Abstract nested transactions

Tim Harris
Microsoft Research
tharris@microsoft.com

Srđan Stipić [*]
Barcelona Supercomputing Center
srdjan.stipic@bsc.es

Abstract

TM implementations typically use low-level conflict detection based on the memory locations that a transaction accesses. This can cause transactions to be re-executed because of *benign conflicts*, for example if two transactions insert items into the same hashtable under different keys which happen to hash to the same bucket: the insertions update the same memory locations, even though the higher-level operations they are performing are commutative.

In this paper we introduce *abstract nested transactions* (ANTs) as a way of improving the scalability of atomic blocks that experience some kinds of benign conflict. The main idea is that ANTs should contain operations that are likely to be the victims of benign conflicts. The run-time system then performs extra book-keeping so that, if an ANT does experience a conflict, the ANT can be re-executed without needing to re-run the larger transaction that contains it. Unlike closed nested transactions (CNTs) this re-execution can happen *after* the ANT has finished running – in our implementation we only re-execute ANTs at the point that we try to commit a top-level atomic block.

Moving code into or out of an ANT is *semantics preserving*: ANTs affect only the program's performance, not its possible results. This helps open the door for future research in automatic ways to place ANTs in programs in order to deal with contention 'hot spots' that they experience.

1. Introduction

Atomic blocks provide a promising simplification to the problem of writing concurrent programs. A code block is marked `atomic` and the compiler and run-time system are responsible for ensuring atomicity during its execution. The programmer no longer needs to worry about manual locking, low-level race conditions or deadlocks.

Atomic blocks are typically implemented over *transactional memory* which provides the abstraction of memory read and write operations which can be grouped together to form a transaction and then committed to the heap as a single atomic step. Rajwar and Larus' recent book summarizes many of the hardware, software, and hybrid implementation techniques being explored [10].

[*] The design and implementation of ANTs was completed while Srđan Stipić was working at Microsoft Research.

```
atomic { // Tx-1                atomic { // Tx-2
  performWork(g_o1);              performWork(g_o2);
}                               }
```

```
void performWork(Object o) {
  g_invocation_count ++;
  // Work on 'o'
}
```

(a) An ordinary implementation of `performWork` introduces conflicts via the global invocation counter.

```
void performWork(Object o) {
  ant {
    g_invocation_count ++;
  }
  // Work on 'o'
}
```

(b) Rewriting `performWork` to use an ANT causes the higher-level update operation to be logged, rather than the low level reads and writes it performs.

Figure 1. A benign conflict between `Tx-1` and `Tx-2`.

In this paper we highlight a number of ways that the performance of programs can suffer when based on TM. This happens due to *benign conflicts* that can occur between transactions. In each of these cases the TM implementation can force one or more transactions to abort because it detects a conflict which does not really matter to the application.

Figure 1(a) shows a running example that we will use. We use a '`g_`' prefix on variable names to indicate that they are shared between threads. Other variables are thread-private. The example involves two transactions, `Tx-1` and `Tx-2`, which call '`performWork`' on different objects. The function increments a count of the number of times that it has been called and then performs some work on the object that it has been passed. The updates to the shared counter will cause `Tx-1` and `Tx-2` to conflict with one another, leading to the re-execution of a whole `atomic` block even if the bulk of `performWork` is non-conflicting.

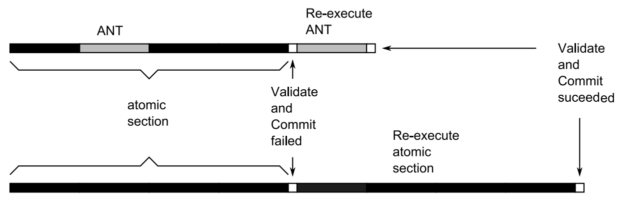

ANT
Re-execute ANT
Validate and Commit suceeded
atomic section
Validate and Commit failed
Re-execute atomic section

Figure 2. Execution time of an atomic section with ANT (top) and same atomic section without ANT (bottom).

We introduce a fuller taxonomy for different variants of this problem in Section 2. As we discuss with the taxonomy, there are several ways that programmers can try to avoid problems from benign conflicts. In some cases it is possible to restructure code, to use *open nested transactions* [13], or to use TM-specific optimization interfaces to avoid benign conflicts. All of these approaches have their problems: manually restructuring code can harm the composability benefits of `atomic` blocks, open nested transactions provide a powerful general purpose abstraction but one which relies on programmer care for correct usage, and TM-specific optimization interfaces are hard to use correctly.

In this paper we introduce a new approach to avoid re-executing whole `atomic` blocks on some kinds of benign conflict. The idea, which we call *abstract nested transactions* (ANTs), is to identify operations that are likely to be the 'victims' of benign conflicts. This lets the TM implementation keep a separate log of the operations being performed by ANTs and, in the event of a low-level conflict, just re-execute the ANTs rather than re-executing the larger transaction containing them. Figure 1(b) shows our example using an ANT to increment the counter.

Figure 2 shows why this can be faster: assuming that the ANT forms a small part of the overall execution time of an `atomic` block, it reduces the amount of work on a conflict.

We discuss the syntax and semantics of ANTs in Section 3. A fundamental design principle we took is that they are *semantically transparent*: marking a block of code as an ANT does not affect the possible results of a program. Our motivation in doing this is that it makes ANTs easier to use and, although we show them being used manually in this paper, in future work they could be placed automatically based on run-time feedback.

This principle guides many aspects of the design and implementation of ANTs: what happens if an ANT conflicts with a transaction that encloses it? What happens if an ANT raises an exception or tries to block using `retry` [5]? What happens if an ANT behaves one way on its first execution and then behaves differently if it is re-executed? We discuss these questions in Section 3.

We have prototyped our implementation of ANTs using STM Haskell [5]. In Sections 4-5 we describe the details of this implementation and evaluate the performance impact of using ANTs. Unlike much of our earlier work on STM Haskell supporting ANTs is actually *more difficult* in Haskell than it would be in other languages.

In particular, we make the following contributions:

- An initial taxonomy of *benign conflicts* that can occur between atomic blocks.
- The idea of *abstract nested transactions* as a way of making the performance of atomic blocks more robust to the presence of some kinds of benign conflict.
- An implementation of ANTs in which they are semantically transparent, being able to be placed around any transactional code and affecting only its performance.
- Specific to our Haskell-based prototype, we describe a new mechanism for comparing possibly-non-terminating computations using lazy evaluation.

Throughout the paper we assume an implementation over an STM using *lazy conflict detection* [12] (that is, detecting conflicts at commit-time in short-running transactions, and periodically in long-running transactions) and *lazy versioning* [12] (that is, recording tentative updates privately for each transaction, and writing them to the heap upon successful commit).

2. Benign conflicts

In this section we identify a number of different kinds of *benign conflict* that can cause a transaction to abort because of the low-level at which conflicts are typically detected in TM implementations. This is an intentionally imprecise definition; as we illustrate, whether or not a given conflict is benign depends on the context in which the transaction experiencing it occurs. However, distinguishing different kinds of benign conflict helps us identify the cases where ANTs are useful and the cases where they are not.

We focus solely on `atomic` blocks with serializable semantics. As the database community has explored, weaker isolation levels can reduce conflicts [1]. However weaker isolation also means that it is no longer possible to reason about `atomic` blocks as executing in isolation from one another; it is unclear whether this would be acceptable as part of a mainstream programming model.

2.1 Shared temporary variables

The first kind of benign conflict occurs when global variables are used for transaction-local storage. Figure 3 shows an example: each transaction starts by writing to `g_temp` and then using it for its own temporary storage. We have seen this in practice when using the `xlisp` interpreter on Bartok-STM [7] and also in the red-black tree implementation that Fraser used in his PhD work [4] in which transactions working near the leaves of the tree will write and then read values in a *sentinel node* [3] whose contents do not need to be retained between operations on the tree.

```
atomic { // Tx-1            atomic { // Tx-2
  workOn(g_o1);               workOn(g_o2);
}                           }

       void workOn(Object o) {
         g_temp = o;
         // Work on 'g_temp'
       }
```

Figure 3. Shared temporary variables. Transactions `Tx-1` and `Tx-2` will conflict because they both write to `g_temp`, even though neither depends on the values written by the other.

```
atomic { // Tx-1            atomic { // Tx-2
  g_obj.x ++;                 g_obj.y ++;
  // Private work             // Private work
}                           }
```

Figure 4. False sharing. Transactions `Tx-1` and `Tx-2` will conflict using the Bartok-STM implementation which detects conflicts on a per-object basis.

Fraser provides a mechanism for disabling conflict detection on such data [4]; if this is mis-used then transactions may no longer be serializable.

Haskell-STM identifies the special case of *transactionally-silent stores* in which a transaction makes a series of updates to a shared field, but the value at the end of the transaction is the same as at the start: the overall access can then be treated as a read rather than a write. This can increase scalability in some cases. However, not all shared temporaries are used in this way.

2.2 False sharing

False sharing occurs when the granularity at which TM detects conflicts is coarser than the granularity at which `atomic` blocks access data.

Figure 4 illustrates this with an example pair of transactions that conflict when run over the Bartok-STM implementation [7]: the two transactions conflict because they both write to fields in the same object. False sharing can also occur in HTMs – for example if conflicts are detected on a per-cache-line basis and two transactions update different words on the same cache-line. Zilles and Rajwar analyze the problem of false sharing in TM implementations that use tagless ownership tables, showing that it may happen more frequently than intuition suggests [15].

In software, false sharing can be avoided by splitting objects into portions that are likely to be accessed separately – the ability to do this is one motivation for detecting conflicts at an object-granularity because it lets the programmer control conflicts when deciding which fields to place in the same object.

```
atomic { // Tx-1         atomic { // Tx-2
 o1 = remove(g_ht, 39);   o2 = remove(g_ht, 49);
 // Work on 'o1'          // Work on 'o2'
 insert(g_ht, 39, o1);    insert(g_ht, 49, o2);
}                        }
```

Figure 5. Using commutative operations with low-level conflicts. Transactions `Tx-1` and `TX-2` work on objects that they look up from hashtable `g_ht` under different keys. The hashtable operations may introduce conflicts if the keys hash to the same bucket in the table.

2.3 Using commutative operations with low-level conflicts

A further source of benign conflicts occurs when transactions use commutative operations that introduce low-level conflicts. When we say that operations `A` and `B` on a shared object are commutative we mean that there is no difference in executing `A`-before-`B` or `B`-before-`A` in terms of the operations' results or the subsequent behavior of the shared object.

One example is the shared counter from Figure 1(a): the increment operations are commutative, but the read and writes that they perform are not.

Another example of this kind of benign conflict is lazy initialization: a data item may have its value computed on-demand by the first transaction to access it. In many cases the computation can safely be performed more than once (wasting time, but giving the same results), although in other cases this is not true (e.g. in implementations of the singleton design pattern, where a common shared object is being instantiated).

Figure 5 shows a more complicated example: two transactions access a shared hashtable (`g_ht`) and perform operations on different keys. These are likely to conflict in the memory locations that they access if the keys happen to hash to the same bucket in the table.

Ni *et al.* use an example like this to motivate the use of *open-nested transactions* (ONTs) [13]. Using ONTs it is possible to prevent `Tx-1` and `Tx-2` from conflicting by (*i*) running the `remove` and `insert` operations in ONTs so that they are performed directly to the hashtable when they are invoked inside transactions, (*ii*) defining compensating operations to roll-back any tentative operations that are made by transactions that subsequently abort, (*iii*) using *abstract-locks* to prevent concurrent transactions performing non-commutative operations on the hashtable – for example insertions under the same key.

Versioned boxes [2] provide mechanisms for dealing with some kinds of low-level conflcit between commutative operations: *delayed computations* that execute at commit time, and *restartable transactions* that perform read-only operations that can be re-executed at commit time to check for benign conflicts. ONTs provide a more general-purpose mechanism to tackle many problems, including this one, but they

```
atomic { // Tx-1          atomic { // Tx-2
 f = listFind(g_l, 1000);  listInsert(g_l, 10);
}                         }

      List listFind(List l, int key) {
        while (l.Next.Key <= key) {
          l = l.Next;
        }
        return l;
      }

      bool listContains(List l, int key) {
        l = listFind(l, key);
        return (l.Key == key);
      }

      void listInsert(List l, int key) {
        l = listFind(l, key);
        if (l.Key != key) {
          l.Next = new List(key, l.Next);
        }
      }
```

Figure 6. Defining commutative operations with low-level conflicts. Transactions `Tx-1` and `Tx-2` will conflict in their accesses to the shared list `g_l` holding sorted integers: `Tx-1` will traverse the list up to the node holding 1000, and `Tx-2` will conflict with these reads when it inserts a node holding 10.

rely on programmer care in defining the compensating actions and abstract locking discipline in order to ensure that atomic blocks using them remain serializable.

2.4 Defining commutative operations with low-level conflicts

A further source of benign conflicts occurs *within* the definition of commutative operations. Figure 6 shows an example: two transactions access a sorted linked list of integers, with `Tx-1` searching the list for an item 1000 and `Tx-2` inserting an item 10. If we assume that the list contains many elements then `Tx-1` will build up a large read-set and conflict with `Tx-2` and any other transactions making updates to the list in the range 1..1000.

ONTs do not provide an obvious solution to this problem: the atomic blocks consist of a single operation on a list, which must be performed atomically whether it is in an ordinary transaction, or in an open one. However, many STM implementations have included 'back doors' by which expert programmers can remove accesses from a transaction's read-set that they believe are unnecessary [9]. In this case `listFind` could be rewritten to retain only its accesses to nodes in the vicinity of the key: earlier nodes would be removed from the read-set and concurrent updates to these nodes would not be treated as conflicts.

```
while (true) {            while (true) {
 atomic { // Tx-1          atomic { // Tx-2
  t = getAny(g_in);         t = getAny(g_in);
  if (t == null) break;     if (t == null) break;
  // Work on t              // Work on t
  put(g_out, t);            put(g_out, t);
 }                         }
}                         }
```

Figure 7. Making arbitrary choices deterministically. Transactions `Tx-1` and `Tx-2` both take work items from an input input pool (`g_in`), work on them, and place the results in an output pool (`g_out`). A deterministic implementation of `getAny` will lead them both to pick the same item.

Using these operations correctly requires great care from the programmer. For example, using them here leads to similar search and insert functions to the non-blocking linked list algorithms by Harris [8] and Michael [11]. Furthermore, adding an additional operation to a data structure can make the implementation of existing operations incorrect. For example, if we added `listDeleteFrom` implemented by cutting off the tail of a list at a specified element, then it would no longer be correct to remove elements from the read-set during a call to `listContains`.

2.5 Making arbitrary choices deterministically

A final example of benign conflict is caused by *making arbitrary choices deterministically*. Figure 7 shows an example. Two threads repeatedly take items from a pool of input items `g_in`, work on them, and place them into an output pool `g_out`. All of the items must be processed, but it does not matter what order this happens in.

If we assume that the input pool is implemented by a shared queue then the two threads' transactions will conflict because they will deterministically select the first item from the queue even though, in this context, any item is acceptable.

ONTs can be used in this case: each thread executes `getAny` in an ONT and uses a compensating action to replace the item. There is one subtlety to beware of in this example – a transaction observing the queue to be empty can only be allowed to commit once there is no possibility of a concurrent call to `getAny` being compensated.

2.6 Discussion

In this section we have introduced a number of ways in which programs can experience benign conflicts. There are a wide range of existing techniques addressing parts of this problem space:

- Converting shared temporaries into transactionally-silent stores reduces conflicts using some STM implementations (Section 2.1) and restructuring how data is partitioned between objects can reduce false sharing (Section 2.2).

- Open nested transactions can be used to avoid serializing commutative operations (Section 2.3) and avoid making arbitrary choices deterministically (Section 2.5).
- Manual optimization interfaces can be used to trim unnecessary reads from transactions read-sets (Section 2.4).

Common to all of these is the use of manual techniques that introduce a risk of changing the behavior of the `atomic` blocks as well as their performance: if they are mis-used then ONTs and read-set reduction interfaces can lead to non-serializable executions of `atomic` blocks.

Our goal is to explore how far we can go with techniques without that risk. The 'abstract nested transactions' in this paper are the first step in that direction. In particular, we aim to tackle the problems of false conflicts (Section 2.2) and atomic blocks using commutative operations with low-level conflicts (Section 2.3).

Why do we not tackle the other problems? Essentially because we believe they are best tackled elsewhere. First, it is likely that shared temporary variables can be identified automatically by modifications to the TM implementation. Second, we believe that scalable implementations of data structures involving arbitrary choice can be built over `atomic` blocks and ANTs using randomization techniques similar to those in Scherer's exchanger [14] – in effect, making the operations *non-deterministic* where possible. Third, we hope that some cases of read-set reduction can be made possible by compile-time analyses. Whether or not these techniques are effective is the subject of future work.

3. Abstract nested transactions

The key idea with ANTs is to identify operations, like those in Sections 2.2-2.3, which are likely to be the victims of benign conflicts when executed over TM. For example, the hashtable operations in Figure 5 could be executed inside ANTs, as could the accesses to `g_obj.x` and `g_obj.y` in Figure 4.

We chose the name ANTs because the programmer can think of them as being handled at a different level of abstraction from the ordinary reads and writes that a transaction performs. For example, an `atomic` block that inserts data into a hashtable within an ANT will only be forced to re-execute if a concurrent transaction inserts a conflicting item into the table, rather than (with a typical hashtable implementation) if a concurrent transaction inserts a value that happens to hash to the same bucket.

We do this without programmer annotations about which operations conflict with one another. Instead, we perform extra book-keeping at run-time which lets us (*i*) identify benign conflicts involving ANTs, (*ii*) recover from benign conflicts by just re-executing the ANTs, rather than re-executing the `atomic` block that contains them.

The main difficulty, of course, is ensuring that it is correct to just re-execute the ANTs. We discuss the mechanisms we use to do this in Section 4 after first discussing the syntax (Section 3.1) and semantics (Section 3.2) of ANTs, and programmer guidelines for how to use them to improve performance (Section 3.3).

3.1 Syntax

The exact way that ANTs are exposed to programmers will depend on the language. In the psuedo-code example in Figure 1(b) we suggested using `ant` blocks. We will use this concise form for examples in the remainder of the paper, both as stand-alone `ant` blocks (`ant{X}`) and as `ant` blocks that return a result (`Y = ant{X}`).

In practice, in a language like C# or Java, it would be more natural to express ANTs using an attribute on individual method signatures, or on a class in which case all methods on the class would execute in ANTs. That would be consistent with the expectation that all operations on a given shared object would be performed through ANTs. Of course, many other possibilities can be imagined, such as creating an ANT-wrapper around an existing object, or designating an object as ANT-wrapped at the point that it is instantiated.

3.2 Semantics

ANTs are semantically transparent. In our pseudo-code, running `ant{X}` is always equivalent to `X`, no matter what operations are performed in `X` and what context the ANT occurs in. The same is true for ANTs returning a result.

This is an important decision: it means that the addition or removal of ANTs is based purely on performance considerations, making it easier to use feedback-directed tools to identify contention. We did consider whether we could use a different implementation of ordinary *closed nested transactions* (CNTs) instead of introducing new notation for ANTs. However, most languages that expose CNTs via nested `atomic` blocks choose to allow exceptions raised inside a CNT to roll-back the nested transaction. This means it is not possible to add or remove CNTs without considering semantic changes to the program.

Another important consequence of our design decision is that an `atomic` block containing ANTs can always be executed in *fallback-mode* in which the ANTs are executed without any special run-time support. We use this idea to simplify our prototype implementation.

3.3 Performance

Our implementation of ANTs is based on re-executing ANTs that experience conflicts without needing to re-execute the whole `atomic` block that contains them.

For instance, using the hashtable example from Figure 5, if the two keys hash to the same bucket then the second transaction to commit can experience conflicts from the first. However, if the `remove` and `insert` operations are implemented using ANTs then the second transaction will abandon its initial execution of `remove` and `insert` and re-execute just these operations rather than needing to re-execute the entire `atomic` block. If the re-execution succeeds without conflict, if the result returned by `remove` is the same upon re-execution, and if this result is the only way that the ANTs interacted with the outer block, then the atomic block and the re-executed ANTs can be committed.

It is important for the programmer to understand how to use ANTs if they are to achieve good performance. The rules are:

- A given piece of data should be consistently accessed inside ANTs, or consistently accessed outside ANTs. Consider the following example:

  ```
  atomic {
    ant { g_temp = remove(g_ht, 39); }
    // Work on 'g_temp'
  }
  ```

  In this example the ANT interacts with the rest of the `atomic` block through `g_temp`, unlike Figure 5 where the interaction is solely through the return value from the ANT. We detect this kind of inconsistent use of ANTs at run-time and switch to fallback-mode.
- ANTs should constitute a small portion of the execution time of the `atomic` block that contains them. Otherwise, there is no practical gain by re-executing just the ANTs.
- ANTs should be likely to experience conflicts whereas the rest of the `atomic` block should not. Otherwise, the whole `atomic` block is likely to be re-executed in any case.

4. Prototype implementation

We have implemented support for ANTs in the *run time system* (RTS) of the Glasgow Haskell Compiler (GHC). Aside from the subtlety discussed in Section 4.3 about detecting equality between Haskell values, the implementation should be applicable to other languages using similar STM algorithms. Although STM-Haskell does not expose nested `atomic` blocks to the programmer, our implementation of ANTs *does* support closed nested transactions which are used internally by the GHC RTS in its implementation of exception handling and the `orElse` and `retry` constructs for composable blocking [5].

GHC's existing support for STM uses lazy conflict detection and lazy versioning using transaction logs that keep track of shared memory accesses. Each transaction's log is a list of log entries containing the following fields: the *memory address* being accessed, the *old value* that the transaction expects to be stored there and the *new value* that it wants to write there. Every read or write to a shared memory location is performed first by scanning through the transaction log and, if the location is not found in the log, the memory access is performed and a new log entry is created in the log.

The STM implementation allows validation and commit operations to run in parallel so long as the locations written to by one transaction do not overlap the locations read or written to by another [6].

4.1 Changes when executing an `atomic` block

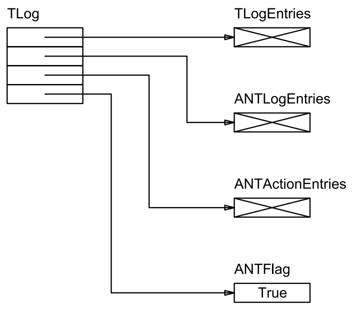
While executing an `atomic` block we differentiate between memory accesses made from within ANTs and those made from the enclosing transaction. To achieve this, the transaction log keeps entries in two separate lists: one list records the accesses in the ANTs, and the other list records all the normal transactional accesses.

The structure of the transaction log (TLog) can be seen on the Figure 8(a). The TLog has four fields. TLogEntries holds the normal transactional accesses. ANTLogEntries holds the accesses made within ANTs. ANTActionEntries records the high-level operations being performed by ANTs. Each is represented by a pair of pointers. The first points to the block of code (closure) that executes the ANT. The second points to the result that was returned by the closure when the ANT was first executed. ANTFlag is used to implement fallback-mode: if the flag is set to True, then ANTs are enabled and logging is done using the ANTLogEntries list. If the flag is set to False, then ANTs are disabled and all logging is done to the TLogEntries list.

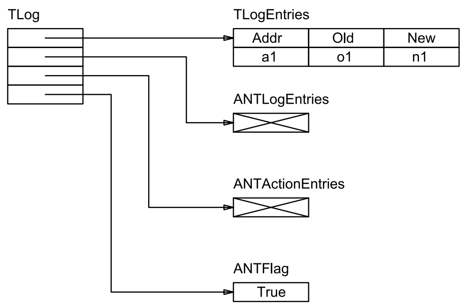
We will show, on a small example, how these logs are used. The following example is written in Java-like pseudo-code:

```
1. atomic {
2.   a1 = <LARGE COMPUTATION>;
3.   r2 = ant { <SMALL COMPUTATION>; }
4.   <LARGE COMPUTATION>;
5.   r3 = ant { <SMALL COMPUTATION>; }
6.   <LARGE COMPUTATION>;
7. }
```

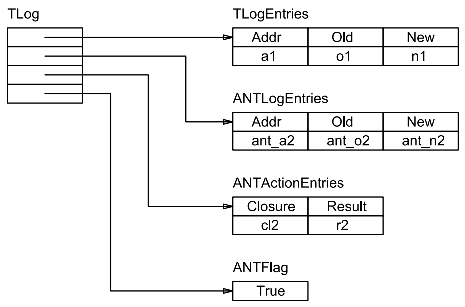
In Figure 8, we can see the structure of the TLog at different stages during the execution of the atomic block. The atomic block starts a transaction by creating an empty TLog (after executing line 1). TLogEntries, ANTLogEntries and ANTActionEntries are empty in the beginning (Figure 8(a)). Line 2 modifies variable `a1` with a tentative change to the TLogEntries (Figure 8(b)). In line 3, the ANT uses the ANTLogEntries for its tentative update: the ANT modifies variable `a2` and the change is logged in the ANTLogEntries. After the execution of the ANT, the pointer to the ANT's code and its result are saved in the ANTActionEntries (Figure 8(c)). Line 4 reverts to using TLogEntries for logging. In the example case, no access to the other transaction variables occured. The ANT in the line 5 changes the variable `a3` and uses the same the ANTLogEntries slot that was used by the ANT in line 3. The return value and the closure are once again saved in ANTActionEntries.



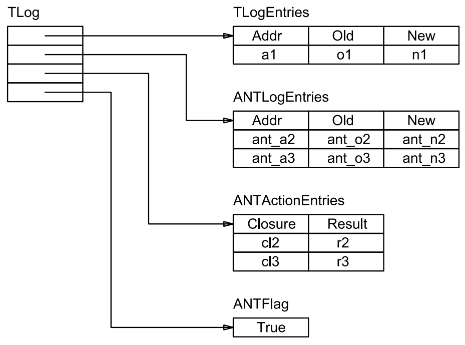
(a) A transaction starts with an empty transaction log.



(b) The transaction writes address `a1`.



(c) The first ANT finishes, having made updates to address `ant_a2` and returned result `r2`.

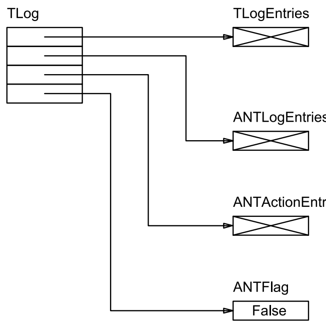


(d) The second ANT finishes.

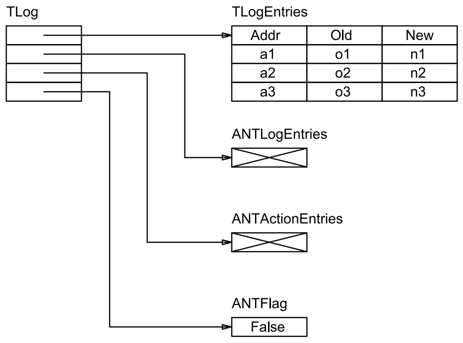
Figure 8. Transaction execution with ANTs enabled.

4.2 Changes when committing an `atomic` block

Ordinarily, at the end of an `atomic` block, the GHC RTS implementation of STM validates the transaction log and commits the updates to memory. The following algorithm explains how we modify the commit phase of a transaction:



a) Transaction re-executes



b) Re-executed transaction finishes

Figure 9. Transaction re-execution with ANTs disabled (ANTFlag set to False).

```
start:
  case validate(TLog):
    OK : commit TLog
    ANTLogEntries and TLogEntries intersect:
      set ANTFlag = False
      restart transaction
    ANTLogEntries invalid, TLogEntries valid:
      re-execute ANTActionEntries
      goto start
    TLogEntries invalid:
      restart transaction
```

This algorithm starts by validating the entire TLog, comprising the entries in TLogEntries and ANTLogEntries. There are four cases to consider:

1. *TLogEntries and ANTLogEntries are all valid.* This means that there have been no conflicts at all: not with the ANTs or with the remainder of the `atomic` block. In this case we commit all the log entries to memory.
2. *TLogEntries and ANTLogEntries intersect.* This occurs when a program uses the same transactional variable in the ANT and outside of it. The whole atomic block has to be re-executed with the ANTFlag set to False,

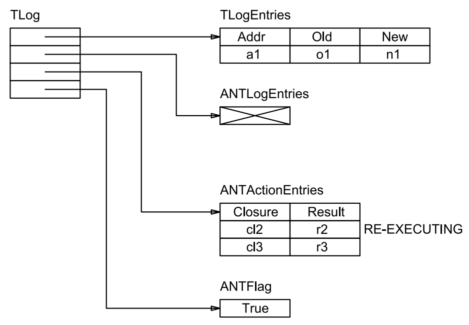


Figure 10. Transaction about to re-execute ANTs.

disabling ANT usage. There is no scalability gained from the ANTs, but the semantics of the atomic section are preserved.

Figure 9 shows what happens in our example. The transaction is restarted (Figure 9(a)) and builds up a single log during its re-execution (Figure 9(b)).

3. *ANTLogEntries are invalid and TLogEntries are valid.* This shows that there has been a conflicting memory access on one or more of the ANTs. This is the case when we can re-execute all the closures that are in the ANTActionEntries. After re-execution of every closure, the new return values from the closures are compared with the return values of the previous execution. If all the values are the *same* (implementation of equality is explained in Section 4.3), we know that any computation in the atomic block that depends on the those values will be unaffected by the re-execution of the closures. If any of the return values has changed then the atomic block has to be re-executed.

   Figure 10 shows what happens in our example. The ANTLogEntries structure is emptied and each ANT from the ANTActionEntries is re-executed in turn. After successful re-execution of these closures the TLog will once again look as shown on Figure 8(d).

4. *TLogEntries are invalid.* In this case there has been a conflict with the main transaction: we must re-execute the whole `atomic` block.

4.3 Implementing equality in RTS

There is a subtle problem in how we implement the equality test between different executions of an ANT. There are two factors to consider. First, for safety, we must err on the side of caution: two results can be claimed distinct when in fact they are equivalent. Second, our design principle that ANTs are semantically transparent means that the implementation of the equality test must not change the semantics of the `atomic` block. This means that we cannot generally use programmer-supplied equality tests unless we wish to trust these to be correct: in C# or Java terminology we would use '`==`' rather than '`.Equals`'.

However, working in Haskell raises another problem: the language uses lazy evaluation and so the result from an ANT may be returned as an un-evaluated closure rather than as a result which can be compared. We cannot simply evaluate the closure in case it is a non-terminating computation that is not needed by the program.

In practice we have not seen closures occurring in this way and so our prototype conservatively uses (*i*) pointer equality between objects with identity (e.g. mutable variables whose addresses can be compared), (*ii*) a shallow comparison function between objects without identity (e.g. boxed integer values). However, in a full implementation we could perform equality tests between a first result `R1` and a second result `R2` as follows:

- If `R1` and `R2` are both objects with identity then use pointer equality.
- If `R1` and `R2` are both objects without identity then recursively compare their constructor tags and fields.
- If `R1` has been evaluated but `R2` has not, then evaluate `R2` and repeat the comparison. This deals with the case where the `atomic` block has forced `R1` to be evaluated and may therefore depend on its result. If `R2` does not complete evaluation promptly then abandon it and re-execute the atomic block in fallback-mode.
- If `R1` has not been evaluated *then* `R1` *and* `R2` *can be treated as equal*. The key insight is that if `R1` was not evaluated then the `atomic` block cannot depend on the (still-unknown) value it may yield.

  However, there is one further caveat in this case: the atomic block may itself return `R1` or store it into shared memory when it commits. In this case we must replace `R1` with `R2` so that, if it is ever evaluated, the commit-time result `R2` is obtained. This can be done in the GHC RTS by atomically overwriting `R1` with an indirection to `R2`. Our earlier paper provides an introduction to the management of closures, indirections, and so on in the RTS [6].
- In other cases treat `R1` and `R2` as distinct.

5. Results

To explore the performance of ANTs we used a synthetic test program with the following structure:

```
atomic {
  v = ant { <SMALL COMPUTATION>; } // A1
  <LARGE COMPUTATION>; // L1
  ant { <SMALL COMPUTATION>; } // A2
}
```

Our test lets us vary the amount of time spent inside the ANTs by varying the amount of work performed in the large computation `L1`. As we discussed in Section 3.3, ANTs should be used for small parts of the transaction the are likely to conflict with other transactions and so we need to quantify what this means.

In our test program the ANT `A1` removes an item from a shared list of key-value pairs and `A2` returns an item to the head of the list. Each thread works on disjoint keys: concurrent invocations will always conflict in their reads and writes to the list, but the operations themselves are commutative. For the `<LAGRE COMPUTATION>`, we uses a simple function performing a private loop of fixed duration. We vary the number of threads operating on the list and the ratio of the large computation's execution time to the ANTs'.

Our test machine ran Windows Server 2003, with 2 quad-core CPUs (in total 8 cores) and 4GB of RAM. All the tests were compiled with optimizations and ran with GHC's heap configured to 512MB of heap so that garbage collection did not play any role in the execution times.

Figure 11, shows the execution time of the test program with 4-thread and 8-thread runs as the size of the large computation is varied. The graph compares the performance of the test using ANTs ('AN Transaction') against the performance with ANTs disabled ('Regular Transaction'). The x-axis shows the 'relative workload' which is the fraction of execution time spent *outside* ANTs.

As we discussed in Section 3.3 we would expect this to be high in the intended uses of ANTs. The implementation using ANTs out-performs the existing implementation when ANTs account for 70% or more of the execution time. The reason for this is that the re-execution time of regular transaction is larger than re-execution time of the transaction with ANTs; the regular transaction has to re-execute the whole `atomic` section, and on the other hand, the transaction with ANTs has to re-execute just ANTs. Of course, by making the size of the ANTs increasingly small, the performance difference could be made arbitrarily good. However, the important result is to understand the kind of range below which ANTs are ineffective.

Conversely, when most time is spent *outside* ANTs, we can see that ANTs slow down the program; for small `atomic` sections, the slowdown can be around 2x. This is because most of the execution time of the transaction is spent in `stmCommitTransaction()` and our prototype effectively introduces two passes over the log entries; one to distinguish the 4 cases in Section 4.2 and another to actually commit the changes to memory. In principle these operations could be combined.

In Figure 12 we compare the performance of our test program using ANTs with the same test program using ONTs in the case where around 8% of time is spent inside the ANTs. Both ANTs and ONTs improve scalability compared with executing all the operations in a single transaction. One would expect ONTs to out-perform ANTs in this workload: no compensating actions are run and so the work performed by ONTs is strictly less than ANTs. This is true aside from

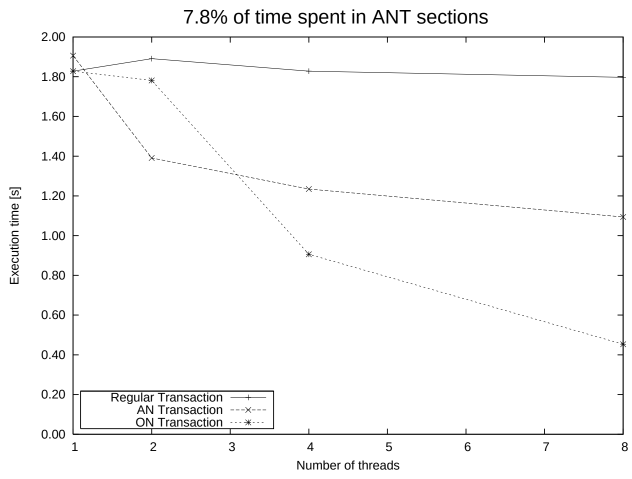


Figure 12. Execution times of regular transactions, ANTs and ONTs.

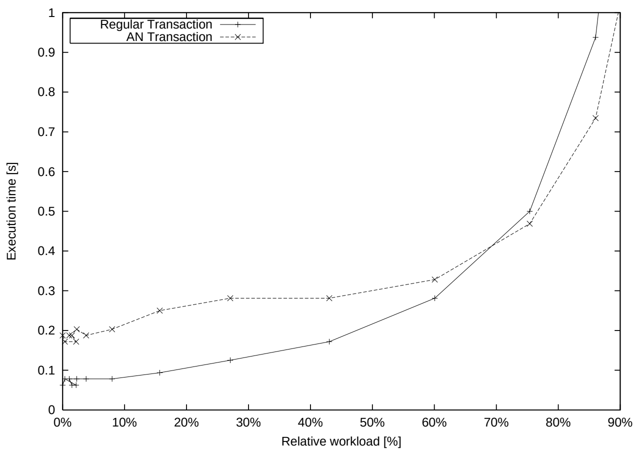
some jitter when running with 2 threads; we are not yet sure what causes this anomaly.

6. Conclusion

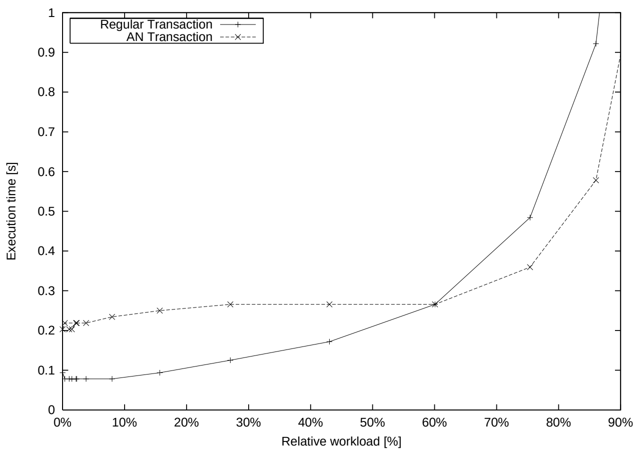
This paper has introduced the idea of *abstract nested transactions* (ANTs) for identifying sections of an `atomic` block that are likely to be the victims of benign conflicts. By re-executing ANTs we can avoid re-executing the whole `atomic` block that contains them. Unlike other techniques for improving the scalability of `atomic` blocks ANTs are semantically transparent and can be used as a performance-tuning technique without risk of changing the semantics or serializability of the code in which they are used.

This is not to say that they provide a replacement for other abstractions such as low-level unsafe optimization interfaces or open nested transactions. However, the unique features of ANTs open the possibility for completely automatic feedback-directed optimization of transactional programs to try to identify contention hot-spots.

We are currently evaluating ANTs in a number of larger STM-Haskell programs, reducing the overhead introduced by their implementation, and expanding the number of cases that can be handled without using fallback-mode. In particular, the test for non-intersection between the logs currently imposes extra work at commit time and excludes uses of ANTs where results pass from an ANT to the enclosing atomic block through shared variables. We are trying to address both of these issues. First, information about overlaps between the logs can be maintained during a transaction's execution (possibly switching to fallback mode eagerly, without re-execution, if an overlap occurs). Second, we believe that overlaps can be tolerated in cases where information flows from ANTs to the enclosing atomic block: we can identify these 'key outputs' and, on re-execution, subject them to the same test that we currently use for the ANTs' return values.



(a) 4 threads



(b) 8 threads

Figure 11. The execution time of the program plotted while varying the relative workload using a list of 32 elements. A low relative workload means that most of the test program is spent *inside* ANTs. A high relative workload means that most of the test program is spent *outside* ANTs.

Acknowledgments

We would like to thank Virendra Marathe, Simon Marlow, Simon Peyton Jones and Satnam Singh for their help and advice during this work, Jim Larus, Enrique Vallejo Gutierrez and Osman Unsal for their comments on earlier drafts of this paper, David Callahan for his suggestions for future work, and the anonymous reviewers for their advice.

References

[1] BERENSON, H., BERNSTEIN, P., GRAY, J., MELTON, J., O'NEIL, E., AND O'NEIL, P. A critique of ANSI SQL isolation levels. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data* pp. 1–10.

[2] CACHOPO, J., AND RITO-SILVA, A. Versioned boxes as the basis for memory transactions. *Sci. Comput. Program. 63*, 2 (2006), 172–185.

[3] CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. *Introduction to Algorithms*. MIT Press, 1990.

[4] FRASER, K. *Practical lock freedom*. PhD thesis, Computer Laboratory, University of Cambridge, 2003.

[5] HARRIS, T., HERLIHY, M., MARLOW, S., AND PEYTON-JONES, S. Composable memory transactions. In *Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming, to appear* (June 2005).

[6] HARRIS, T., MARLOW, S., AND JONES, S. P. Haskell on a shared-memory multiprocessor. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell* (Sept. 2005), pp. 49–61.

[7] HARRIS, T., PLESKO, M., SHINNAR, A., AND TARDITI, D. Optimizing memory transactions. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation* pp. 14–25.

[8] HARRIS, T. L. A pragmatic implementation of non-blocking linked lists. In *Proceedings of the 15th International Symposium on Distributed Computing* (Oct. 2001), vol. 2180 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 300–314.

[9] HERLIHY, M., LUCHANGCO, V., MOIR, M., AND SCHERER, III, W. N. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual ACM Symposium on Principles of distributed computing* (2003), pp. 92–101.

[10] LARUS, J. R., AND RAJWAR, R. *Transactional memory*. Morgan & Claypool, 2006.

[11] MICHAEL, M. M. High performance dynamic lock-free hash tables and list-based sets. In *SPAA '02: Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures* pp. 73–82.

[12] MOORE, K. E., BOBBA, J., MORAVAN, M. J., HILL, M. D., AND WOOD, D. A. Logtm: Log-based transactional memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*. Feb 2006, pp. 254–265.

[13] NI, Y., MENON, V. S., ADL-TABATABAI, A.-R., HOSKING, A. L., HUDSON, R. L., MOSS, J. E. B., SAHA, B., AND SHPEISMAN, T. Open nesting in software transactional memory. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*. pp. 68–78.

[14] SCHERER III, W. N. *Synchronization and concurrency in user-level software systems*. PhD thesis, University of Rochester, 2006.

[15] ZILLES, C., AND RAJWAR, R. Transactional memory and the birthday paradox. In *19th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)* (June 2007).