases the cost of developing high performance parallel software.

Software transactional memory (STM) is a relatively recent approach to the development of parallel applications. In this approach, the programmer designates sections of code as *transactions*, and it is the STM system's responsibility to execute these transactions as if they were atomic — as if the entire body of the transaction were executed at a single moment of time. This is achieved by optimistically running transactions, but keeping enough bookkeeping to transparently abort and restart transactions when necessary, similar to the way concurrency of transactions is managed in database systems.

In a language with strong static checks, misuse of the transactional system can be detected at compile time, and a wide range of the difficult-to-find bugs that can occur in parallel programs simply cannot occur. Furthermore, the transactional model increases the extent to which programmers can apply purely local reasoning when writing a single part of the parallel program, whereas in lock-based programming the internal details of how locks are used in various unrelated parts of the system must often also be considered for correct results. These properties make parallel programs based on software transactional memory easier to develop and easier to maintain, having the potential to dramatically decrease the costs of producing parallel programs, and therefore widen the class of programs that are worth the effort of parallelising.

While very simple coarse-grained locking approaches outperform STM systems for low levels of parallelism, and highly tuned fine-grained locking approaches often outperform STM systems at all levels of parallelism, STM promises to combine the simplicity and reliability of coarse-grained locking with the high scalability of fine-grained locking. If achieved, this would make it an attractive approach for many applications. The class of applications for which STM is well-suited is determined by the overheads incurred by the system; the lower the overhead for the "fast path" of mostly uncontended access, the lower the number of cores needed for STM-based programs to out-perform approaches based on coarse-grained locking.

This report describes the an STM system implemented for the language Mercury. Mercury is a pure logic programming language under development at the University of Melbourne. My STM system is built on an earlier implementation of STM for Mercury, and uses advanced techniques recently developed, with the aim of producing a system closer to being attractive for real-world use. It also incorporates two novel contributions: a strategy for logging writes that avoids the need to search a write set for transactional reads to find earlier writes by the same transaction, and a system of *checkpoints* that mitigates the cost of aborting transactions by allowing them to resume execution part-way through instead of rolling back to the start, in some circumstances.

Section 2 briefly describes the Mercury language, and gives more background on software transactional memory and the motivation for using it. Section 3 describes the earlier implementation of

STM for Mercury, and sections 4 and 5 provide detailed descriptions of the improvements I have made to this system. Section 6 describes features of the system I have planned for, but are not yet implemented, while section 7 concludes with a discussion of future areas of research.

2 Background

2.1 Mercury

Mercury is a logic programming language developed at the University of Melbourne, with an emphasis on reliability and efficiency. It was designed to support the development and maintenance of large software systems. The syntax of Mercury is derived from that of Prolog, but the two are semantically quite different. Some of the key features are described below, largely taken from material found in [15], and [25]; a more complete description of the language can be found in [1].

Mercury contains a strong type system based on a many-sorted logic type system derived from the Hindley-Milner type system. Mercury supports parametric polymorphism as well as higher-order types.

Mercury has a strong *mode* system. A mode of a predicate specifies the initial and final instantiation states of arguments in a call to that predicate. A predicate can have multiple modes; each is a *procedure*. For example, one mode of the standard `append/3` predicate is `append(in, in, out)`. A call to that procedure must have the first two arguments instantiated before the call ("`in`"), and must have the last argument free before the call ("out"). The mode guarantees that output variable will be bound to the result when the procedure returns (and the input variables will remain bound to their values). The modes in which a predicate can be used must be declared or inferred. A called predicate imposes a set of constraints on the instantiation states of arguments in the call: input variables must be ground at the time of the call, and output variables will be ground after the call. The compiler will attempt to rearrange goals in a procedure so that all variables are produced before they are consumed. If that is not possible then the program will be rejected by the compiler. The end result is that we will always know, at compile time, which mode of a predicate is called at each call site, and which variables are instantiated when.

Mercury also has a strong *determinism* system. Determinism is associated with procedures and indicates how many solutions, if any, a procedure could return. The four main determinisms are "`det`", "`semidet`", "`multi`" and "`nondet`". A procedure with determinism `det` will always succeed and produce exactly one solution. A `semidet` procedure will also produce a single solution if it succeeds but can possibly fail, producing no solutions. Procedures with determinisms `multi` and `nondet` are capable of producing more than one solution. Procedures that are `nondet` can also fail whilst procedures that are `multi` must produce at least one solution. Another determinism a procedure can have is `erroneous`. This determinism identifies procedures which will never properly return, and is used by predicates that will always throw an exception or halt the program.

The mode system can also record the *uniqueness* of a term. A term is "unique" if there is only one live reference to it. A term which has no live references to it is said to be "clobbered". With these instantiations, the argument mode `di` (destructive input) can be used to indicate an instantiation mapping from unique to clobbered, and the argument mode `uo` (unique output) maps from free to unique. With uniqueness, it is possible to allow for destructive updates whilst maintaining declarative semantics. Mercury uses this method to handle I/O. Mercury views I/O predicates as describing a relationship between two states of the world: the one before the action and the one after. The difference between the initial and final I/O states includes the effect of the action, and the effects of all other actions executed by other programs since the creation of the initial I/O state. An I/O predicate uses the `di` and `uo` modes to destructively update the state of the world. Since the world has only one state at any one time, the old states of the world can no longer be used. This prevents backtracking over predicates which perform I/O.

Mercury also has a *purity* system. By default all Mercury goals are pure — their results depend only on the values of their input arguments — but it is possible to write impure Mercury code. Impure goals are annotated with `impure`, and predicates containing such goals must also

be marked `impure`, unless the programmer explicitly promises to the compiler that the predicate implements a pure interface. An informal way to think about the compiler's treatment of impure goals is that their position in the "left-to-right" order of execution cannot be changed, since their results may depend on being executed in their positions as written. Much code that interfaces with other languages is necessarily impure; the compiler assumes foreign language code is impure unless promised otherwise by the programmer. Nevertheless, Mercury wrappers can often provide a pure interface; this is how many of the pure predicates used to do I/O are implemented, for example.

2.2 Lock-based Synchronisation

The use of threads within a single process to achieve parallel execution, rather than using multiple processes, is often attractive for a number of reasons. Threads are relatively cheap to create and manage. As they share the same address space, threads also permit the parallel execution streams to share data very efficiently. This comes with a cost, however; access to data that is shared between threads must be carefully coordinated in order to ensure that program invariants are maintained.

The traditional method for synchronising access to shared data is to use locks. A lock is a software object, usually provided by the operating system, which can be held by a single thread at a time. If a thread requests a lock that is currently available, it becomes the lock-holder until it explicitly releases the lock. If the lock is already held by another thread, the thread requesting the lock *blocks* until the lock becomes available. It is guaranteed that when a lock becomes available, exactly one of the threads waiting for the lock will be granted the lock and allowed to continue as the lock holder.

Locks are used in data synchronisation by identifying *critical sections* of code, and creating an association between locks and critical sections; when a thread wishes to execute a critical section, it must acquire the associated locks beforehand, and release the locks afterwards. This ensures that only one thread can be running the critical section at a time. Provided all the code that manipulates the shared data is properly contained in critical sections, this ensures that threads will never see the shared data in an intermediate state while another thread is modifying it. Identifying the critical sections, and determining what lock(s) are needed to protect each one is the most significant challenge in designing concurrent programs.

Coarse-grained locking uses a small number of locks (perhaps only one) to protect access to the shared data of the program. This strategy is simple and direct, and usually minimises the overhead of acquiring and releasing locks. However, for many programs the total amount of shared data is much larger than the amount of data accessed in any one critical section. Coarse-grained locking seriously limits the benefit such programs can receive from parallel execution, as completely independent critical sections are often unable to execute in parallel.

Fine-grained locking makes use of a much larger number of locks. A common strategy is to associate each shared object with a lock used to protect access to that object individually. This allows critical sections that access disjoint sets of objects to execute in parallel, allowing performance to scale much more effectively with the number of processors available to execute the program. If the granularity is too fine, the overheads of acquiring and releasing locks can become significant, but there are much more serious problems with this synchronisation strategy.

With fine-grained locking schemes, each critical section will usually need to hold a number of locks to safely execute. The particular set of locks is usually determined at runtime based on the arguments to the procedure containing the critical section. The central problem is that these locks must be requested one at a time (in some cases the need for some locks may be determined based on data protected by other locks), and multiple threads requesting a common group of locks in different orders can result in deadlock: a set of threads, each of which is blocked waiting for a lock held by another thread in the set, with the result that none of them will ever resume execution or release the locks they are holding. The need to avoid deadlock requires complex and precise *locking protocols*, which are difficult to design and easy to make a mistake in following. The compiler is usually incapable of enforcing the locking protocol (or checking that the protocol is correct in the

first place), and the bugs caused by these failures may only show up under extremely precise timing conditions during parallel execution, making them very difficult to reproduce, and hence to debug (especially as compiling programs for debugging, or with extra debugging messages inserted, will change the timing characteristics of the program).

There are other problems associated with lock-based synchronisation as well. *Convoying* is the loss of parallelism that results when a thread holding a lock is unexpectedly delayed, and many other threads queue up "behind" it. They must then run one at a time when the lock finally is released, further reducing parallelism. Pre-emptive scheduling techniques used by modern operating systems can add an unbounded amount of delay to the time a thread holds locks. The memory hierarchy can also add delays that are essentially random from the point of view of the programmer, as execution stalls while waiting for data to reach the CPU cache from main memory, or even from virtual memory stored on disk. *Priority inversion* is another problem, when both high-priority and low-priority threads access the same data. The operating system will normally run the high-priority thread in preference to the low-priority one, but if the low-priority thread already holds a lock when the high-priority thread requests it, there is nothing that can be done; the high-priority thread will be forced to wait for the low-priority one. Furthermore, if there are other medium-priority threads in the system (possibly not even part of the same process), the operating system will typically prefer to spend execution time on the medium-priority tasks than the low-priority one, preventing it from running and releasing its locks, and so also delaying the high-priority thread. Effectively, the high-priority thread's priority is reduced to that of the low-priority thread it is forced to wait for, until it is actually able to run again. Priority inheritance is one well-known technique for addressing priority inversion, but it must be implemented in the operating system, and may not be available.

Perhaps the most serious problem with lock-based synchronisation, however, is that it is not composable; it is in general not possible to combine correctly synchronised lock-based operations into correctly synchronised composite operations. Consider a module that encapsulates the concept of a bank account, supporting `withdraw` and `deposit` operations, and internally using a separate lock to protect access to each account[1]. This scheme, on its own, is impossible to deadlock. But when a client module needs to implement an atomic transfer operation between two accounts, the natural method of implementing it (as a withdraw from one account followed by a deposit to the second account) will fail; it will be possible for other threads to observe a state in which the transferred money is in *neither* account, as separate locks are acquired and released around the two sub-operations. The transfer operation, which logically is a simple combination of the withdraw and deposit operations, has quite distinct synchronisation requirements; the locks for both accounts must be held for both operations (with the added complexity of having to ensure that executing `transfer(AccountA, AccountB, Amount)` in parallel with `transfer(AccountB, AccountA, Amount)` cannot result in a deadlock). This cannot be achieved unless the lock acquisition is separated from the `withdraw` and `deposit` operations, and the locks are made public and exported from the module. They must also potentially be exported from the client module to any further clients, to allow still higher level operations to be correctly synchronised. Since all levels of the program may need access to the locks, their use also cannot be an internal implementation detail of any operation, and must be carefully documented so that the locking protocol can be followed by the program as a whole. This has the consequence that the difficulties of writing correct lock-based code cannot be left to experts and hidden inside software libraries; all of the programmers working on a program that uses locks to synchronise access to shared data will have to deal with these issues, significantly increasing the complexity and cost of developing such a system.

2.3 Software Transactional Memory

An alternative approach to synchronising shared data is *transactional memory*. It was originally proposed as a hardware mechanism by Maurice Herlihy and J. Eliot Moss, based on an extension

[1]This example is covered in more detail, including Mercury code, in [15].

of multiprocessor cache-coherency protocols[9]. Nir Shavit and Dan Touitou proposed adopting the transactional model, but providing an implementation completely in software[21]. Software transactional memory has been the subject of much research in the last 15 years, promising a new methodology for developing parallel programs that combines the conceptual simplicity of coarse-grained locking with fine-grained locking's ability to scale to highly parallel execution. Early STM systems incurred large overheads, and so failed to out-perform sequential execution without a large number of processors, but as more sophisticated implementation techniques have been developed, software transactional memory has begun to appear to be an attractive alternative to lock-based synchronisation for many applications.

The idea of transactional memory is to provide access to shared in-memory data in a way similar to that provided by database transactions. It is an *optimistic* synchronisation technique; instead of ensuring that only one thread at a time may execute a critical section, all threads are allowed to execute *transactions*, and the underlying system promises to detect and abort execution of transactions that become "unsafe". This requires that the entry and exit points of transactions are known to the system, and that all code executed by a transaction is safe to re-execute any number of times, which typically excludes all I/O operations, as well as permanent modifications of globally visible in-memory data, from being performed inside transactions.

At a broad level, all implementations of software transactional memory achieve this in the same way. When a transaction is entered, a *transaction log* is created. All access to *transaction variables* are recorded to this log. When the transaction reaches its end, the system must prove that the recorded accesses in the log are still valid in the current global state of the system. If so, the transaction *commits*, otherwise it *aborts*. A transaction that aborts is transparently re-executed until it successfully commits. Most practical STM systems will detect the need to abort at other times as well as at the end of the transaction.

What exactly constitutes a transaction variable varies across implementations; all of the following schemes have been used:

- A transaction variable is a container-like data structure containing a single value.
- Each word of memory (usually excluding the local stack) is considered a separate transaction variable.
- Each object (usually excluding those on the local stack) is considered a separate transaction variable.
- Instances of a class inheriting from a particular abstract base class are transaction variables.

Data not stored in a transaction variable, whatever the notion of transaction variable that is in use, is considered to be outside the responsibility of the STM system, and is not synchronised. The way transaction variables are accessed also varies considerably. Sometimes special read and write operations are provided for transaction variables. Sometimes a variable is first opened for either read only access, or for read-write access, which returns a reference that can be used to access the data stored in the variable (this scheme is reminiscent of the way files are accessed in many languages, except the reference returned usually permits direct manipulation, unlike a file descriptor which must be manipulated through special interface procedures). Other systems simply allow normal operations on transaction variables, and the STM system must redirect these accesses to the log.

Demarcating sections of code as abortable transactions does not provide reliable synchronisation alone. Just as is the case for database transactions, the STM system must be able to ensure that transactions conform to certain safety properties; informally we require that the history of transactions executed in an application is equivalent to some alternative history in which the transactions were executed serially. It might seem like the safety criteria for STM system would match ones studied in the context of transactional database systems, such as linearisability[12], serialisability[16], etc, but many STM implementers have argued for the need for properties that these criteria do not formally provide. Guerrauoui and Kapałka have argued that none of the standard correctness criteria used in the database world are sufficient for the purposes of STM,

and formally define the *opacity*[5] correctness criterion. Informally, opacity extends serialisability with the constraint that even transactions that eventually abort may not observe a state inconsistent with an alternate serial history of the transactions. In this report I use this requirement, but only informally.

2.3.1 Conflict Detection and Acquire Strategies

A common classification of STM systems is to split them into *eager* and *lazy* STM systems. This involves two separate issues that are often conflated: the acquire strategy, and write logging.

The terms "eager STM" and "lazy STM" take their names from the eager and lazy acquire strategies. When a transactions wishes to write to a transaction variable, it must *acquire* the variable sometime between when the write is discovered (when the programmer's transactional code executes a write) and when the transaction attempts to commit. Acquiring a variable is a globally visible declaration of the transaction's intent to write to the variable, and must prevent any other transactions from reading or writing the variable until it is released by the acquirer committing or aborting. Systems using the *eager acquire* strategy acquire variables as soon as a write is executed, while those using the *lazy acquire* strategy acquire all the transaction's written variables just before attempting to commit.

This has a direct effect on the way in which conflicts between transactions are detected, but also on the nature of such conflicts. The usual definition of conflicts between transactions states that two transactions conflict if they both access the same transaction variable, and at least one of the accesses is a write. I would prefer to refer to these as *potential conflicts*, since a reader and a writer can both commit consistently if the reader commits first.

When writes are acquired eagerly, a transaction that attempts to read a variable after it has been acquired by some other transaction will be prevented from continuing; it must either (a) abort and restart, (b) abort the writer so that it can read the variable, or (c) wait for the writer to release the variable. This means many conflicts between transactions can be noticed early and one of them aborted rather than wasting work executing a transaction that will only abort later, although there is always the possibility that a writer will acquire a variable only after it has already been read. In effect, such STM systems attempt to maintain the property that most active transactions can commit if they do not access any more variables.

When writes are acquired lazily, a reader will only be blocked by a writer that is in the process of committing. As such, most conflicts are only noticed *after* the writer has committed (which causes the reader's observed state to become invalid). In such cases the reader simply has to abort; there is no choice of aborting the writer or waiting until after the writer is done with the variable. This permits two transactions to both continue to run until one of them commits, even if at most one of them can commit, potentially wasting work. There are several advantages to this strategy, however. When two transactions conflict, aborting one of them early is a mistake if the other later has to abort for another reason; the aborted transaction might have gone on to commit if it had been allowed to continue. Also, eager acquire systems detect *potential conflicts*, not true conflicts (a write-write conflict is a true conflict, and two symmetric read-write conflicts make a true conflict); thus they abort (or at best delay) one of a pair of conflicting transactions when under lazy acquire both may have gone on to commit, achieving full parallelism. STM systems using lazy acquire are much less prone to livelock than eager acquire systems. Livelock occurs in STM systems when the attempt to execute some set of transactions repeatedly causes all of them to abort, resulting in no progress being made. When an eager acquire system detects a (potential) conflict between two transactions, one of the transactions will usually abort. The aborted transaction re-executes and is very likely to eventually access the same variable that caused the conflict again. If the other transaction involved in the conflict is still running, a new conflict will be detected, and if the re-executing transaction is allowed to continue this time, livelock is practically guaranteed to result. With lazy acquire on the other hand, a transaction can only cause others to abort when it commits or just before it commits, so a stable livelock situation is almost impossible.

The other issue that is often referred to under the umbrella of "lazy vs eager STM" is the write logging strategy. The first alternative is *redo-logging*, in which a transaction's writes are stored

privately in a log, and "redone" to the real transaction variables when the transaction commits. The other is *undo-logging*, in which a transaction writes directly to transaction variables, keeping a log so that the writes can be "undone" if the transaction aborts. The reason the write-logging strategy is sometimes not identified as a separate issue from the acquire strategy is that only an eager acquire system can support undo-logging, and the ability to use undo-logging has often been the motivation for adopting eager acquire, so that it has been taken as self-evident that an eager acquire system also uses undo-logging.

2.3.2 Blocking and Alternatives in Transactional Code

An STM system provides guarantees on what state will be observed by a transaction only in terms of consistency with a possible sequential ordering. It could also happen that the programmer would like to be able to prevent a transaction continuing based on program level conditions. It would be easy to include a self-abort capability in the interface to the STM system, but a far more elegant construct was developed in Concurrent Haskell[6]. That construct is the retry operation.

A transaction retries only when instructed to by the program. In its simplest form, the retry operation is equivalent to a self abort. However, if the retrying transaction were to re-execute right away, it would be quite likely to observe the same state again, and thus retry again, and so on until a change to a transaction variable finally changes the execution path taken in the transactional code. This would be a form of spin-lock, and would very wasteful. Blocking locks are far more appropriate than spin locks in most situations, and the same reasoning applies here. Therefore, the retry operation blocks the thread executing the transaction. The thread is only re-scheduled when another transaction commits a write to one of the transaction variables read by the retrying transaction, allowing it to possibly take a different execution path. This provides a simple and elegant way for transactional code to block until an arbitrary condition is met.

A further construct introduced in [6] is the `or_else` operator. This allows transactional code to be combined as alternatives. The STM system executes `A` or_else `B` by first running `A`. If `A` completes successfully, then the transaction is considered complete. If `A` retries, however, then `B` is attempted. If `B` also retries, then the entire transaction retries, blocking on the variables read by either `A` or `B`. Of course, either A or B could contain further nested `or_else` alternatives, creating quite complex flow control. This allows blocking implementations of transactional actions to be easily converted into non-blocking ones that return a success/failure indicator, and vice versa.

3 The Original Prototype STM System

My project has built upon Leon Mika's 2007 honours project[15]. In this project he developed an STM system for Mercury, which I will hereafter refer to as the prototype system.

The basic concepts and syntax used for writing transactional code in Mercury were developed by Mika, and I have not changed these aspects of the system. Figure 1 shows a simple artificial example of how these constructs are used. A transaction variable is created with `new_stm_var`, with a starting value. Mercury's view of transaction variables are simply values of the type `stm_var(T)`, which contain a value of type `T`, and can only be accessed with the predicates `read_stm_var` and `write_stm_var`. The *atomic scope* is a new type of goal introduced to Mercury by Mika. It requires two parameters, `outer` and `inner`. The `outer` parameter identifies either a state variable[2] or a pair of regular variables which identify the unique I/O state before and after the transaction is executed. The `inner` parameter identifies the unique "state of the STM system" at the beginning and end of *one particular execution attempt* of the transaction. This STM state serves much the same purpose as the I/O state; it encapsulates the "side effects" of code that accesses transaction

[2]The concept of a "state variable" is a Mercury innovation over Prolog syntax. A state variable, which is written as a regular variable name with **"!"** prepended, stands for two variables wherever it appears (typically an input argument followed by an output argument). This provides a convenient way to "thread" arguments through a series of calls without having to explicitly name all the intermediate values. The compiler interprets the order in which the goals containing a state variable are written as implicitly identifying which outputs and inputs are matched, and thus the order in which the goals must be executed.

```
new_stm_var(0, TVar, !IO),
atomic [outer(!IO), inner(!STM)] (
    read_stm_var(TVar, X, !STM),
    write_stm_var(TVar, X+1, !STM)
)
```

Figure 1: Syntax for using STM in Mercury

variables, the threading of the STM state explicitly provides the execution sequence of the goals that access transaction variables, and its uniqueness prevents backtracking over such accesses.

Mika's implementation provides a lazy STM system with redo-logging, a preference I have also kept. Undo-logging is often an attractive feature for an STM system for an imperative language such as C, since programmers using such languages expect to be able update small parts of large objects in-place. In a pure declarative language such as Mercury, data terms are immutable; most data structures are "updated" by constructing new values and using them in place of the old. When a value is constructed as a small update to a large structure, however, the two values can usually share large parts of their sub-structure. This means that when an immutable value is retrieved from a transaction variable with `read_stm_var`, it is simply a reference to the same data that is still stored in the variable; if the value "contained" in the variable is later updated, the earlier retrieved reference will still be valid. If the data stored in transaction variables were updated in-place, using the undo-log strategy, then `read_stm_var` would have to copy the entire term to be sure it was not later over-written. Thus using undo-logging in a declarative language like Mercury would actually entail *more* copying, and *less* direct access to the values stored in transaction variables.

Mercury does have a concept of unique values that can be destructively updated because the compiler statically guarantees that the updating code has the only reference to the value. While it would be possible to allow unique values to be placed into transaction variables and destructive update to be performed on them, this would be a strange understanding of the "unique" concept; we expect the values stored in transaction variables to be referenced many times, as they constitute shared data, and the "unique" values would also have to have secret references to them kept to be re-instated when transactions abort (which would again involve more copying). Instead, unique values are simply prevented from being placed into transaction variables by the declared mode of `write_stm_var`.

The prototype system also provides an implementation of the `retry` and `or_else` concepts. The implementation of `retry` is simply a predicate that can be called to retry the current transaction; `or_else` would normally be used by the programmer in the form of special syntax in the atomic scope. See Mika's honours report for a more detailed description of this interface.

While the prototype system proved that STM could fit quite well into the Mercury language, it contains a number of deficiencies when considered for practical use. It does not ensure opacity, and so could in some circumstances allow applications to fail to terminate, when an analysis based on an "as-if-serial" understanding of transactional semantics would conclude that this was not possible. Mika's system also incurs significant overheads while executing transactions; a transaction containing n access to transaction variables consumes $O(n^2)$ time searching its own log. It also uses a simple implementation based on a single global lock, which limits achievable parallelism, and provides no possibility for *contention management* (see section 6.3).

My project was to improve the STM system, with the goal of getting closer to a system that was attractive for use in real software projects. In this section I will describe the significant differences of my implementation compared to the prototype. These improvements consist of both utilising more advanced techniques for implementing STM systems (some of which were not widely published at the time of Leon Mika's honours project), and some innovations I have not seen described elsewhere. My most novel contributions are a system of *checkpoints* allowing transactions to sometimes only rollback partially after an abort, and a strategy for logging writes

that avoids the need to search the transaction log when reading variables.

Since this report is mainly concerned with the internal implementation details of STM systems, I do not devote space to the programmer-level interface to the STM system used in Mercury. Leon Mika's honours report[15] has a more detailed description of this interface, which is unchanged in the new system.

4 Non-blocking STM

A major motivation for adopting software transactional memory is to avoid problems associated with lock-based programming. These problems can be divided into two categories. Problems such as deadlock and consistency errors result either from failure of programmers to follow the locking protocol, or from an incorrect locking protocol. Other problems, such as convoying, vulnerability to thread failure, and priority inversion, can occur even if all the locks are correctly used. Using a software transactional memory implementation based on locks avoids the first class of problem (assuming that the STM system is correctly implemented and that a program does not mix the use of locks with transactions), but problems of the second class can still occur. It is therefore desirable to have an efficient non-blocking implementation of software transactional memory.

Non-blocking synchronisation algorithms can be classified according to the progress guarantees they provide. For a system of processes contending for access to some set of shared data:

1. *Wait-free[8]* methods guarantee that all processes will make progress in a finite number of operations. Neither starvation nor livelock can occur.

2. *Lock-free* methods guarantee that some process will make progress in a finite number of operations. Starvation can occur, but not livelock.

3. *Obstruction-free*[10] methods guarantee that any process will make progress in a finite number of operations if no other process performs conflicting operations. Livelock and starvation are both potential problems.

In the context of software transactional memory, making progress can be understood as successfully committing a transaction. These progress guarantees can obviously only be provided for transactions that are themselves always of finite duration.

Wait- or lock-free implementations are obviously desirable, but involve a significant performance trade-off that seems to be unavoidable. The problem is that wait- or lock-freedom requires much more coordination between transactions so that they can observe protocols about which transactions can acceptably be aborted. This in turn requires much greater visibility of a transaction's actions, which tends to make even read-only accesses (which should ideally be fully parallelizable) interfere with each other, as consistency must be negotiated between threads during meta-data updates.

The much weaker guarantees of obstruction-freedom permit more efficient implementations. A common technique is to provide an obstruction-free STM system with a relatively separate *contention manager*, a module responsible for making decisions about which transaction to abort when a conflict is detected. The contention manager can provide pragmatic assurance of progress to a system, even when no theoretical guarantees are possible. Furthermore, it is possible to choose contention management algorithms that do provide the system with the stronger guarantees of lock- or wait-freedom, provided the contention management interface is sufficient. Since contention managers are much simpler and easier to change than the entire STM implementation, and since the optimal contention management strategy is in general workload dependent, the ideal situation seems to be for a base-line obstruction-free STM system to provide a family of contention managers for the application programmer to choose from, or even facilities for writing application-specific contention managers. This then allows the trade-off decision between efficiency and guarantees of progress to be made for each application.

4.1 Consistency Without Locks

The prototype implementation makes use of a single global lock to provide consistency. This lock is held during the final validation and commit/abort phase, thus simply ensuring atomicity. Implementing STM with non-blocking techniques requires careful design, as both validation and attempting to commit are extended operations, and it now cannot be assumed that other transactions are not committing writes while these operations are in progress.

The prototype system has an interesting property that is not maintained by my implementation. I refer to this as *ideal laziness* — the *only* way a transaction ever causes another to abort is by committing, and only if the commit is successful. This means that it is not possible for livelock to occur. A non-blocking system with this property would be lock-free. However, the ideal laziness of the prototype is critically dependent on its use of a single global lock, so it cannot be straightforwardly transformed into an ideally lazy non-blocking system.

4.1.1 Validation

How to perform validation itself is the first challenge. The declarative nature of Mercury makes this slightly different than for object-based STM systems implemented for languages such as C and Java. Those systems typically create a new copy of the data stored in a transaction variable when it is opened for write access; the programs then use normal imperative code to update this new copy in-place, and finally when the transaction commits the new copy is made the current version of the transaction variable. This guarantees that every write will change the address of the object stored in the transaction variable[3].

However, Mercury's data terms are immutable[4]. Consequently, transaction variables are not "opened for read access" and "opened for write access"; reads retrieve values, and writes supply new values. The advantage of this is that only single words are copied in and out of the STM system (above the increased copying naturally displayed by pure declarative languages); for Mercury terms larger than a single word, the word will point to memory cells containing extra data, and these memory cells can quite safely be shared between transactional and non-transactional references to the same value. The disadvantage, however, is that it is entirely possible for a transaction to write a value into a variable that is not just equivalent to one that earlier occupied the same variable, but *is* the exact same value with the same memory address. While seeming harmless enough, this possibility means that validation based purely on the values or addresses stored in transaction variables cannot ensure consistency if other commits can occur during the validation sequence. Consider a transaction T that has reads of V_1 and V_2 in its log, and where V_1 still has the value observed by T but V_2 does not. It is possible that after successfully validating V_1 but before validating V_2 both variables are overwritten, but that V_2 is actually restored to the value T has observed. Observing this value again is a false indicator of consistency; V_1 and V_2 never simultaneously had the values needed. For sound validation, we need to prove the existence of a single moment in time when *all* variables in the log had their observed values. Therefore we must be able to detect whether a write has been committed to a variable, not just whether the variable still has the value we last observed.

There is another consistency problem that must be solved. Some designs for STM systems[7, 20] provide for consistency between transactions that commit, but do not provide the guarantee that transactions that abort will not observe an inconsistent state before they abort. In these designs, a transaction that observes an inconsistent state would be doomed to abort during pre-commit validation, but in the meantime it runs on the basis of inconsistent data; a violation of opacity. The obvious way to prevent this is to validate the entire read set so far before every read, which is very expensive, requiring $O(n^2)$ overhead for a transaction performing n reads.

[3]This is not true for undo-log systems, which write the original data in-place. These systems also cannot trust that a write to a variable previously read will be accompanied by a change in its address. In general an STM system must be designed carefully if changes of address are to be reliable indicators of writes, otherwise the system must use the techniques I discuss, or something equivalent.

[4]I briefly mention in section 3 why the exceptions to this are not relevant to STM.

Techniques for tolerating this inconsistency that have been proposed include trapping exceptions and memory faults, and discarding such errors if the transaction's log is not valid, under the assumption that the error only occurred due to faulty data. Likewise periodic validation of long running transactions could potentially be used to break infinite loops caused by inconsistent data. However, these mechanisms are awkward, and in library-based STM systems for imperative languages cannot completely rule out the possibility that some irreversible operation will be erroneously carried out by a transaction that read bad data. Procedures called in the transaction may even decide to print an error message and terminate the program if they receive unexpected input. It is up to the programmer to be sure that, even for observed states that arbitrarily break normal program invariants, nothing will go wrong. But this is exactly the sort of reasoning that makes concurrent programming with locks so difficult, which STM is supposed to obviate the need for. This is made all the worse because the natural intuitive understanding of transactions that execute "as if serially" promises that such inconsistent states cannot occur. Nonetheless, in the context of a strongly typed declarative language, where data is immutable and I/O operations are statically prevented from occurring inside transactions, there is little a transaction could do based on bad data that can't be undone by rolling back the call stack and letting the garbage collector reclaim all the new values created. Therefore it may be interesting to allow the system to optionally run with weaker consistency guarantees as a performance boost. Even in a language like Mercury, however, non-termination and exceptional conditions such as divide-by-zero errors may occur which the programmer would expect to be impossible from an intuitive idea of transactional semantics. I do not believe requiring programmers to reason about such transient states is acceptable for a realistic STM system, and many STM designers concur.

The solution to both the inadequacy of purely value- or address-based validation and the inefficiency of validating the log on every read, is to use timestamp-based validation[18]. Timestamp based validation uses timestamps for both transactions and variables to reason about whether accesses are consistent. TL2[3] introduced a very efficient scheme based around a global counter. When a transaction begins, it takes the current value of the global counter as its timestamp. When a transaction commits, it increments the global counter and writes the new value of the global counter into the variables it has written, ensuring that their timestamps are greater than those of any transactions that started before the time of the commit. Therefore, if a variable has a timestamp less than or equal to that of the accessing transaction, it must have had its current value when the transaction started. This indirectly proves that it had its value at a time when all the other variables in the transaction's log had the values recorded in the log, since they will have been checked against the same timestamp as well. This is the result required for internal consistency. Note that it is still possible for external consistency to be violated (the other variables in the read set might not still have the values they had at the beginning of the transaction), which will only be detected during commit-time validation, but if the observed state is internally consistent then the transaction will not do anything the programmer could not be prepared for.

In TL2, a transaction simply aborts if the timestamp check fails. This wastes effort re-executing transactions due to easily resolvable "conflicts". If a transaction T with timestamp $T.ts$ reads a variable V with timestamp $V.ts$, and $V.ts > T.ts$, but all other variables in T's log still have their current values, then consistency would not be compromised by allowing the read of V. An alternative is to use extensible timestamps[17]. The intuition is that, if T's log is valid when it reads V, then T is equivalent to a transaction that started now and performed its current work instantaneously. Therefore when a transaction accesses a variable with a larger timestamp, instead of aborting it re-reads the global counter and takes this value as its timestamp, and then performs a full validation with this new timestamp as shown in the pseudo-code of algorithms 1 and 2. The call to `abort` is assumed to not return normally, instead returning to the beginning of the transaction.

We maintain the invariant that whenever a transaction performs a full validation, it first sets its timestamp to a fresh read of the global counter (this step is actually Incorporated into `validate` in the pseudo code and in my implementation). Thus, checking each variable's timestamp against the transaction's timestamp establishes that they have not changed since validation started. Given that, if we also observe that each variable's value matches the logged value then we know that a

Algorithm 1 Pseudo-code for the read operation

```
read(Transaction, TVar):
    Snapshot = current_value_and_timestamp(TVar)
    if Snapshot.ts > Transaction.ts then:
        validate(Transaction)
    Transaction.log.record_read(TVar, Snapshot.value)
    return Snapshot.value
```

Algorithm 2 Pseudo-code for the validation algorithm

```
validate(Transaction):
    Transaction.ts = global_counter
    for each Record in Transaction.log
        Snapshot = current_value_and_timestamp(Record.var)
        if Snapshot.ts > Transaction.ts
           or Snapshot.value != Record.value then:
            abort(Transaction)
```

point in time existed when every variable had its observed value (just before validation started).

This does mean that reading a new variable is no longer an $O(1)$ operation; a pathological schedule of commits can force a transaction to spend $O(n^2)$ time performing validation on each of n reads. However, with TL2's simple scheme the same schedule would instead cause the same transaction to abort n times. Executing n prefixes of a transaction is far worse than validating n times. It takes a transaction slightly longer to abort when the new value is *not* consistent with the existing log, but aborting is a time-wasting operation anyway; it is preferable to optimise for avoiding aborts than to optimise aborts themselves.

During a full validation, the transaction aborts if a logged variable's timestamp is greater than that of the transaction, even if the logged value matches. This corresponds to the case where writes have been committed to the transaction, but the observed value has either remained or been restored by multiple commits. By similar reasoning to that used for an initial read, it would be possible to treat the transaction as an equivalent one that started later, if we establish that all of its reads remain valid at the current time. However, because of the possibility of other transactions also overwriting logged variables while validation is in progress, this would require obtaining a new timestamp and restarting validation from the start. To maintain non-blocking semantics we would have to place an arbitrary limit on the number of times this is permitted to happen. Furthermore, we identify when values are the same by pointer equality, not full value equality, so we can only detect this situation when exactly the same value is restored to a transaction variable. Given the rarity of this case, it is probably not worth attempting this recursive validation. Much of the same effect could instead be had by treating writes that put back the same value as reads, preventing unnecessary conflicts from being detected in the first place.

4.1.2 The Commit Phase

The above process ensures consistency for running transactions. We also need to ensure that a transaction can commit in a way that preserves consistency and atomicity, despite the fact that other transactions are active (and possibly committing) throughout. Zhang et al[26] investigate the issues involved in the design of commit sequences for timestamp-based STM systems, and present several specific commit sequences. The commit sequence I have implemented in my system, which I describe below, is based on the V4 algorithm from their paper.

The timestamp-based strategy I have presented requires that when a transaction commits the global counter must be incremented, and that all to-be-written transaction variables have their timestamps updated. A transaction must also prove that all the variables it has read still have

their observed values. Finally, it must make the new values of its writes the current values of those variables, but the combination of validating reads and updating writes must be an atomic operation.

The semantics of *acquiring* transaction variables is crucial here. Simply calling validate at the end of the transaction establishes that a point in time existed when all variables read had their observed values. But that point in time was just before validation started; even during validation some of the read variables may be overwritten by other transactions' commits. This opens up the possibility that consistency will be violated in other transactions' reads.

I will illustrate with an example. Consider transactions T_1 and T_2, and transaction variables V_1 and V_2. T_1 is in the process of committing, and has validated its reads, including V_1, but has not yet finalised its commit. When it does it will commit a write to V_2. At this point V_1 is overwritten by another commit, and then T_2 reads both V_1 and V_2.

The key question is what should happen when T_2 reads V_2, in order to maintain a consistent as-if-serial order of transactions? Since T_2 has read a value for V_1 that is later than the value of V_1 used by T_1, in the serial ordering it must come after T_1, and therefore it should read the value for V_2 that is written by T_1. But this assumes that T_1 does in fact commit, which is not yet established; another transaction could abort T_1. Until T_1 commits or aborts, the logical state of V_2 is unstable, depending on whether T_1 *will* commit or abort in the future. T_2 could itself abort T_1, in order to fix this state, but T_1 is in its commit phase and has already completed its commit-time validation; it is almost guaranteed to successfully commit, and is a much safer 'bet' for the system to give preference to. If this is the first time T_2 has read V_2, then T_2 does not depend on any particular value for V_2, it just needs to get one. Waiting a little to give T_1 the chance to commit has a high chance of not requiring either transaction to abort, but T_2 must be allowed to abort T_1 eventually, or non-blocking progress has been compromised (consider what would happen if the thread running T_1 crashed in this state). Deciding exactly what to do in this case is the problem of contention management. Algorithms 1 and 2 have not taken into account the possibility of acquired variables; they need to be modified so that reading the current value and timestamp of a variable can fail, and if this happens the contention manager is invoked. I discuss this more in section 6.3.

This is the reason that the status of having acquired a transaction variable must represent not just globally visible exclusive permission to *write* to that variable, but must also prevent any other transactions from *reading* that variable as well. This property makes it possible for a transaction to establish that it is safe for it to commit, and go on to do so, regardless of what other transactions do after it has established safety (unless they actually abort it). This includes the possibility that a transaction T may commit at a later point in real time than other transactions who overwrite variables in T's read set. These other transactions will have later timestamps, even though they actually complete their commits first. Perhaps counter-intuitively, this possibility is actually safe, provided T acquires the variables in its write set *before* attempting to validate its read set. Validation establishes that at some point t in time T's read set was valid; if the variables in T's write set were acquired before t, then only transactions that *do not depend on any variables in T's write set* can have committed after t, and so T can be either inserted into the serial ordering before these other transactions or left out, depending on whether it eventually commits or aborts, without compromising consistency.

The use of a global counter that is updated when transactions commit presents an opportunity for optimisation. Validation during commit does not necessarily need to be a full run of the `validate` algorithm discussed in the previous section. If the transaction observes that the global counter still matches the value of the transaction's timestamp, then no other commits have taken place since this timestamp was obtained. Since a validation was performed when the timestamp was obtained, which proved that the transaction was valid at that time, and no other writes have been committed since then, the transaction must still be valid. In these cases "validation" during commit can consist of observing the counter. If the counter does not match the transaction's timestamp, then a full validation must be performed.

The global counter is a potential bottleneck; since every committing transaction must modify it, transactions executing their commit sequences in parallel are forced to serialise at this point.

We would like to reduce this overhead if possible. One obvious way to avoid doing this is to not update the counter for read-only transactions and for transactions that abort, since neither group should have any effect on other transactions. In fact, since read-only transactions have no writes that need acquiring or new timestamps and cannot invalidate any other transaction, they do not need an extended commit phase at all — provided they have ensured *internal* consistency while in-progress, they simply need to check one last time that they haven't been aborted by some other transaction.

If we want to avoid updating the global counter for a transaction that aborts, transactions must validate before the counter is updated. However, there is a subtle problem with this idea when combined with the idea of skipping validation if the global counter hasn't changed. Consider two transactions that both attempt to commit and have both acquired all of their variables. They next need to prove that their reads are still current. If both transactions last validated (obtaining new timestamps) after the most recent commit by any transaction, both can observe the global counter to still match their own timestamp, and may decide to skip validation. But if there are symmetric read-write conflicts between them (each has read a variable the other has written), then they cannot both commit! We need at least one of them to validate in order to notice the other's acquire. The key issue is that the status of having acquired a variable is supposed to be a globally visible property. But if it is possible for a transaction to commit without running a full check on the variables it has read, then the acquire is not globally visible. Essentially, now the process of acquiring variables cannot be considered complete *until the global counter is updated.* This forces other transactions to run a full validation in their commit phase, which ensures they will notice the committing transaction's acquires. But since validation in the commit phase must take place *after* variables have been acquired, this implies that we cannot validate before updating the counter!

Another way of reducing contention on the shared counter is to allow some transactions that commit in parallel to share timestamps. A transaction T attempting to commit must update the global counter using a compare-and-swap operation[5]. Before that it must have read the global counter to calculate the incremented new value; call the time at which the global counter was read time t. If the compare-and-swap fails, then some other transaction has incremented the counter since time t, and so a full validation must be performed to ensure that T notices any of its acquires and committed, as discussed earlier. However, if T had already acquired all of its variables before time t, then the update that has been done by some other transaction was also after T acquired its variables. This update will suffice to make later transactions validate and notice T's acquires — we don't need to repeatedly try to increment the global counter until we succeed. This allows a set of transactions to commit with the same timestamp. For this to happen, though, all of them must have completed their acquires and read the global counter before any of them attempt to update the counter. From this state, the first transaction to attempt to update the counter, T_0, will be successful, and may have concluded that it doesn't need to perform a full validation, and so miss any acquires by other transactions in the set. Therefore, in the serial order of transactions, T_0 must come before any others in the set. The rest of the transactions in the set will all fail to update the counter, and will run a full validation. Since they have all already acquired all their writes before any of the validations started, they are guaranteed to be completely independent from each other if the validations succeed, and also guaranteed not to depend on any variables written by T_0. Therefore any serial order putting the remaining transactions after T_0 is consistent, and all of them may commit safely.

The issues I have discussed above are the origin of the trade offs explored in the various commit sequences of Zhang et al[26]. The strategy adopted in their V4 algorithm, which according to their results performs well in many conditions, is the only one I have implemented thus far, and is shown in algorithm 3. After writes are acquired, the timestamp is speculatively checked and a validation run if necessary. Then the timestamp is incremented; if this fails then a second validation is

[5]Compare-and-swap is a machine-level atomic operation. It takes as parameters an address, an old value, and a new value. If the old value is still stored at the address, it stores the new value there and returns true — this is done atomically so the value stored at the address cannot be modified in between. Otherwise it leaves the value stored at the address unchanged and returns false.

Algorithm 3 Pseudo-code for the commit phase

```
commit(Transaction):
    acquire_writes(Transaction.log)
    SnapTS = global_counter
    if Transaction.timestamp != SnapTS then:
        validate(Transaction)
    Transaction.timestamp += 1
    if not compare_and_swap(&global_counter, SnapTS,
           Transaction.timestamp) then:
        validate(Transaction)
    update_acquired_vars_timestamps(Transaction)
    commit_and_release_writes(Transaction)
```

necessary. Note carefully that `validate` has the side effect of reading the global counter into `Transaction.timestamp`! Therefore just before the compare-and-swap, `Transaction.timestamp` is either equal to the value of the global counter read into SnapTS, or to the value of the global counter read during validation. The timestamp written into the acquired variables just before they are released is required to be some value greater than the timestamp that existed when they became unavailable for reading, so that transactions with timestamps earlier than this will re-validate upon reading the new values. This is satisfied by the transaction's timestamp at the point `update_acquired_vars_timestamps` is called. It is assumed that `acquire_writes`, `update_acquired_vars`, and `commit_and_release_writes` are appropriately implemented to call abort rather than return if they cannot complete successfully.

An important note here is that this commit phase sometimes validates twice. The first is an attempt to discover that the transaction is invalid before updating the global counter. This is likely to speed up an application consisting of many very short transactions. In such an application validation is short, so it is unlikely that the global counter is updated by another transaction, so the first validation is likely to be able to serve as the single necessary one during commit. For longer transactions however, it is probably more important to avoid validating twice than to avoid serialising transactions unnecessarily on the counter — furthermore, the validation of longer transactions takes longer, greatly increasing the chance that the global counter will be modified during the first validation, so longer transactions are both more affected by double validation and more likely to do so. There is no reason why all transactions need to use exactly the same commit sequence, and the size of the transaction log can be consulted during commit. Therefore, if an appropriate cut-off point were identified, it would be possible to have smaller transactions use the given algorithm to commit, and longer transactions use a slightly different algorithm that guarantees they will validate only once.

4.2 Data Layout

So far my description of the STM system has treated transaction variables themselves rather abstractly. My discussion has just assumed that current values and timestamps can be retrieved from them, that they can be acquired, and that updated versions are atomically released when a transaction commits. In this section I discuss how transactional meta-data is organised in the system, so as to achieve these properties. The prototype system's use of a single global lock and lack of timestamp-based validation allows transaction variables to be represented as simply a reference to the value they contained. A non-blocking system requires a more complicated representation. The scheme I have chosen is similar to those of RSTM[14] and DSTM[11], and is shown in figure 2.

A value being manipulated in the STM system is always stored together with a timestamp. The data value itself is treated as completely opaque by the STM system, and is an ordinary Mercury value. The Mercury implementation allows all values to be passed around as a single

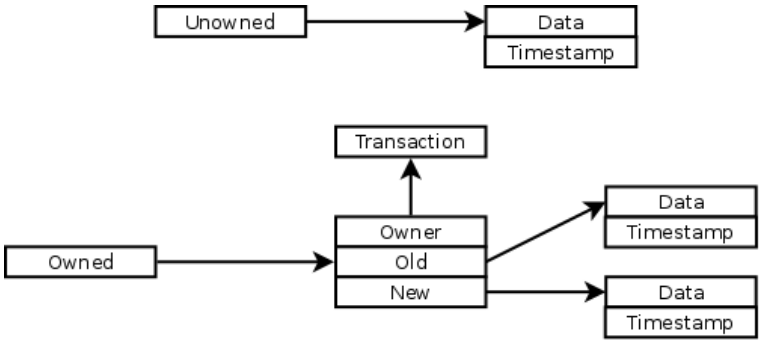

Figure 2: Data layout in the STM system.

machine word; the interpretation of that word is type-dependent, and may involve references to memory cells elsewhere. Mercury's pure declarative nature ensures that these other memory cells will never be modified[6], so the STM system does not need to worry about them. Combined with this fact, Mercury's strong, statically checked, polymorphic type system allows the STM system to forgo recording the types of the data stored; the type checker is responsible for ensuring that a given transaction variable is always used to retrieve and store values of the same type.

Transaction variables are represented by a variable header containing a data reference, and can be in either the *owned* or *unowned* states. The owned or unowned status of the transaction variable is indicated by a tag in the low-order bit of the data reference, so a single atomic instruction can update both the status and the data reference.

In the unowned state, the variable header's data reference points directly to the current logical value and timestamp of the transaction variable. Identifying the current logical value of an owned transaction variable is more complicated. In the owned state, the data reference instead points to an *ownership record*, which identifies old and new versions of the variable's value and timestamp, and the *transaction descriptor* of the transaction that currently owns the variable. A transaction descriptor stores the meta-data needed by the thread running each transaction (including the log and the timestamp, which we have seen in the discussion in section 4.1). It contains only one "public" field, which may be both read and written by other transactions — the transaction's *status* (other private fields of the transaction descriptor are not shown in figure 2). This status is one of `ACTIVE`, `ABORTED`, and `COMMITTED`. If the owner's status is `COMMITTED`, then the new value of the variable is current. If the owner is `ABORTED`, then the old value is current. If the owner is `ACTIVE`, then the transaction variable is acquired and must not be accessed without first aborting the owner, in order to ensure consistency. However, a transaction always uses the new value if it is itself the owner.

After identifying whether to follow the new or old reference however, a transaction accessing an owned variable must then double-check that the variable header still points to the same ownership. This prevents a race in which, after the ownership is identified but before the owner's status is checked, the thread that ran the owner begins a new transaction (or makes another attempt at the same transaction) and reuses its transaction descriptor.

To acquire a variable, a transaction allocates a new data record containing the value and timestamp of the new version (the timestamp may not be final, but no other transaction will read it until and unless this transaction commits). It then allocates a new ownership record pointing to its own descriptor, the data record of the "current" version of the variable, and the newly constructed data record. It then uses a compare-and-swap to change the data reference in the variable to a pointer to the ownership record, with the low-order bit of the pointer indicating the variable is owned.

When a transaction commits, it need only use a compare-and-swap to change its status field to

[6]Under some circumstances Mercury allows data to be modified in-place, but such data is not permitted to be stored in transaction variables. See section 3 for more discussion.

COMMITTED. This has the effect of simultaneously releasing all acquired variables and changing their logical values to the values written by the transaction. Likewise, when a transaction aborts it sets its status to ABORTED, releasing any variables it had acquired. To abort another transaction, its status is compare-and-swapped to ABORTED. It will not notice that it has aborted until the next time it calls an operation provided by the STM system, but setting its status to ABORTED instantly frees up all of its acquired variables for use by other transactions.

After a transaction commits or aborts, it enters a cleanup phase, in which it sets all of its owned variables back to the unowned state (with the appropriate value). This phase is not necessary in that failure to perform it does not impede any other transaction, but the just-finished transaction is likely to perform this quicker than other transactions, as the cache of the CPU that has been executing it is likely to contain many of the variables the transaction has been accessing (this is particularly true for a transaction that has just committed, as the commit sequence manipulates the meta-data of all acquired variables just before finalising the commit). Any other transaction that accesses an owned variable in a final state (with a committed or aborted owner, possibly because it has just aborted the owner) performs the cleanup itself, early. This incurs no additional overhead in the case of a new transaction wishing to own the variable — it is just a matter of where the old value of the new ownership record comes from. If an owner's compare-and-swap fails when attempting to clean up after itself, this indicates that some other transaction has accessed the variable in the meantime, so the owner does nothing and moves on. The owner's cleanup phase *is* however necessary before beginning a new transaction and reusing a transaction descriptor. The thread must ensure there are no more references to its transaction descriptor so that its new status is not mistaken as identifying the version of its previously finalised writes.

I have previously made reference to situations that must be treated carefully because of the possibility of a transaction descriptor being reused. An obvious way to avoid this issue would be to simply allocate a new transaction descriptor for every transaction, letting the garbage collector deal with the old ones when they are no longer referenced. The main reason for not doing this is to allow for contention management strategies that need book-keeping which persists across aborts — simple and obvious strategies such as increasing a transaction's priority if it repeatedly aborts would be impossible if every re-execution of a transaction began with a newly allocated descriptor.

There is an issue with this scheme for a system that uses lazy acquire. If a transaction reads a variable it has previously written, but not yet acquired, it should return the previously written value, not the current value as defined above. The usual approach is to search the write set before every read, to see if the variable has already been written. Spear at al[23] suggest that evaluation of the trade-off between eager and lazy acquire "appears to have been clouded by the fact that many comparisons between eager and lazy STM have used TL2[3] as the lazy representative", and cite lookups in TL2's linear write buffer as a major source of overhead (even though TL2 uses a bloom filter to try to avoid these lookups). They show that if the write set is implemented as a data structure with efficient lookup characteristics, such as a hash table, then the performance of lazy acquire systems can match that of eager acquire systems.

I have devised an alternative method of solving the same problem. The method for obtaining the current logical value of a transaction variable must already be prepared to find an owned variable and determine which version to use based on the owner. We could use this capability to find previously written values where we ordinarily look for the values of variables if a transaction was able to *own* a variable without *acquiring* it. This requires differentiating between two different kinds of active transactions, and so we replace the `ACTIVE` transaction status with two new status codes: `ACTIVE_LAZY` and `ACTIVE_EAGER`. When a transaction reads an owned variable it treats an `ACTIVE_LAZY` owner the same as an `ABORTED` owner; it uses the old value (unless, of course, it is itself the owner).

It is very easy to support both eager and lazy acquire with this scheme. Lazy transactions begin in the `ACTIVE_LAZY` state and acquire all their writes in the commit phase simply by using compare-and-swap to update their status from `ACTIVE_LAZY` to `ACTIVE_EAGER`. Eager transactions simply begin in the `ACTIVE_EAGER` state. The decision to use eager or lazy acquire could even be made on a per-transaction basis, though I have not examined the usefulness of such a feature.

The "lazy acquire" gained by this scheme is not identical to the normal meaning of lazy acquire,

however. It allows any number of readers to execute in parallel with a single writer, but conflicts between writers must be resolved early, as only a single transaction can own a variable at a time. It would be possible to allow multiple writers to co-exist by having either the new or old pointers in an ownership record contain a tagged data reference as in the variable header itself, thus possibly forming a chain of ownership records. The overheads of such a system would be intolerable, however, and the benefit very small. The chief advantage of lazy acquire is that it allows readers of a variable to execute in parallel with writers of the same variable, as some of the readers may commit before a writer commits and invalidates them, achieving parallel use of the same data. The advantage of allowing multiple writers to execute in parallel is much more questionable, as at most one of them can commit and the time spent speculatively executing the rest is wasted work. The only potential advantage lazy acquire can claim is that aborting all but one of these transactions early will increase wasted work if the transaction that is allowed to continue later aborts for another reason, while one of the aborted transactions might have been able to commit. Spear et al refer to this "partially lazy" acquire as *mixed invalidation*[24], and describe a system for implementing it that is very similar to the one I have described here. Their discussion approaches the concept specifically with the goal of detecting write-write conflicts eagerly while delaying the detection of read-write conflicts. They do not discuss the impact of this mixed invalidation on write set lookups, though it does not seem likely that their system would look up writes in a separate write set when it could find them from the transaction variables themselves.

A further advantage of using this representation is that the write set can be implemented as a simple linear data structure without compromising performance. Since now the only thing it is used for is a traversal of *all* writes (during the commit phase, or when cleaning up after an abort or commit), there is no benefit to it being anything other than a simple extensible array. There is a small performance penalty, in that reading owned variables is necessarily more complicated than reading unowned ones, and variables are owned much more of the time with this scheme than with true lazy acquire. However I expect this to be more than compensated for by avoiding lookups in the write set, which would affect *all* transaction reads regardless of whether any other transaction is interested in writing the variable.

5 Partial Roll-back with Checkpoints

The prototype Mercury STM implementation includes the `retry` and `or_else` constructs originally developed in Concurrent Haskell[6]. I have described the meaning of these in section 2.3.2.

Consider a transaction which reads and writes some transaction variables, then enters a nested transaction. The nested transaction contains a call to `retry`, and has `or_else` alternatives. When the `retry` is reached, transaction execution must roll-back, but only to the beginning of the nested transaction. Likewise, writes must be abandoned before the execution of the `or_else` alternative, but only those of the nested transaction, not its parent. This requires that the nested transaction's log is kept separate from that of the parent in some way. These capabilities can be generalised and used even without the presence of nested transactions making use of `retry` and `or_else`, to provide a "checkpoint" system for transaction execution. The basic idea is to split the log into segments, chained together, and validate them in chronological order. When validation of a particular log entry fails, it is known that every earlier log segment contained only valid log entries. Therefore, if we could roll-back the transaction to the state it had just before the invalid log segment was created, we could save repeating the work done in some prefix of the transaction.

In both the prototype implementation and my system, the mechanism used to implement transaction roll-back is exception handling. Algorithm 4 shows pseudo-Mercury for a procedure that might be generated from an atomic block in the user's program. It is greatly simplified (and sweeps under the rug complexities of passing the unique STM state across the exception handling interface), but shows the essential logic. Appendix B shows the actual code used to implement a simple transaction in both the prototype and new STM systems, which gives a fuller appreciation of the complexities involved.

A new predicate must be generated corresponding to each atomic scope, so that recursion can

Algorithm 4 Transaction roll-back by exception handling

```
atomic [outer(!IO), inner(!STM)] (
    do_something(Input, Output, !STM)
or_else
    do_something_else(Input, Output, !STM)
)

⟹

atomic_scope_0(Input, Output, !STM) :-
    try (
        do_something(Input, Output),
        commit(!STM)
    catch stm_exception_abort ->
        discard_log(!STM),
        atomic_scope_0(Input, Output, !STM)
    catch stm_exception_retry ->
        discard_writes(!STM),
        try (
            do_something_else(Input, Output, !STM),
            commit(!STM)
        catch stm_exception_abort ->
            discard_log(!STM),
            atomic_scope_0(Input, Output, !STM)
        catch stm_exception_retry ->
            block_until_read_vars_change(!STM),
            discard_log(!STM),
            atomic_scope_0(Input, Output, !STM)
        )
    ).
```

Algorithm 5 Checkpointed execution for single transaction goals

```
stm_goal(Goal, Output, !STM) :-
    create_checkpoint(Checkpoint, !STM),
    try (
        Goal(Output, !STM)
    catch stm_exception_abort(CaughtCheckpoint) ->
        ( if CaughtCheckpoint = Checkpoint then
            rollback_to_checkpoint(Checkpoint, !STM),
            stm_goal(Goal, Output, !STM)
        else
            throw(stm_exception_abort(CaughtCheckpoint))
        )
    ).
```

implement re-execution. The code originally contained in the atomic scope is executed inside an exception handler, allowing the STM system to simply throw an `stm_exception_abort` exception whenever it detects a reason to abort. Likewise the implementation of the `retry` predicate is to simply throw `stm_exception_retry`. The prototype system generates code to implement this logic when compiling each atomic scope, and consequently atomic scopes with multiple `or_else` alternatives generate chains of nested exception handlers inside the retry arms. My system instead implements this logic in higher order predicates in the Mercury library, so all the compiler is responsible for is factoring out the contents of atomic scopes into separate predicates and constructing a few higher-order calls. Higher-order code is often less efficient in Mercury than normal code, so it might seem preferable to generate specialised versions of this code for each atomic scope, as in the prototype system, but exception handling in Mercury is implemented by passing a higher-order term to a `try` predicate, so there is no getting out of it. I do arrange in my system that the closures constructed to be passed into the STM system are the very same ones that are passed into the exception system, so that no additional overhead is incurred constructing closures that wrap closures.

Note that nested transactions need a different structure; aborts should be uncaught so that they propagate to the topmost level, and retries are not caught in the final or_else alternative. Only the outermost transaction should contain the code to wait and then re-execute. In the prototype implementation, `commit` called by a nested transaction merges its log entries back into the parent's log. My system does not call commit in a nested transaction.

It should be immediately obvious that when execution is inside multiply-nested transactions, there will be several layers of exception handler live on the call stack. Each level of transaction will already have its own log, so when an inconsistency is found during validation in a log with depth N, the system has enough information to conclude that the prefix of the transaction executed up to the creation of a particular exception handler is still valid. To avoid re-executing that prefix, we only need a way of targeting a thrown exception to a particular handler. The mechanism for doing this is shown in algorithms 5 and 6, this time using higher-order code that is closer to the library code found in my implementation (the pseudo-code shown in algorithm 4 resembles the generated code produced by the prototype implementation). The key point is that `stm_exception_abort` now has a parameter, indicating which checkpoint should catch the exception. Internally, a checkpoint is in fact identified with a log segment; when the validation routine in the runtime finds an invalid entry it returns the address of the log segment that contained the entry as a "checkpoint". The library then throws an exception containing that checkpoint, which will be caught by the exception handler set up at the point where the checkpoint (log segment) was created. `rollback_to_checkpoint` discards all log segments from the indicated checkpoint onwards (including the checkpoint itself, which contained the first invalid entry), ready for a new one to be created again in the recursive call.

A point that may not be immediately appreciated is that the "rollback" that happens when

Algorithm 6 Checkpointed execution for `or_else` alternatives

```
or_else(Goal, OrElseGoal, Output, !STM) :-
    create_checkpoint(Checkpoint, !STM),
    try (
        Goal(Output, !STM)
    catch stm_exception_abort(CaughtCheckpoint) ->
        ( if CaughtCheckpoint = Checkpoint then
            rollback_to_checkpoint(Checkpoint, !STM),
            or_else(Goal, OrElseGoal, Output, !STM)
        else
            throw(stm_exception_abort(CaughtCheckpoint))
        ),
    catch stm_exception_retry ->
        disown_writes(Checkpoint, !STM),
        or_else_retried(Goal, OrElseGoal, Checkpoint, Output, !STM)
    ).

% Encapsulate the second exception handler, which would otherwise
% be in the stm_exception_retry arm above, making it harder to read.
or_else_retried(Goal, OrElseGoal, OuterCheckpoint,
        Output, !STM) :-
    create_checkpoint(Checkpoint, !STM),
    try (
        OrElseGoal(Output, !STM)
    catch stm_exception_abort(CaughtCheckpoint) ->
        ( if CaughtCheckpoint = Checkpoint then
            rollback_to_checkpoint(Checkpoint, !STM),
            or_else_retried(Goal, OrElseGoal, OuterCheckpoint, Output, !STM)
        else if CaughtCheckpoint = OuterCheckpoint then
            rollback_to_checkpoint(OuterCheckpoint, !STM),
            or_else(Goal, OrElseGoal, Output, !STM)
        else
            throw(stm_exception_abort(CaughtCheckpoint))
        )
    ).
```

Algorithm 7 Pseudo-code for the entry-point predicates

```
atomic_transaction(Goal, Output, !IO) :-
    get_stm_state(STM0, !IO),
    top_level_transaction(Goal, Output, STM0, STM),
    set_stm_state(STM, !IO).

top_level_transaction(Goal, Output, !STM) :-
    try (
        stm_goal(Goal, Output, !STM),
        commit(!STM)
    catch stm_exception_abort(_) ->
        rollback_to_top(!STM),
        top_level_transaction(Goal, Output, !STM)
    catch stm_exception_retry ->
        block_until_read_vars_change(!STM),
        rollback_to_top(!STM),
        top_level_transaction(Goal, Output, !STM)
    ).

nested_transaction(Goal, Output, !STM) :-
    stm_goal(Goal, Output, !STM).
```

a transaction retries before calling or_else_retried does not discard the log segment. Variables written by the goal that retried are reset, as the effects of these writes must neither be seen by the alternative goal nor eventually committed, but the log segment is left with its reads intact. This is necessary in case the alternative goal also retries; if the entire transaction retries, then it must block until the values of any of the variables it read are changed. Thus, a transaction that attempts each of a number of or_else alternatives in turn executes with an increasingly long chain of log segments. The chain cannot grow without bound for a finite transaction, as it resets whenever the entire transaction retries, but the presence of this chain and the checkpoints it represents could significantly reduce wasted work when variables read by a recent alternative goal are invalidated by other transactions. The "abort to top" strategy would require re-executing *all* alternative goals (which will do exactly the same thing as before, and so retry) until one is reached that read the invalidated variable. Checkpointing allows execution to restart immediately at the particular alternative goal that needs to be re-executed.

The "entry-point" predicates are shown in algorithm 7. `atomic_transaction` is the entry-point for running an STM transaction as an I/O action, while `nested_transaction` is an STM action. These two predicates, along with or_else seen in algorithm 6, are the interface that programmers could use to run transactions; the atomic scope syntax, which is usually more convenient, can be easily transformed into calls to these predicates by the compiler (although I have not yet implemented the required changes to Mika's source-to-source transformation to target this simpler interface instead of the prototype system, so atomic scopes are currently broken).

The predicate `top_level_transaction` is not visible to users, but implements most of the actual logic of `atomic_transaction`, which is why I have presented it along with the "entry point" predicates. The `atomic_transaction` predicate merely "translates" between the I/O state and the STM state, while `top_level_transaction` contains an extra layer of exception handling, to implement the blocking semantics of retry when all `or_else` alternatives have been exhausted, and is the target of recursion to implement re-execution.

Programmers can add checkpoints to their transactions simply by introducing nested transactions. Indeed, this is the only real purpose of a nested transaction with no `or_else` alternatives, which would otherwise be equivalent to just directly executing the nested transaction's code (this is exactly how the prototype system views and implements nested transactions with no alternat-

```
try(Goal, Result),
(
    Result = ok(Output)
;
    Result = exception(E),
    ( if E = ... then
        ...
    else if E = ... then
        ...
    else
        rethrow(Result)
    )
)
```

Figure 3: Typical pattern for using the try predicate

ives). The STM system is transparently capable of rolling back to the beginning of any nested transaction, or the beginning of any `or_else` alternative.

5.1 Impure STM Goals

As described above, the checkpoint capability is, unfortunately, fairly ineffective. The major reason for this is the location of the call to `commit`, inside `top_level_transaction`. Transactions perform their most stringent validity checks during the commit sequence, and so this is the most likely time for aborts to be discovered. At the point commit is called, however, execution has left the scope of all the exception handlers set up to catch checkpointed aborts; there is no alternative but re-executing the entire transaction from the beginning (indeed, there is little point to implementing an abort from here by throwing an exception). Checkpoints are only of any use in reducing wasted re-execution work in the case when an abort is discovered during transaction execution. Even then, if execution has left the nested transaction that needs re-executing, the abort must be sent to a "higher" transaction that is still executing. More formally, execution will restart to the beginning of the inner-most transaction that encloses both the earliest location that accessed a variable which has been invalidated, and the location of the current execution.

In section 6.4 I discuss a radically different way of executing transactions that greatly improves the utility of checkpoints, which I have designed but not implemented. My system as implemented does include a refinement of the system outlined above, however, which gains a small improvement to the chance of a checkpoint being applicable, and also reduces the overhead of using the exception handling machinery to implement transaction roll-back.

In the algorithms presented in the last section, I used a convenient high level notation to express exception handlers using try and catch. Mercury does have very similar syntax, but this is translated by the compiler into code that uses the higher-order `try` predicate. This predicate takes a higher-order closure, which must produce a single output argument. An example of how this is typically used is shown in figure 3.

If more than one output is needed, the closure must wrap them in a tuple; the Mercury compiler can automatically transform the code used in a try goal so that it does this. This exact same transformation is needed for code inside an atomic scope, and for the same reason; the higher-order predicates that are used by the underlying implementation need a closure that only has one output argument. In the case of the atomic scope, however, the closures also need to take a pair of arguments (one "destructive input" and one "unique output") for the STM state. This means that when the closure is passed into the exception handling system, another closure has to be generated with only a single output argument; this argument is a pair containing the output STM state argument wrapped up together with the output argument of the STM closure, which is itself probably a tuple wrapping the outputs of the programmer's original code. Constructing multiple

layers of closure containing multiple layers of tuple-packing is needless work, and may interfere with the compiler's ability to optimise away some of the inefficiencies of using higher-order code.

The way I have addressed this issue paves the way for increasing the effectiveness of checkpoints, at the expense of considerably messier code inside the STM library module, though it implements substantially the same algorithms as outlined in the previous section.

In algorithm 7, it can be seen that the "source" of the STM state, which is threaded through the transaction system and the user's transaction code, is a call to `get_stm_state`, and it is eventually "consumed" by `set_stm_state`. All `get_stm_state` does is retrieve a pointer to the statically allocated transaction descriptor for the executing thread, while `set_stm_state` is a no-op (it exists mainly to avoid depending, at this level, on the fact that transaction descriptors are statically allocated). If these calls could be moved *inside* the individual transaction goals, the STM state wouldn't have to cross the boundaries of the exception system at all, and the STM closures would have just one output argument, matching up exactly with the exception handler's closures.

Since the transaction goals do not have access to the I/O state, doing this requires `get_stm_state` and `set_stm_state` to be impure. This has the further results that the STM closures, and consequently the interface predicates `atomic_transaction`, `nested_transaction`, and `or_else`, are also impure. Calls to them can be pure only when the closures provided would be pure if it were not for the call to `get_stm_state`. All of the predicates implementing the "scheduling" logic of the STM system must now be impure as well; the required impurity annotations and the loss of conceptual clarity in the implementation are the main cost of this strategy; any compiler optimisations that are prevented by this impurity would have been inapplicable due to the threaded STM state variables anyway.

Since using these predicates correctly is more difficult, and users of the STM system obviously shouldn't know about an internal detail like `get_stm_state`, I have introduced parallel versions of the interface predicates. There are pure versions that take closures which need STM state arguments, and construct the corresponding impure closures before passing them on to the rest of the STM system; these versions have all the potential efficiency issues outlined above. There are also impure versions of the three interface predicates, contained in a "private interface" which will not be documented in the Mercury library documentation. These versions take impure closures which have only an output argument, and assume the closures themselves call `get_stm_state` and `put_stm_state`. These predicates should not be used by users directly; rather they will be used in the code generated by the compiler from atomic scopes. The compiler will insert a call to `get_stm_state` into the predicates it generates that contain the user's code, along with the output tuple-packing.

Since there is now a "secret" interface that relies on being used correctly by the compiler, I also introduce the notion that calls to `commit` should be moved into the STM goals themselves. The predicates which are used as alternatives at the top level of a transaction (i.e. all those appearing as the main goal or an `or_else` alternative of an top-level atomic scope) will have a call to `commit` inserted into them by the compiler. This means that when a top-level transaction with a number of `or_else` alternatives attempts to commit to any one of those alternatives, it will still be within the scope of all the checkpoint exception handlers. This additional detail makes the impure interface dangerous for anyone other than the compiler to use directly, as the behaviour of the system will be very strange if closures contain calls to `commit` when they shouldn't, or vice versa. A closure should contain a call to `commit` if and only if it *eventually will be* passed to `atomic_transaction` rather than `nested_transaction` (possibly indirectly, via `or_else`, which is the same for both top level and nested transactions), which may not be obvious at the point the closure is constructed if this interface is used in general ways by the programmer.

5.2 Limitations of the Checkpoint System

Overall, even with the enhancement from using impure STM goals, the checkpoint system is only of limited use. It only provides a significant benefit with long and complex transactions under heavy contention, when many "large" aborts are happening. While this could potentially increase

the level of contention for which an application with complex transactions is still usable, it does not help the fast path of executing small non-contending transactions.

While the cost of the extra log segments themselves is small, the added complexity of the exception handlers could have a much more significant cost. A programmer who wishes to avoid unnecessary exception handling scopes can always do so, by calling sub-goals directly rather than putting them inside nested transactions, but when multiple exception handlers are live on the stack the abort process requires each handler to catch the exception and check to see whether execution should restart at that level, or be thrown further outward. All those exception handling scopes must also be passed "on the way out" when no exception is raised. This overhead is incurred for every checkpoint, regardless of whether it actually prevents re-execution work. Without sufficient cases where work *is* saved, the application will run slower than if it had used fewer checkpoints.

Experimental evaluation is needed to analyse the costs and benefits of using checkpoints as implemented by this system. Whether they help or hinder performance is probably highly application-dependent. Fortunately the cases where something equivalent to the checkpoint system is *necessary* correspond exactly to those cases where a nested transaction is the only way to express the meaning of the program (i.e. when using nested transactions with `or_else` alternatives); in other cases, the programmer can choose whether or not to use a nested transaction based on whether they expect a checkpoint to be of use. Thus it should be safe to always create a checkpoint for every nested transaction.

6 Planned Future Extensions

I have implemented a system with the features described in detail in the previous sections. This implementation is not quite fully functional, however, and some more work is needed to bring it to a minimal state of completeness.

One problem is that the source-to-source transformation implementing the atomic scope syntax has not yet been changed to produce calls to the higher order library predicates instead of the large body of generated code emitted by the prototype; this change will be fairly mechanical, as the information used in the transformation is exactly the same, so the general framework can be reused.

The use of `retry` and `or_else` alternatives is not fully functional either, primarily because the blocking mechanism does not work with the new implementation. Again, this will be substantially similar to that in the prototype system, the existing mechanism just needs to be appropriately worked into the structure of the new implementation.

A much more critical problem is that there is a race-condition bug in my implementation of this system, which I discovered when I attempted to benchmark the performance of the STM system. Updates performed by transactions are occasionally being lost when there is high contention; the probability of this happening is low enough that it was not observed in any of the the much more controlled tests I was performing during the development of the system, but produces clearly incorrect behaviour in more substantial programs. This issue will obviously have to be resolved before the system is usable, and renders any detailed performance analysis fairly meaningless, which is why this report does not contain any experimental performance evaluation.

In addition, there are several features I have prepared for while designing and implementing the core, but which are not fully functional yet. In the following subsections I briefly discuss the most significant.

6.1 Access Visibility

The system I have described in detail always uses invisible reads and visible writes, although I intended the system to support both variants of read and write access.

Invisible reads are a key performance optimisation of more recent STM systems. The "invisible" concept describes the fact that no other transaction can detect the read. Visible reads, on the other hand, are observable by other transactions. This requires the presence of meta-data that is

read by all transactions, and updated for all accesses, causing significant synchronisation overhead (even if it is merely that contained implicitly in the cache coherency protocols of multiprocessor systems). There is an interesting trade-off, in that a transaction does not need to perform extra work validating its visible reads; before committing a write to a variable that has been visibly read, a writing transaction *must* first abort any visible readers. Therefore a transaction can validate its visible reads simply by checking that the status field of its transaction descriptor is not `ABORTED`. However, even this performance advantage is usually outweighed by the increased meta-data contention.

Visible reads are necessary, however, for providing any sort of guarantee of progress or fairness. When reads are invisible, writing transactions that attempt to commit are unable to detect the transactions their writes will force to abort, and so cannot observe protocols concerning when one transaction can abort another. The conflict will only be detected later, when no decision can be made about which of the two transactions should be aborted, as one of them has already committed.

The concept of write visibility is not normally discussed as a parallel of read visibility. Normally visible writes are implied by eager acquire with undo-logging, and invisible writes are implied by any form of redo-logging. The data layout I have used in my system separates these concepts somewhat, allowing the possibility of writes being either visible or invisible, with the similar distinction that transactions are responsible for checking the validity of their invisible writes, but can assume they will be aborted before their visible writes are invalidated.

Writes are always visible in the current version of my system, as provided by the ownership mechanism. There is potentially less reason to have invisible writes than invisible reads; successfully committing writes fundamentally involves an update to data read by all transactions, so much less is saved by keeping the writes private during execution. Further, the only way to achieve invisible writes would be to buffer writes into a private log, requiring this log to be searched before every read as discussed in section 4.2. If the write log were a simple linear structure, this would add $O(n)$ overhead to each of n reads. A more complex data structure could significantly improve efficiency, at the cost of implementation complexity, and would still impose at least some constant overhead. Since using visible writes mandates the use of mixed-invalidation, it is possible that multiple long-running transactions with many writes in common will live-lock without good contention management. This prospect would be much less likely with invisible writes and true lazy acquire, since in that case a transaction would only cause other transactions to abort by committing. Livelock would still be possible, as it is possible that two transactions in their commit phase will both abort each other due to acquiring variables before committing, but this is exceedingly unlikely with lazy acquire.

There is no reason why read or write visibility must be an all-or-nothing decision. I have designed my system to support transactions that may perform any given access either visibly or invisibly, and allow the contention manager to decide at runtime how to perform each access. This would allow the use of contention management policies such as the one described in [23], which runs transactions using invisible reads by default for speed, but begins using visible reads for transactions that repeatedly fail to commit, achieving low latency in the common case but eventually providing some assurance of likely fairness. I believe a similar strategy could be combined with the Greedy contention manager[4], which provides a provable upper bound on the time taken by the system (compared to an optimal off-line list scheduler) at the cost of using entirely visible reads, to provide a system with a provable upper bound and good performance in the absence of significant contention. The idea would be to ensure that transactions spend only a bounded time in the "invisible" mode in which any priority they have cannot be guaranteed to be observed, thus allowing a provable bound on the total execution time of the combination of this period and the visible mode execution.

6.2 Eager Acquire

I have mentioned earlier that my system makes it fairly trivial to use either eager acquire or lazy acquire (really mixed invalidation until invisible writes are implemented). I have not yet

```
contention_manager(OtherTrans, TVar, !STM)

cm_init_transaction(!STM)
cm_start_transaction(!STM)
cm_commit_transaction(!STM)

init_cm_status(CM)

cm_choose_read_mode(TVar, Visible, !STM)
cm_choose_write_mode(TVar, Visible, !STM)
```

Figure 4: Contention management interface

implemented a mechanism for making this determination, however, short of manually changing the source code in a few places. At the very least, the source code should be changed so that there is a single point of control for compiling this decision into the system, which would be a fairly simple matter of introducing a small number of conditionally defined macros in the C runtime.

6.3 Contention Management

The contention management interface in my system is not fully implemented, and so I have not implemented any contention management strategies. Figure 4 shows the interface a contention manager must implement, while figure 5 shows the interface that is provided by the STM system to contention managers. Apart from this, the contention manager is completely independent of the rest of the STM system.

When a transaction discovers that it cannot access a transaction variable because of another transaction, it calls `contention_manager`. `OtherTrans` is a *transaction reference*; this is a type that is opaque at the Mercury level, but in the C runtime is a pointer to a transaction descriptor; exactly the same C type as the STM state itself. The difference is that the unique STM state conceptually identifies the state of the transactional system for the transaction being executed, while a transaction reference is non-unique, and serves to identify another transaction. Different operations are permitted on the two; a running transaction keeps private state (such as the log segments) in its own descriptor, which is updated by predicates such as `read_stm_var`. Transaction references are represented as a different Mercury type, so that such predicates cannot be used to modify the private state of other transactions. The only operations permitted on a transaction reference deal with the *public* part of a transaction's state: its transaction status, and its *contention management status* (see below).

An implementation of `contention_manager` may cause the calling transaction to abort, which throws an exception. If it does not throw an exception, the calling transaction immediately repeats

```
get_cm_status(CM, !STM)
update_cm_status(CM_Old, CM_New, Success, !STM)

get_cm_status_of_other(OtherTrans, CM, !STM)
update_cm_status_of_other(OtherTrans, CM_Old, CM_New,
    Success, !STM)

cm_abort_self(STM)
cm_abort_other(OtherTrans, !STM)
```

Figure 5: Interface provided to contention managers

whatever action it had failed at before calling `contention_manager`, assuming that some corrective action has been taken (such as aborting the conflicting transaction or executing a wait). Thus the strategy for resolving conflicts is completely encapsulated inside this predicate.

To give the contention manager slightly more information, cm_init_transaction is called when a transaction first begins, `cm_start_transaction` is called every time a transaction begins forward execution (once just after `cm_init_transaction`, and also just before execution resumes after every abort), and `cm_finalise_transaction` is called after a transaction commits successfully. Note that only top-level transactions are relevant here; the contention manager is unaware of the presence of any nested transactions.

In `cm_choose_*_mode`, `Visible` is a boolean output argument, providing the contention manager with control of individual access visibility. For current contention management algorithms that I am aware of, the `TVar` parameter is not necessary, but is included to support possible contention management strategies that identify highly contended variables and treat them differently.

The predicates `get_cm_status` and `update_cm_status` allow the contention manager to maintain some per-transaction state, called its contention management status. An implementation of a contention manager must provide a type for this status as well as the predicates in figure 4. The initial value of the contention management status should be provided by `cm_init_transaction`. Likewise, `get`/`update_cm_status_of_other` allows access to the contention management status of another transaction. Compare-and-swap update predicates are provided rather than simple set predicates, since the contention management status is accessed concurrently by multiple threads. A transaction's initial contention management status must be provided by the implementation of init_cm_status.

The contention manager also has access to `cm_abort_self` and `cm_abort_other`, if it determines that a conflict should be resolved by aborting a transaction.

This interface should be sufficient to implement many of the contention management strategies that have been proposed, such as Polka[13], Greedy[4], and Patient[23].

6.4 Transaction Execution and Roll-back by Continuation Passing

As detailed in section 5, the mechanism for implementing roll-back of the execution state is by exception handling. Every location to which execution may need to roll-back requires a predicate which sets up an exception handler and then calls the code that will possibly roll-back. When an exception is caught, the predicate always recursively calls itself, so that the exception handler will be set up again in case the second attempt also aborts. It would be a lot simpler if the abort-and-roll-back process could be implemented by simply calling the predicate at which execution needs to resume, bypassing all this extra machinery. This can be achieved if transactional code is written in *continuation passing style*.

Continuation passing style is most often seen in the context of functional languages. A function written in this style takes an extra argument, representing the continuation of the call to the function. Instead of returning a value normally, such a function calls the continuation, passing it the "return value"; alternatively it may call some other continuation passing function, passing a continuation to the called function also. If some function eventually returns a value directly, it will pass back to the location where the first continuation passing function was called.

This idea is fairly easy to apply to transactional code in Mercury. At the lowest level, the only places from which an abort can occur are the STM primitives `read_stm_var`, `write_stm_var`, and the implicit call to `commit` at the end of a transaction. Algorithm 8 compares simple pseudo-code for standard and continuation-based versions of these `read_stm_var`. Instead of returning the read value in an output value or raising an exception if the transaction needs to abort, `read_stm_var_cont` either passes the read value to the "normal continuation" passed in as an argument, or calls an "abort continuation". Corresponding pseudo-code for the continuation based versions of `write_stm_var` and `commit` is shown in appendix **??**, and is quite similar. Since `write_stm_var` is a consumer of data rather than a producer its normal continuation does not take a value argument. The `Output` argument (which is threaded through these calls regardless of

Algorithm 8 Reading transaction variables with continuations

```
read_stm_var(TVar, Value, !STM) :-
    try_read_stm_var(TVar, Result, !STM),
    (
        Result = ok(Value)
    ;
        Result = abort(Checkpoint),
        throw(stm_exc_abort(Checkpoint))
    ).

⟹

read_stm_var_cont(TVar, NormCont, Output, !STM) :-
    try_read_stm_var(TVar, Result, Output, !STM),
    (
        Result = ok(Value),
        NormCont(Value, Output, !STM)
    ;
        Result = abort(AbortCont),
        AbortCont(Output, !STM)
    )
```

which continuation is called) represents the output of the *entire transaction*. The normal continuation passed to `commit_cont` (which is *not* given access to the STM state) is expected to finally bind `Output` to the necessary value once the transaction has committed successfully.

Without further support from the compiler, this scheme would require programmers to write code in continuation passing style, at least partially. Calls can be made to ordinary non-transactional predicates (predicates that do not take a pair of STM state arguments, basically) in the direct style, but any call to the primitive STM operations will either call the continuation it receives, or call into some earlier point of the transaction execution. Such calls must therefore be made as tail-calls (the "rest" of a predicate calling a primitive STM operation must be split off into a separate predicate anyway, in order to give a name to the normal continuation so it can be passed to the primitive). In fact, since any predicate with access to the STM state may call one of these primitives, all such predicates must be called as tail-calls, and must be written in the continuation passing style. The Mercury compiler is able to optimise these tail-calls to avoid consuming $O(nk)$ stack space for a transaction with maximum call-depth n after k aborts.

The abort continuations are the obvious counterpart to checkpoints thrown as exceptions. They would be stored in the log segments, in much the same way as the address of a log segment is taken as the identifier of a checkpoint in my current implementation; when a nested transaction is entered, creating a new log segment, it could store a reference to the closure that was given as the body of the transaction. But there are much more interesting possibilities for such a system. The checkpoint system based on exception handling requires that the scope of the exception handler corresponding to a checkpoint is still live in order to abort to a checkpoint, which means that checkpoints can only be created at particular points in the system. With the continuation-based system, an abort is simply a tail-call to an appropriate closure. Any time a primitive STM operation is called, it is passed a normal continuation, *which is an appropriate target for a later abort*. Creating a checkpoint thus requires only splitting off a new log segment and storing a reference to the continuation before it is called.

While creating a checkpoint for every primitive operation would likely be horrendously inefficient, due to the allocation of many tiny log segments, possibilities are opened up for checkpoints to be automatically created at sensible places, without requiring the programmer to create nested transactional structure for this purpose. This could possibly be done by the compiler based on

code analysis, or dynamically by the runtime system; a simple policy such as "create a checkpoint for every N transaction variable accesses" would impose only a little overhead for reasonable N, but has potential to dramatically reduce the costs of aborting transactions — every checkpoint created can remain useful all the way up to and including the attempt to commit.

The disadvantage of such a system, of course, is requiring programmers to program in continuation-passing style, which is often difficult and error-prone. The only practical option is to allow programmers to program in direct style, and transform their code into continuation-passing style.

Transformations from direct style to continuation-passing style are well known. However, transforming all of the code called from within an atomic scope would do too much; only "STM actions" (predicates that take a pair of STM state arguments) need to be in continuation-passing style. The transformation to continuation-passing style creates many small higher-order predicates, which is likely to be a performance hit. Rompf et al[19] describe a selective transformation driven by type annotations, which transforms only those parts of the program that actually need the functionality of continuations. The techniques described seem likely to be of use in implementing a transformation of transactional code into continuation-passing style.

There are some potential muddy patches for this approach, however. Mercury compiles modules (mostly) independently. When a transaction calls an STM action predicate defined in another module, the source code of that predicate may not be available to transform. It may perhaps to be reasonable to enforce that predicates taking a pair of STM state arguments are always transformed into this style when they are compiled, so we can assume that such predicates in other modules will be transformed at the time when that module is (or was, or will be) compiled. Polymorphic higher-order predicates can still pose a problem; an STM state pair is a quite natural fit for the accumulator arguments to higher-order predicates such as `foldl`, when processing a list of transaction variables, for example. When compiling the `list` module, the compiler has no way of knowing that `foldl` will be called in this way. It seems inevitable that the exception-handling approach to executing transactions would have to remain, even if a transformation into continuation-passing style were the preferred implementation. An exception-based STM action could be called from within continuation-based code by wrapping it in an exception handler, and using the result of `try` to determine which continuation to call when it returns.

Since the checkpoints enabled by the continuation-passing style are so much more flexible than the exception-based ones, some complications must be resolved that simply do not arise in the implementation I described in section 5. For example, when an `or_else` alternative has retried in the original scheme, its writes must be discarded, and will never be needed again; if the transaction aborts to that alternative, it will start at the beginning of the alternative, before any writes were made by it. With the continuation-based checkpoints, it is possible for a transaction to restart part-way through an `or_else` alternative, even after it has retried. Either this possibility must be carefully prevented, or when writes are "discarded" on retry they must still be available for the rollback process to attempt to reinstate them.

The possible combination of exception-based checkpoints and continuation-based checkpoints also complicates matters. An abort continuation must not be called to attempt to roll back "past" a checkpoint exception handler. The STM system would have to throw an abort to be caught by the inner-most exception handler that also encloses the desired abort destination, and from that position the abort continuation could be called directly.

Despite the issues that would have to be overcome, I believe this is a promising approach for further investigation.

6.5 Application-specific Settings

There are a number of parameters in this system (such as the contention management policy, acquire strategy, access visibility, etc), even if the current system only implements one option for most of them. Available research into STM systems leads me to expect that the "optimal" setting for these parameters will be application dependent, or even dependent on a particular workload for an application. However, changing these parameters in the current version of my

system requires modifying and recompiling the Mercury standard library module implementing STM, and (in some cases) the entire Mercury system including the C runtime. This is not a very practical arrangement. Ideally the user should be able to choose these parameters with either command line arguments to the Mercury compiler or declarations in the program (the atomic scope syntax can be very naturally extended to support setting per-transaction options, by adding extra parameters besides the `inner` and `outer` state variables).

One way to provide much more flexibility for these decisions would be to make them dependent on the values of environment variables, but that precludes the compiler optimising away the decision logic when the value needs to remain constant (the Greedy contention manager, for example, requires all transactions to always use visible accesses, which would allow `cm_choose_read_mode` and `cm_choose_write_mode` to be optimised away by the compiler if inter-module optimisation is enabled). An appropriate way to address this issue needs to be investigated.

7 Conclusion

The system I have implemented provides the framework for a practical software transactional memory system. This system implements STM in a non-blocking manner, largely inspired by descriptions of RSTM [14, 23], and uses timestamp-based validation[18] to minimise the overhead of providing opacity[5]. The system allows the possibility of a contention manager providing stronger progress guarantees. It uses mixed-invalidation, which is similar to lazy acquire for read-write conflicts, but similar to eager acquire for write-write conflicts, to avoid the need to search the write set on every read. This optimisation has probably been implemented before, as [24] defines mixed-invalidation and compares a system that implements mixed-invalidation to one that only approximates it, but they make no mention of the effect on write set lookups.

My implementation also makes use of a checkpoint system to mitigate the cost of aborting by rolling back transactions only partially. I am unaware of any previous research that discusses such a system.

Interested readers my find the source code of my implementation under `/home/mercury/bmellor/honours/mercury`, on the departmental servers. This directory contains a full CVS checkout of the Mercury system, with my changes applied. My implementation is not at the time of writing stable enough to be incorporated into the main Mercury codebase, but I intend to do the necessary cleanup work and submit it to the Mercury team in the near future.

With a moderate amount of further work, as outlined in the previous section, and some general optimisation of the system, I believe this system could be a practical alternative to lock-based synchronisation for many applications. There are, of course, a number of avenues for further research.

Given that the ideal characteristics of an STM system are workload dependent, adaptively altering those characteristics at runtime is a promising idea. I have tried to design my system so that little modification would be required to support this (for example, the contention management interface potentially supports dynamic adaptation of access visibility and conflict resolution policies), but suitable heuristics need to be investigated.

Contention management policies are an obvious area for future work. Besides implementing policies proposed by existing research, there is scope for new policies to be developed. In particular, most research into contention management has been done in the context of imperative languages such as C/C++ or Java. Mercury programs have significantly different characteristics to those assumed by this research, and there may well be different possibilities and requirements for good contention management in a Mercury STM system. This needs exploration.

The checkpoint system presents opportunities for further research. As implemented, they do not seem to be terribly helpful, but with further extensions (such as those possible if the execution of transactions were based on continuations) there are many possibilities here. Heuristics for the automatic creation of checkpoints (at runtime or compile-time) could be investigated. Another possibility may be checkpoint-aware contention managers — can some approximation of "abort

the transaction that will lose the least work" be efficiently implemented?

I suspect some form of profiling feedback could be very useful. If profiling could identify which transactions are likely to abort, which ones are likely to be read-only, which variables cause the most conflicts, etc, then there is the possibility of fine-tuning various policies based on this data. One example might be to attempt to insert checkpoints after strings of likely-uncontended accesses before accessing a variable that is often contended. Mercury contains a deep profiler[2], which could potentially be used to collect some of this information, but at present it does not support programs that make use of exceptions, and so will not work for STM programs. It is likely though that collecting data to be used to estimate the answers to questions such as "which transactions are likely to abort?" would benefit from having the code compiled in a closer form to the release version than a version compiled for general profiling, to minimise the effect of different timing characteristics on abort rates. Exactly what profiling data would be useful and what could be done with it would be an interesting area for further research.

Further research into the transactional model itself will also play an important role in determining how widely it is adopted. The composability of transactions is often claimed as their greatest advantage over lock-based synchronisation, but this advantage is often overstated. While transactions *themselves* can be easily composed, logical units of code implemented using transactions often cannot be composed. This is due to the way the transactional model forces code to be refactored so that irreversible actions such as I/O are performed outside transactions; the natural way to deal with this is to write procedures that encapsulate a transaction accessing shared data together with related I/O actions. But now this combined procedure cannot be called from within a transaction, and there is no way to combine two such procedures into an atomic procedure with the effect of both — exactly the same problem that was cited for locks in section 2.2! One partial way of addressing this problem is to allow transactions to request *inevitability*, a guarantee by the system that the transaction *cannot* abort, which is only granted to one transaction at a time. Such a transaction can safely perform irreversible operations, including I/O. A more radical idea is the TIC (Transactions with Isolation and Cooperation) model[22], which allows transactions to break the normal transactional model in ways that are not safe in general, but in a disciplined manner that directs the programmer's attention to the places that need special care, and prevents such a loss of safety from being introduced accidentally (this is similar in spirit to Mercury's treatment of impurity). I believe that some such extensions will be necessary for software transactional memory to achieve its promises, but it is not yet known which extensions are useful.

Beyond the STM system itself, there is the matter of data structures. STM implementations that work on the basis of memory words transparently work on existing data structures. Systems (such as my implementation) that use explicit transaction variables, while being more flexible in terms of allowing the programmer to determine the granularity of sharing, require new data structures to be developed making this granularity of sharing explicit in the form of where transaction variables appear. As an example, suppose a list of items needs to be shared. Should the data type used be `stm_var(list(T))`, `list(stm_var(T))`, or `stm_var(list(stm_var(T)))`? Manipulating such lists may be awkward, and involve the re-implementation of many standard list operations to work on lists of transaction variables instead. More complicated data structures have even more possibilities for corresponding transactional versions. How much of the process of re-implementing them for use with transactions can be automated? Developments in this area could greatly enhance the usability of software transactional memory.

8 Acknowledgements

I would like to thank my supervisor, Zoltan Somogyi, for all his support, even when I didn't make effective use of it. Thanks also to Leon Mika for breaking the ground before my work.

A Continuation-based Write and Commit

In section 6.4, I showed a comparison of pseudo-code for the exception-based and continuation-based versions of `read_stm_var`. Since they are so similar, I did not explicitly show the corresponding pseudo-code for `write_stm_var` and `commit` in the main body of my report. I do so here, for interest's sake, since there are small but significant differences that may be more fully appreciated directly.

Algorithm 9 Writing transaction variables with continuations

```
write_stm_var(TVar, Value, !STM) :-
    try_write_stm_var(TVar, Result, !STM),
    (
        Result = ok
    ;
        Result = abort(Checkpoint),
        throw(stm_exc_abort(Checkpoint))
    ).

⟹

write_stm_var_cont(TVar, Value, NormCont, Output, !STM) :-
    try_write_stm_var(TVar, Result, Output, !STM),
    (
        Result = ok,
        NormCont(Output, !STM)
    ;
        Result = abort(AbortCont),
        AbortCont(Output, !STM)
    )
```

Algorithm 10 Committing with continuations

```
commit(!STM) :-
    try_commit(Result, !STM),
    (
        Result = ok
    ;
        Result = abort(Checkpoint),
        throw(stm_exc_abort(Checkpoint))
    ).

⟹

commit_cont(ReturnCont, Output, !STM) :-
    try_commit(Result, !STM),
    (
        Result = ok,
        ReturnCont(Output)
    ;
        Result = abort(AbortCont),
        AbortCont(Output, !STM)
    )
```

B Code Comparison of the Prototype and the New STM System

To demonstrate the difference between my system and the prototype implementation, as well as to permit a glimpse of the full complexities I have glossed over in the descriptions in my report proper, I present here in full the "top layer" code needed to implement a simple example transaction in both systems. Here "top layer" refers to the wrapper code that the rest of the program calls, and which calls the code for the body of the transaction. The "bottom layer" would be the internal implementation of `read_stm_var`, `validate`, etc, which I do not attempt to present here. Predicates that are undefined in both sets of code are implemented by C code with access to the STM support code in the runtime.

The code shown for my system can be found on the departmental servers, in the file `/home/mercury/bmellor/honours/mercury/library/stm.m`, which contains the STM library module. The files containing the STM support code in the runtime are `mercury_stm.h`, `mercury_stm_types.h`, and `mercury_stm.c`, all found in `/home/mercury/bmellor/honours/mercury/runtime/`.

The predicate used for this example is the `transfer` predicate from the bank account example, implemented in terms of two assumed predicates `withdraw` and `deposit`:

```
:- pred transfer(int::in, stm_var(account)::in, stm_var(account)::in,
    io::di, io::uo) is cc_multi.

transfer(Amount, AccountA, AccountB, !IO) :-
    atomic [outer(!IO), inner(!STM)] (
        withdraw(Amount, AccountA, !STM),
        deposit(Amount, AccountB, !STM)
    ).
```

From this, the prototype system produces the following code, all generated by the compiler. I have taken this example directly from the appendix of Leon Mika's honours report[15].

```
:- pred transfer(int::in, stm_var(account)::in, stm_var(account)::in,
    io::di, io::uo) is cc_multi.

transfer(Amount, AccountA, AccountB, IO0, IO) :-
    'StmExpanded_toplevel_transfer_5_1_2'(Amount, AccountA, AccountB,
        IO0, IO).


:- pred 'StmExpanded_toplevel_transfer_5_1_2'(stm_var(account)::in,
    stm_var(account)::in, int::in, io::di, io::uo) is cc_multi.

'StmExpanded_toplevel_transfer_5_1_2'(AccountA, AccountB, Amount,
        IO0, IO) :-
    'StmExpanded_rollback_transfer_3_1_0'(AccountA, AccountB, Amount),
        IO = IO0.
```

```
:- pred 'StmExpanded_rollback_transfer_3_1_0'(int::in,
    stm_var(account)::in, stm_var(account)::in) is cc_multi.

'StmExpanded_rollback_transfer_3_1_0'(Amount, AccountA, AccountB) :-
    promise_pure (
        impure stm_create_transaction_log(STM0_Aux_1),
        Closure_Aux_3 = 'StmExpanded_wrapper_transfer_6_1_1'(Amount,
            AccountA, AccountB),
        unsafe_try_stm(Closure_Aux_3, ExceptionResult_Aux_4,
            STM0_Aux_1, STM_Aux_2),
        (
            ExceptionResult_Aux_4 = exception(ExceptUnivVar_Aux_6),
            (
                RollbackExcpt_Aux_12 = rollback_invalid_transaction,
                type_to_univ(UnivPayload_Aux_11, ExceptUnivVar_Aux_6),
                UnivPayload_Aux_11 = RollbackExcpt_Aux_12
            ->
                impure stm_discard_transaction_log(STM_Aux_2),
                'StmExpanded_rollback_transfer_3_1_0'(Amount, AccountA,
                    AccountB)
            ;
                (
                    RollbackExcpt_Aux_10 = rollback_retry,
                    type_to_univ(UnivPayload_Aux_9, ExceptUnivVar_Aux_6),
                    UnivPayload_Aux_9 = RollbackExcpt_Aux_10
                ->
                    impure stm_lock,
                    impure stm_validate(STM_Aux_2, ValidResult_Aux_8),
                    (
                        ValidResult_Aux_8 = stm_transaction_valid,
                        impure stm_block(STM_Aux_2)
                    ;
                        ValidResult_Aux_8 = stm_transaction_invalid,
                        impure stm_unlock
                    ),
                    impure stm_discard_transaction_log(STM_Aux_2),
                    'StmExpanded_rollback_transfer_3_1_0'(Amount,
                        AccountA, AccountB)
                ;
                    impure stm_lock,
                    impure stm_validate(STM_Aux_2, ValidResult_Aux_7),
                    impure stm_unlock,
                    (
                        ValidResult_Aux_7 = stm_transaction_valid,
                        rethrow(ExceptionResult_Aux_4)
                    ;
                        ValidResult_Aux_8 = stm_transaction_invalid,
                        'StmExpanded_rollback_transfer_3_1_0'(Amount,
                            AccountA, AccountB)
                    )
                )
            )
        ;
                ExceptionResult_Aux_4 = succeeded(DummyResult_Aux_5)
        )
    ).
```

```
:- pred 'StmExpanded_wrapper_transfer_6_1_1'(int::in,
    stm_var(account)::in, stm_var(account)::in, stm_dummy_output::out,
    stm::di, stm::uo) is cc_multi.

'StmExpanded_wrapper_transfer_6_1_1'(Amount, AccountA, AccountB,
        Stm_ResultVar, STM0, STM) :-
    withdraw(Amount, AccountA, STM0, V_17),
    deposit(Amount, AccountB, V_17, STM),
    Stm_ResultVar = stm_dummy_output,
    promise_pure (
        impure stm_lock,
        impure stm_validate(STM, Stm_Expand_IsValid_Aux_0),
        (
            Stm_Expand_IsValid_Aux_0 = stm_transaction_valid,
            impure stm_commit(STM),
            impure stm_unlock
        ;
            Stm_Expand_IsValid_Aux_0 = stm_transaction_invalid,
            impure stm_unlock,
            Stm_Expand_Rollback_Aux_1 = rollback_invalid_transaction,
            throw(Stm_Expand_Rollback_Aux_1)
        )
    ).
```

In my implementation, the code corresponding to the `transfer` predicate is as follows. This code must be written by hand at the moment, but it can be seen that it would be quite easy to mechanise this so that the compiler implements atomic scopes in terms of the new system instead of the prototype.

```
:- pred transfer(int::in, stm_var(account)::in, stm_var(account)::in,
    io::di, io::uo) is det.

transfer(Amount, AccountA, AccountB, !IO) :-
    promise_pure (
        impure impure_atomic_transaction(transfer_body(Amount, AccountA,
            AccountB), _, !IO)
    ).


:- impure pred transfer_body(int::in, stm_var(account)::in,
        stm_var(account)::in, {}::out) is cc_multi.

transfer_body(Amount, AccountA, AccountB, {}) :-
    impure get_stm_state(STM0),
    withdraw(Amount, AccountA, STM0, STM1),
    deposit(Amount, AccountB, STM0, STM1),
    commit(STM1, STM),
    impure set_stm_state(STM).
```

As can be seen, only a small amount of code needs to be generated for each atomic scope in the new system. The rest of the "scheduling" logic is in higher order library predicates, shown below.

```
:- impure pred impure_atomic_transaction(impure pred(T), T, io, io).
:- mode impure_atomic_transaction(in(pred(out) is cc_multi), out, di, uo)
    is cc_multi.

impure_atomic_transaction(Closure, Output, !IO) :-
    some [!STM] (
        impure get_stm_state(!:STM),
        init_transaction(!STM),
        impure top_level_transaction(Closure, Output),
        end_transaction(!STM),
        impure finalise_stm_state(!.STM)
    ).


:- impure pred top_level_transaction(impure pred(T), T).
:- mode top_level_transaction(in(pred(out) is cc_multi), out) is cc_multi.

top_level_transaction(Closure, Output) :-
    impure get_stm_state(STM0),
    start_transaction(STM0, STM1),
    impure set_stm_state(STM1),
    ( try []
        impure checkpoint_goal(Closure, Output)
    then
        true
    catch stm_exc_retry ->
        impure rollback_to_top,
        impure top_level_transaction(Closure, Output)
    catch stm_exc_abort(_) ->
        impure rollback_to_top,
        impure top_level_transaction(Closure, Output)
    ).


:- impure pred checkpoint_goal(impure pred(T), T).
:- mode checkpoint_goal(in(pred(out) is cc_multi), out) is cc_multi.

checkpoint_goal(Closure, Output) :-
    some [!STM] (
        impure get_stm_state(!:STM),
        stm_create_checkpoint(Checkpoint, !STM),
        ( try []
            impure Closure(Output)
        then
            stm_mark_checkpoint_inactive(Checkpoint, !STM),
            impure set_stm_state(!.STM)
        catch stm_exc_abort(Checkpoint) ->
            rollback_to_checkpoint(Checkpoint, !STM),
            impure set_stm_state(!.STM),
            impure checkpoint_goal(Closure, Output)
        )
    ).
```

References

[1] The Mercury Language Reference Manual. Technical report, Department of Computer Science and Software Engineering, University of Melbourne, Melbourne, Australia, 2009.

[2] Thomas C. Conway and Zoltan Somogyi. Deep profiling: engineering a profiler for a declarative programming language. Technical report, Department of Computer Science and Software Engineering, University of Melbourne, Melbourne, Australia, 2001.

[3] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *Proceedings of the 20th International Symposium on Distributed Computing*, 2006.

[4] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Toward a theory of transactional contention managers. In *Proceedings of the 24th Symposium on Principles of Distributed Computing*, 2005.

[5] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th Symposium on Principles and Practice of Parallel Programming*, 2008.

[6] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. Technical report, Microsoft Research, 2006.

[7] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. In *Proceedings of the 2006 Conference on Programming Language Design and Implementation*, pages 14–25, New York, NY, 2006.

[8] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13:124–149, 1993.

[9] Maurice Herlihy, J. Eliot, and B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 289–300, 1993.

[10] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, pages 522–529, 2003.

[11] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd annual symposium on Principles of distributed computing*, pages 92–101, New York, NY, 2003. ACM.

[12] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12:463–492, 1990.

[13] William N. Scherer III and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the 24th annual ACM symposium on Principles of Distributed Computing*, pages 240–248, New York, NY, 2005. ACM.

[14] Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer III, and Michael L. Scott. Lowering the overhead of nonblocking software transactional memory. Technical report, Department of Computer Science, University of Rochester, 2006.

[15] Leon Mika. Software transactional memory in Mercury. Technical report, Department of Computer Science and Software Engineering, The University of Melbourne, 2007.

[16] Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, 1979.

[17] Torvald Riegel, Pascal Felber, and Christof Fetzer. A lazy snapshot algorithm with eager validation. In *Proceedings of the 20th International Symposium on Distributed Computing)*, pages 284–298, 2006.

[18] Torvald Riegel, Christof Fetzer, and Pascal Felber. Time-based transactional memory with scalable time bases. In *Proceedings of the 19th Symposium on Parallel Algorithms and Architectures*, pages 221–228, New York, NY, 2007.

[19] Tiark Rompf, Ingo Maier, and Martin Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform. In *Proceedings of the 14th International Conference on Functional Programming*, pages 317–328, New York, NY, 2009.

[20] Bratin Saha, Ali reza Adl-tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *Proceedings of the 11th Symposium on Principles and Practice of Parallel Programming*, pages 187–197, 2006.

[21] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the 14th Symposium on Principles of Distributed Computing*, pages 204–213, New York, NY, 1995.

[22] Yannis Smaragdakis, Anthony Kay, Reimer Behrends, and Michal Young. Transactions with isolation and cooperation. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 191–210, New York, NY, 2007. ACM.

[23] Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott. A comprehensive strategy for contention management in software transactional memory. In *Proceedings of the 14th Symposium on Principles and Practice of Parallel Programming*, pages 141–150, New York, NY, 2009.

[24] Michael F. Spear, Virendra J. Marathe, William N. Scherer III, and Michael L. Scott. Conflict detection and validation strategies for software transactional memory. In *Proceedings of the 20th International Symposium on Distributed Computing*, 2006.

[25] Peter Wang. Parallel Mercury. Technical report, Department of Computer Science and Software Engineering, The University of Melboure, 2006.

[26] Rui Zhang, Zoran Budimlić, and William N. Scherer III. Commit phase in timestamp-based STM. In *Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures*, pages 326–335, New York, NY, 2008.