Chapter 1

`unreadTVar`: Extending Haskell Software Transactional Memory for Performance

Nehir Sonmez[1 2], Cristian Perfumo[1 2], Srdjan Stipic[1 2], Adrian Cristal[1], Osman S. Unsal[1], Mateo Valero[1 2]

Abstract: As new trends in computer architecture lead towards shared-memory chip multiprocessors (CMP), the rules for programming these machines are significantly changing. In the search for alternatives to deadlock-prone lock-based concurrency protocols, Software Transactional Memory (STM) extensions to Haskell have provided an easy-to-use lock-free abstraction mechanism for concurrent programming, using atomically composed blocks operating on transactional variables. However, as in the case for linked structures, the composition of these atomic blocks require extra attention, as the transactional management might act overconservatively by keeping track of more variables than necessary, causing an overall decrease in performance. To remedy this situation, we have extended the Transactional Memory module of the Glasgow Haskell Compiler (GHC) 6.6 to support a construct that allows the removal of a transactional variable from the readset. Although this construct that we term `unreadTVar`, when not applied carefully, might put the strong atomicity guarantees of STM at risk, the experimentations done with linked lists and binary trees show that it can significantly improve execution time and memory usage when traversing transactional linked structures.

[1]Barcelona Supercomputing Center, Barcelona/Spain;
Email:{nehir.sonmez,cristian.perfumo,srdjan.stipic,adrian.cristal,osman.unsal,mateo.valero}@bsc.es
[2]Computer Architecture Department – Universitat Politècnica de Catalunya

1.1 INTRODUCTION

Chip multiprocessors have arrived and are dominating the microprocessor market, demanding efficient use of parallelism and easier methods in programming shared-memory parallel architectures. In this era, traditional mechanisms such as lock-based thread synchronization, which are tricky to use and non-composable, are becoming less likely to survive. Meanwhile, the use of imperative languages that cause uncontrolled side effects is becoming questionable. Consequently, while strongly-typed functional languages are attracting more attention than ever, lock-free Transactional Memory (TM) is a serious candidate to the future of concurrent programming. As Harris et al. state in their work [HMPJH05], STM can be expressed elegantly in a declarative language, and moreover, Haskell's type system (particularly the monadic mechanism) avoids threads to violate the restrictions that a transactional scheme demands regarding the access to shared variables. This situation is more likely to happen under other programming paradigms, for example, as a result of access to memory locations through the use of pointers.

The Transactional Memory approach, which is inherited from database theory, allows programmers to specify transaction sequences that are executed atomically, ensuring that all operations within the block either complete as a whole, or automatically rollback as if they were never run. Atomic blocks simplify writing concurrent programs because when a block of code is marked atomic, the compiler and the runtime system ensure that operations within the block appear atomic to the rest of the system [HPJ06]. Transactional management of the memory can be implemented either in hardware (HTM) [HM93], or software (STM) [ST95]. As always, there is an intermediate point that incorporates both approaches, called Hybrid TM (HyTM) [DFL$^+$06, KCJ$^+$06]. TM schemes attempt to optimistically interleave and execute all transactions in parallel. A transaction is committed if and only if other transactions have not modified the section of the memory which its execution depended on. As a consequence, the programmer no longer needs to worry about deadlocks, manual locking, low-level race conditions or priority inversion [HPST06].

In this paper, the following contributions are made:

- The problems related to synthetic but abundantly used linked structures which suffer from serious performance and memory problems when realized transactionally are investigated.

- By extending the Haskell STM with the `unreadTVar` operation, which removes transactional variables from the readset, a more efficient way of constructing the atomic blocks that operate on linked structures was provided. Although the idea has been previously implemented in imperative languages such as C, C++ and Java [SK06, HLMS03], this is the first implementation of such a feature for a functional language, which also supports composability.

- Specifically, linked lists and binary trees were focused on in order to present absolute speedup results that show substantial increase in system performance

using the proposed approach. Moreover, safety issues and potential downsides are discussed in detail.

The rest of this paper is organized as follows: In Section 1.2, the STM library of Concurrent Haskell is summarized, after which, in Section 1.3, an example case of linked structures, namely the singly-linked list, is presented with an explanation of the problem of false conflict and the proposed solution. Section 1.4 introduces the new function: `unreadTVar`, demonstrates its implementation and its usage. Section 1.5, discusses performance results on linked lists and binary trees versus the potential safety issues and trade-offs. Conclusions are derived on Section 1.6.

1.2 BACKGROUND: STM IN CONCURRENT HASKELL

The Glasgow Haskell Compiler 6.6 [htt] provides a compilation and runtime system for Haskell 98 [III04], a pure, lazy, functional programming language. The GHC natively contains STM functions built into the Concurrent Haskell library [JGF96], providing abstractions for communicating between explicitly-forked threads.

1.2.1 The Monadic World

STM provides a safe way of accessing shared variables between concurrently running threads through the use of monads [DHM+06], having only I/O actions in the `IO` monad and STM actions in the `STM` monad. Programming using distinct STM and I/O actions ensures that only STM actions and pure computation can be performed within a memory transaction (which makes it possible to re-execute transactions), whereas only I/O actions and pure computations, and not STM actions can be performed outside a transaction. This guarantees that `TVar`s cannot be modified without the protection of `atomically`, and thus separates the computations that have side-effects from the ones that are effect-free. Utilizing a purely-declarative language for TM also provides explicit read/writes from/to mutable cells; memory operations that are also performed by functional computations are never tracked by STM unnecessarily, since they never need to be rolled back [HMPJH05].

1.2.2 Transactional Execution with `TVar`s

Threads in STM Haskell communicate by reading and writing transactional variables, or `TVar`s. All STM operations make use of the `STM` monad, which supports a set of transactional operations, including allocating, reading and writing transactional variables, namely the functions `newTVar`, `readTVar` and `writeTVar`, respectively, as it can be seen on Table 1.1.

Transactions are started within the `IO` monad by means of the `atomically` construct. When a transaction is finished, it is validated by the runtime system that it was executed on a consistent system state, and that no other finished transaction may have modified relevant parts of the system state in the meantime [HK05]. In

Running STM operations	Transactional Variable Operations
`atomically::STM a->IO a`	`data TVar a`
`retry::STM a`	`newTVar::a->STM(TVar a)`
`orElse::STM a->STM a->STM a`	`readTVar::TVar a->STM a`
	`writeTVar::TVar a->a->STM()`

TABLE 1.1. Haskell STM Operations

this case, the modifications of the transaction are committed, otherwise, they are discarded. The Haskell STM runtime maintains a list of accessed transactional variables for each transaction, where all the variables in this list which were written are called the "writeset" and all that were read are called the "readset" of the transaction. It is worth noticing that these two sets can (and usually do) overlap.

Operationally, `atomically` takes the tentative updates and applies them to the `TVars` involved, making these effects visible to other transactions. This method deals with maintaining a per-thread transaction log that records the tentative accesses made to `TVars`. When `atomically` is invoked, the STM runtime checks that these accesses are valid and that no concurrent transaction has committed conflicting updates. In case the validation turns out to be successful, then the modifications are committed altogether to the heap.

1.2.3 Other STM Tools

STM Haskell also provides means for composable blocking. The `retry` function of the `STM` monad aborts the current atomic transaction, and re-runs it after one of the transactional variables that it read from has been updated. This way, the atomic block does not execute unless there is some chance that it can make progress, avoiding busy waiting by suspending the thread performing `retry` until a re-execution makes sense. Conditional atomic blocks or join patterns can be implemented with the `orElse` method. This statement can be used to form alternative execution paths [HCU+07]: i.e. in `a orElse b`, if `a` invokes a `retry`, then `b` is executed; if `b` also retries, the whole statement does. The expressiveness provided by these two functions reflect the programmer's intent more accurately by allowing the runtime to manage the execution of the atomic block more efficiently and intelligently [HK05].

STM is robust to exceptions and uses methods similar to the exception handling that GHC provides for the `IO` monad. The function `atomically` prevents any globally visible state changes from occurring if an exception is raised inside the atomic block [HMPJH05].

```
data LinkedList =
     Start {nextN :: TVar LinkedList}
     | Node { val :: Int, nextN :: TVar LinkedList}
     | Nil
```

FIGURE 1.1. Data declaration for a transactional linked list in Haskell

1.3 PROBLEM STATEMENT

1.3.1 Lock-Free Linked Structures in Haskell

Linked structures such as lists, trees and graphs are simple but very useful mechanisms for modelling a wide range of abstractions in several application domains such as operating systems and compiler design. A desirable feature of these structures when they attempt to exploit concurrency is that they should be able to be shared and accessed safely and concurrently by many different threads. For this reason, lock-free shared memory data structures have long been studied in imperative languages [Fra03, Sun04, HLMS03]. The data declaration of a transactional singly-linked list, similar to [HPJ06], is shown on Figure 1.1. Here, a Haskell implementation that uses transactional variables which refer to the tail of the list is illustrated. A linked list is defined as a `Start` node that does not hold a value; (possibly) regular nodes holding a value and a transactional reference to the next node of the list; and `Nil`, the end of the list.

Figure 1.2 contains the code for a function that inserts an element in a sorted linked list of integers. The `readTVar` adds the current node to the readset, and by moving to the next node by reading its `TVar`, we are able to compare the two node values to see whether the insertion should take place in between. If this is the case, we create a new node, link it to the next node and write a `TVar` (inside the `newListNode` function), and then write the new link from the previous node to the new one by writing another `TVar`, as shown on line 8. In other words, list traversal can only be accomplished by reading `TVars`, and the links to and from the newly created node can only be applied by writing transactional variables.

1.3.2 The Large Readset Size Issue

Traversing a transactional linked structure (either to insert, delete or search an element) consists of reading as many transactional variables as the nodes that follow, i.e. a transaction to insert an element in the position *n* of a transactional linked list would have a readset with (*n-1*) elements at commit time. Given that *n* could be very large, some obvious questions arise: Is it necessary to collect all of the elements read by the transaction in the dataset? Is it possible to remove some of those variables that were read earlier in the traversal from the readset?

```
1  insertListNode :: TVar LinkedList -> Int -> STM ()
2  insertListNode curTNode numberToInsert = do
3    { curNode <- readTVar curTNode
4    ; let nextNTNode = nextN curNode
5    ; let doInsertion nextNNode =
6        do
7        { newNode <- newListNode numberToInsert nextNNode
8        ; writeTVar nextNTNode newNode
9        }
10   ; nextNNode <- readTVar nextNTNode
11   ; case nextNNode of
12       Nil -> doInsertion nextNNode
13       Node {val=valnextNNode, nextN=nextNnextNTNode}->
14         do
15         {if (valnextNNode == numberToInsert)
16           then return ()
17           else if (valnextNNode > numberToInsert)
18             then doInsertion nextNNode
19             else do
20               {insertListNode nextNTNode numberToInsert}
21         }
22   }
```

FIGURE 1.2. Linked List Node Insertion

1.3.3 The Problem of "False Conflict"

As a consequence of the large readset size issue, the false conflict problem appears. Imagine a list with 1000 elements and two threads operating over it, with one transaction (*T1*) wanting to insert or delete some element in the very beginning of the list, and another one (*T2*) wanting to operate on another element close to the end of the list. In order to get to the position to operate on, *T2* should accumulate a significant number of variables in its readset (pointers up to the current position). Now, if *T1* committed, one pointer near the beginning of the list (i.e. in *T2*'s readset) would be modified and as a result of this, *T1* would be rolled back. This scenario can be seen in Figure 1.3.

It can be intuitively said that with this approach, the probability of conflicts (and thus rollbacks) while inserting and deleting elements on a linked list is directly proportional to its length. To tackle this undesirable characteristic, it would be nice to "forget" those elements in the readset that are far behind with respect to the current position of the traversal. This approach, which was previously discussed by [HM93], and implemented on imperative languages by [Fra03, SK06] involves the use of inverse functionality of reading a transactional variable, i.e. "unreading" it. Thus, if we had not accumulated the variables in our readset indiscriminately, and instead had kept a fixed-size readset that moves forward as the list is traversed, the probability of conflicts would not have been as large as the length of the list [Her06]. The idea of avoiding false conflict with a fixed-size

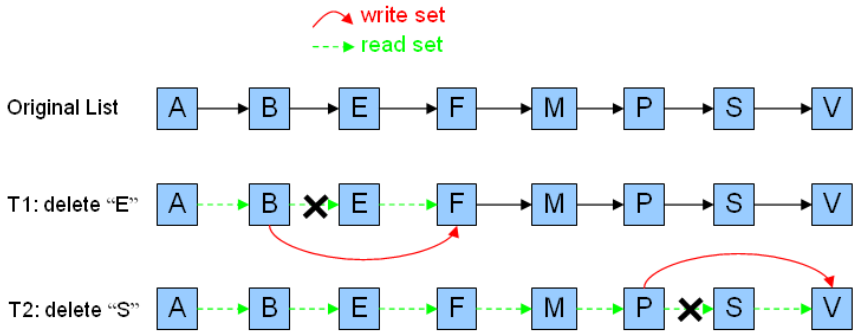

FIGURE 1.3. False Conflict

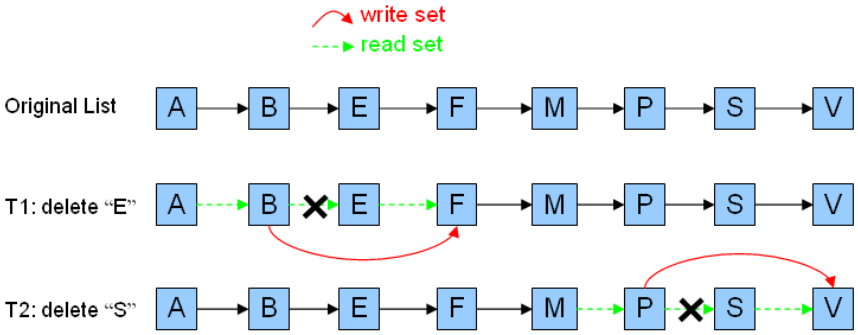

FIGURE 1.4. Avoiding False Conflict: *T1* and *T2* Operate on the List Concurrently

readset is depicted in Figure 1.4.

A smaller readset implies a fewer number of conflicts and aborts as discussed in [PSU+07]. In the next section, we relax the constraints of STM to be able to have a constant readset size on our linked structures, and thus we aim for a faster execution time.

1.4 PROPOSED SOLUTION: `unreadTVar`

Since we are using Haskell to perform our tests on STM and its API defines `readTVar` as the function to read a transactional variable, we decided to call the function that "forgets" variables in the readset `unreadTVar`. The signature of this function is as follows:

```
unreadTVar :: TVar a -> STM ()
```

The semantics of `unreadTVar x` are:

- If `x` is not in the current transaction's readset, nothing happens.
- If `x` is both in the current transaction's readset and its writeset, nothing happens.
- If `x` only in the current transaction's readset, `x` is removed from it.

Recall that when a transaction *T1* reaches the commit phase, it has to check whether another committed transaction *Ti* has changed the value of any of the variables in *T1*'s readset, and if so, *T1* must rollback. Removing a variable from the readset means that the transaction will not look after it anymore. In other words, at commit time it does not matter at all if that variable has been modified by another transaction or not.

1.4.1 Implementation of `unreadTVar`

The implementation of `readTVar` was done by adding a function to the transactional library implemented in C (*STM.c*) and "plumbing" it through until making it available as a regular Haskell function in the *Conc.hs* file. This function traverses the dataset to identify the `TVar` to "unread" and to expulse it if it has only been read (i.e. no writes have been performed on it). The already existing infrastructure was almost completely suitable for the modification and apart from adding the new `unreadTVar` function, only one line of existing code had to be altered in a non-harmful way.

A dataset in Haskell STM is maintained in a linked list of so-called chunks [HPST06]. Each chunk is a data structure that holds a limited number of transactional variable descriptors. When a chunk is full, another one has to be allocated and linked to the rest of the dataset to expand its capacity. Since the ordering of the variables within the set is not important, "unreading" was implemented by exchanging the last variable in the chunk with the unread one and decrementing the `nextFreePosition` index that refers to that chunk. Originally, the traversal of the dataset was done by reading all variables in each chunk except for the potentially incomplete one (the last one that has been inserted), in which case, the last variable to be examined was the `(nextFreePosition - 1)`st. The introduced modification consists of always traversing each chunk up to the last used position, because by adding the "unread" functionality, the variable that is "unread" can be in a chunk that is not the current one (last one that has been added) and thus causing incomplete chunks in the middle of the linked list of chunks. Finally when the transaction commits, all chunks are freed.

1.4.2 Using `unreadTVar`

As an example of the intended use, Figure 1.5 presents the modification that has to be done to increase the performance and the memory usage of the function previously shown in Figure 1.2. In this case, by using `unreadTVar`, the size of the readset varies between one and three elements on each iteration. Without `unreadTVar` use, the readset grows by one element on each iteration, and

```
19            else do
              { unreadTVar curTNode
20            ; insertListNode nextNTNode numberToInsert
              }
```

FIGURE 1.5. `unreadTVar`'s Use in Linked List Node Insertion

finally, in the extreme case of inserting or deleting the last element of the list, contains the whole set of elements of the list.

Furthermore, it is worth noticing that even with this approach, there is still a chance that two transactions that operate on elements that are far away from each other conflict, but this is quite unlikely to happen, i.e. the transaction on the element that is closer to the beginning of the list must commit while the other transaction is traversing the same point of the list.

1.5 PERFORMANCE VERSUS SAFETY

1.5.1 Experimental Results

As pointed out above, having multiple transactions that operate on list elements that are far away from each other (with respect to traversal order) is a scenario that harms the performance of traditional linked structures because of the large number of rollbacks that will take place. To reproduce these settings, a program which forks two threads was created, where each of these threads atomically inserts and atomically deletes one element that is close to the beginning, and another one close to the end of a linked list. This way, the duration of each thread is the same on average, and there is a variety of (both long and short) transactions involved in the execution. Figure 1.6 summarizes the results of several such experiments concerning lists of different sizes, performed on an Intel 1.66 GHz dual-core machine with 1 GB of RAM and 2 MB shared L2 cache, running Linux. Two threads operate as described above, on the third element from the beginning and on the third from the end, with list sizes varying from 10 to 10,000.

As it can be seen, without using `unreadTVar`, the bigger the list, the more time it takes on average to operate on an element when the list has a considerably large size. This is because a bigger readset implies two overheads: The probability of re-execution because of rollbacks is greater, and on the other hand, commits are slower because there are more variables to check for conflicts [PSU+07]. Working with a small list, there is a high chance of conflict between transactions, no matter whether `unreadTVar` is used or not, causing the "U-shape" of the curves. As explained in Section 1.3.3, the probability of conflicts always increases with list size.

In order to investigate the scalability of `unreadTVar`, more experiments have been conducted on a four dual-core processor SMP server with Intel Xeon 5000 processors running at 3.0 GHz with 4MB L2 cache/processor and 16GB of

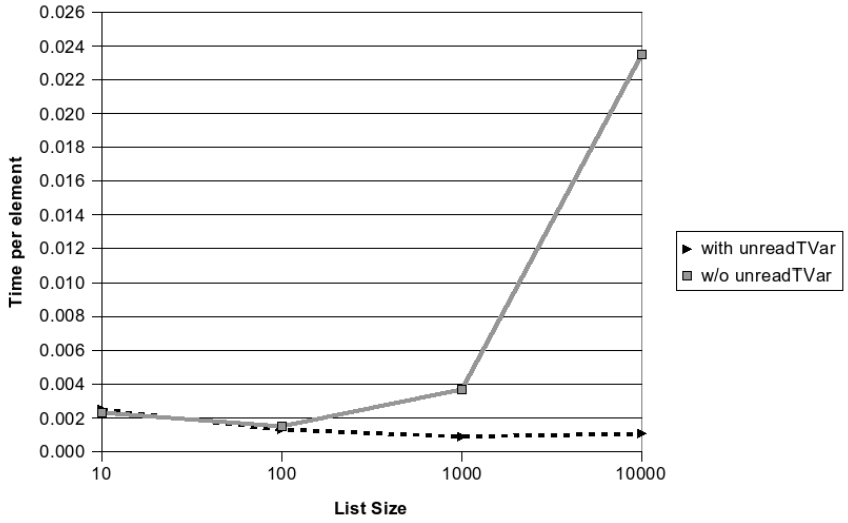

FIGURE 1.6. Time Taken per Element at Different List Sizes

total memory, where all of the reported results are based on the average of five executions. For the sake of variety regarding to the applications, binary trees were also analysed in an analogous way to the linked list. The basic experiment consisted of 8000 operations on one shared data structure (i.e. either linked list or binary trees), initially filled up only with even numbers to hold half of its capacity. Each operation was one atomic insertion of a random number and one atomic deletion of another one, both between 1 and 10000. The workload has been divided for multithreaded versions in such a way that the total number of operations performed for all threads is equal to 8000. For each data structure, three versions of the program were developed: a transactional version that uses the original STM library, another transactional version that uses the modified STM library to include the `unreadTVar`, and finally a sequential version implemented using `IORefs`. `IORefs` are analogous to `TVars` in the sense that both types of variables provide access to mutable memory cells, however; `IORefs` do not incorporate any synchronization mechanism. In Figures 1.7 and 1.8, the execution times are plotted for both data structures showing that:

- For both structures, the `unreadTVar` outperforms the original transactional library. Although more measurements might be needed on larger systems to confirm this, the results suggest that further performance improvements are possible beyond four dual-core processors.

- The `unreadTVar` version scales better for the linked list than the original STM approach. More number of cores would be needed to make a conclusion for the case of the trees.

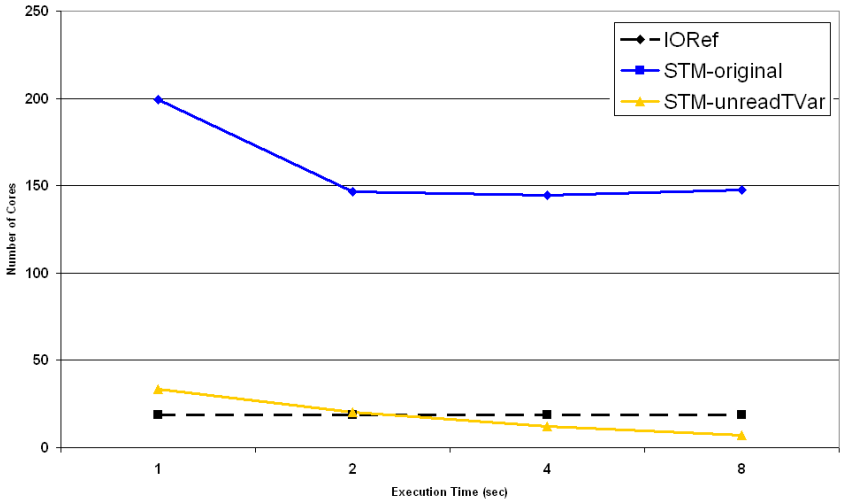

FIGURE 1.7. Scalability of `unreadTVar` on Linked Lists

- There is a breakpoint where the `unreadTVar` version starts showing absolute speedup. With eight cores, the `unreadTVar` version is 274% and 190% faster than the sequential `IORef` version for linked lists and binary trees respectively.
- The regular transactional version is unable to show any absolute speedup even with 8 cores, compared to the sequential `IOref` version.

Based on the observations itemized above, `unreadTVar` turned out to be substantially faster than the original transactional version for transactions with a larger readset because it reduces the probability of conflicts and, therefore, rollbacks. The synthetic applications presented in this paper act as a proof of concept that the technique is useful in cases where linked structures are heavily utilized. For example, CCHR [LS07], a constraint solver developed in Haskell with TM as the parallel programming model, uses linked lists to write and read shared data that consists of the rules that the solver is trying to simplify.

1.5.2 Safety Issues

Imagine that we want to implement the function `sum::LinkedList->STM Int` that calculates the sum of all the elements in a linked list. Then, a subtle difference in the semantics will appear if one version that uses `unreadTVar` is compared to another one that does not use it. In the first case, there is a possibility that the calculation yields a number that represents the sum of a list that never existed. To see this, imagine that in time *t* when list traversal is around the middle of the list, one element *A* is deleted near the beginning of the list and by the time

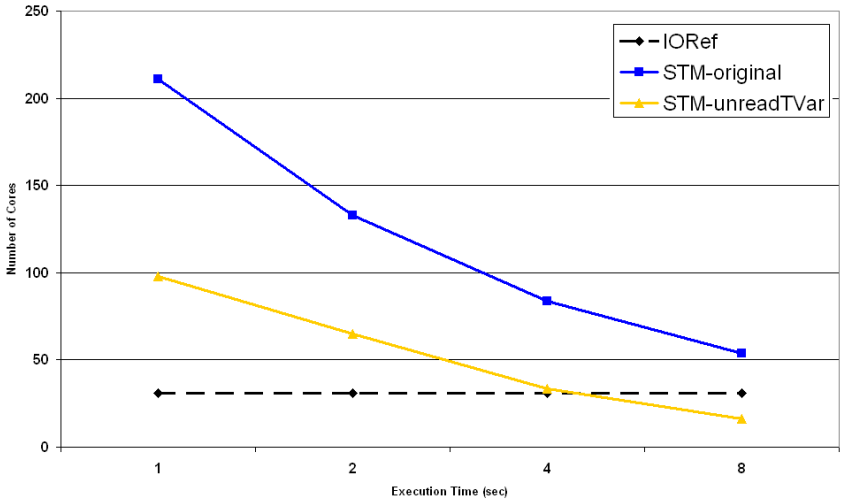

FIGURE 1.8. Scalability of `unreadTVar` on Binary Trees

t + 1, another element *Z* is deleted around the end. Now, when the calculation of the sum is over, the value will be equal to the addition of the list containing the element in the beginning but without containing the one in the end, and actually such a list never existed. This condition could be valid or invalid depending on the programmer's criteria. Figure 1.9 shows the list at time *t*, at time *t + 1*, with *A = 1* and *Z = 9* and the non-existent list seen by the *sum* function. So, by changing the strong semantic guarantees of STM, possible race conditions are introduced. Depending on the abort policy, if the readset is completely emptied, at that exact point there is no reason to rollback the transaction. Therefore, to comply with the semantics of the program and to avoid race conditions, the dangers of completely emptying the transaction's readset by using `unreadTVar` should be taken into account.

As a price for the performance improvement discussed previously, the safety promises of STM have been broken. Now the application has left the safe grounds and might behave inconsistently. This is the reason why while using `unreadTVar`, the programmer has to be aware of these consequences and always has to keep an eye on the readset. As a light-weight approach to drawing a line between the safe and the unsafe world, a Haskell module called `UnsafeSTM` can be introduced (Figure 1.10). This module could be used alongside an unsafe lifting function to bring a computation into the (safe) STM monad. This would make it clear to the programmer that she is leaving the safe ground provided by Haskell's type system. Although with this method the syntax is made a little more complex, writing programs this way would also make it clearer at which points `unreadTVar` is usefully applied.

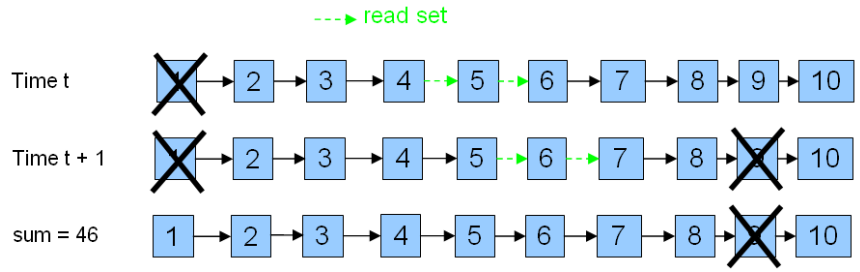

FIGURE 1.9. Calculation of the Sum of a List that Never Existed

```
module UnsafeSTM where
import GHC.Conc -- The STM library
newtype UnsafeSTM a = STM a
unreadTVar :: TVar a -> UnsafeSTM ()
unsafeLiftSTM :: UnsafeSTM a -> STM a
```

FIGURE 1.10. The unsafeSTM module

Since list insertion and deletion allow the use of `unreadTVar`, whereas calculating the list sum does not, the necessity of a criterion to decide whether to use the function or not arises. According to [HLMS03], this operation that does conflict reduction is "for designing shared pointer-based data structures such as lists and trees in which a transaction reads its way through a complex structure". The idea is that if it is necessary to have a snapshot of the linked structure that is being worked on, `unreadTVar` should not be used. On the other hand, if the programmer, during traversal, only cares about a smaller area around the current node and not the whole structure, then it is a candidate situation to allow forgetting objects. The authors of this paper call this "the snapshot criterion".

One transaction that uses `unreadTVar` can coexist in the same program with another that does not use it. However, in terms of composability, they can not be composed together safely inside the same atomic block, as the reduced readset that `unreadTVar` introduces might remove some variables that are useful for the other function to operate correctly. In other words, composing the sum function together with an insertion function that uses `unreadTVar` inside the same atomic block is a situation that must be refrained from.

Missing an opportunity for `unreadTVar` does not affect the original semantics of the program, however, adding too many could harm correctness. The `unreadTVar` could also be used on an already-complete program for fine-tuning the transactions for increased performance [SK06].

1.6 CONCLUSIONS

This paper introduced an efficient mechanism, the `unreadTVar`, to increase the performance of applications that use linked structures realized transactionally on Haskell. This was achieved by modifying the Haskell STM and adding the functionality of removing items from the readset of a transaction. Our results have shown consistency with the imperative versions of this functionality, where to the best of our knowledge, this was the first implementation for a functional language. We have provided performance comparisons run on actual hardware that showed substantial performance improvements due to the mechanism. The `unreadTVar` approach made two improvements to the implementation of transactional linked lists. First, by providing the transactions with a smaller dataset to work with, it significantly decreased the probability of having rollbacks. Second, since a smaller number of `TVar`s had to be checked for consistency before committing, commits were made faster.

The most important drawback of using the `unreadTVar` is that it requires more care from the programmer. Since it is used for performance optimization, it requires the knowledge of exactly when and with which variable to use it. Specifically, as shown in the case in Figure 1.9, the usage does not apply for all sorts of operations that can be done with a linked structure, but preferably only the ones that fit the snapshot criterion. Although we attempted to add an extra level of security with the lifting function while using unreadTVar, the approach might be more accurate with a compiler that is able to identify where to use it safely. Since compared to the sequential version, the original transactional version is unable to show any speedup even with 8 cores, approaches like the `unreadTVar` are necessary as significant ways of optimizing the performance of STM in Haskell.

ACKNOWLEDGEMENTS

This work is supported by the cooperation agreement between the Barcelona Supercomputing Center - National Supercomputer Facility and Microsoft Research, by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contract TIN2004-07739-C02-01 and by the European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC). The authors would like to thank Tim Harris, Eduard Ayguadé Parra, Roberto Gioiosa, Paul Carpenter, all at BSC-Nexus-I, Marco T. Morazán and the anonymous reviewers for all their helpful suggestions.

REFERENCES

[DFL+06] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. *SIGOPS Oper. Syst. Rev.*, 40(5):336–346, 2006.

[DHM+06] Anthony Discolo, Tim Harris, Simon Marlow, Simon L. Peyton Jones, and Satnam Singh. Lock-free data structures using stm in haskell. In *Eighth*

International Symposium on Functional and Logic Programming (FLOPS), volume 3945 of *Lecture Notes in Computer Science*, pages 65–80. Springer, 2006.

[Fra03] Keir Fraser. *Practical lock freedom*. PhD thesis, Cambridge University Computer Laboratory, 2003. Also available as Technical Report UCAM-CL-TR-579.

[HCU+07] Tim Harris, Adrian Cristal, Osman S. Unsal, Eduard Ayguade, Fabrizio Gagliardi, Burton Smith, and Mateo Valero. Transactional memory: An overview. *IEEE Micro*, 27(3), 2007.

[Her06] M. Herlihy. Course slides for cs176 - introduction to distributed computing: Concurrent lists, 2006.

[HK05] Frank Huch and Frank Kupke. Composable memory transactions in concurrent haskell. In *17th International Workshop on Implementation and Application of Functional Languages (IFL)*, 2005.

[HLMS03] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing*, pages 92–101. ACM Press, 2003.

[HM93] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of theTwentiethAnnual International Symposium on Computer Architecture*, 1993.

[HMPJH05] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM Press.

[HPJ06] Tim Harris and Simon Peyton-Jones. Transactional memory with data invariants. In *First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.

[HPST06] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 14–25. ACM Press, 2006.

[htt] http://www.haskell.org. Haskell official website.

[III04] Hal Daume III. Yet another haskell tutorial, 2004.

[JGF96] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Conference Record of POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308, St. Petersburg Beach, Florida, 21–24 1996.

[KCJ+06] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. In *Proceedings of Symposium on Principles and Practice of Parallel Programming*, Mar 2006.

[LS07] Edmund S. L. Lam and Martin Sulzmann. A concurrent constraint handling rules implementation in haskell with software transactional memory. In *DAMP '07: Proceedings of the 2007 workshop on Declarative aspects*

of multicore programming, pages 19–24, New York, NY, USA, 2007. ACM Press.

[PSU+07] Cristian Perfumo, Nehir Sonmez, Osman S. Unsal, Adrian Cristal, Mateo Valero, and Tim Harris. Dissecting transactional executions in haskell. In *TRANSACT 07*. Aug 2007.

[SK06] Travis Skare and Christos Kozyrakis. Early release: Friend or foe? In *Workshop on Transactional Memory Workloads*. Jun 2006.

[ST95] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213. ACM Press, 1995.

[Sun04] Hakan Sundell. *Efficient and Practical Non-Blocking Data Structures*. PhD thesis, Department of Computing Science, Chalmers University of Technology, 2004.