VCCTM Benchmarks

Part I

Introduction

Chapter 1
Introduction

1.1 Introduction to Transaction Verification

We present a static verification approach for programs running under snapshot isolation (SI) and similar relaxed transactional semantics. In a common pattern in distributed and concurrent programs, transactions each read a large portion of shared data, perform local computation, and then modify a small portion of the shared data. Requiring conflict serializability in this scenario results in serial execution of transactions or worse, and performance suffers. To avoid such per- formance problems, relaxed conflict detection schemes such as snapshot isolation (SI) are used widely. Under SI, transactions are no longer guaranteed to be serializable, and the simplicity of reasoning sequentially within a transaction is lost. We present an approach for statically verifying properties of transactional programs operating under SI. Differently from earlier work, we handle transactional programs even when they are designed not to be serializable.

1.2 Introduction to VCCTM Benchmarks

VCCTM is a static verification technique for C programs providing concurrency with transactions. To test VCCTM we used different TM implementations in terms of relaxation techniques that they include. In this document, we explain how VCCTM is applied to applications written with transaction diverging relaxation techniques. We expain briefly what the algorithm of each benchmark is, then explain what kind of transactions are running this algorithm, what correct execution is for a transaction to be completed, what is the proof argument that has to be encoded and be verified

by VCCTM.

Part II

Applications in Benchmark

Chapter 2
Labyrinth

2.1 Algorithm

Simply this is the algorithm for joining two points on three dimensional array. The algorithm guarantees finding shortest interconnection between two points. This routing algorithm connects two source grid cell to the target grid cell in two phases: expansion and backtracking. Expansion performs breadth-first search from the source grid until the target cell is located. During the grid search, each cell is checked for not being occupied and numbered according to its distance from source grid cell. Occupied cells can not be crossed directly, and routing must get round the occupied cell.

Backtracking starts when expansion phase achieves to locate the target grid cell. Backtracking starts at the target grid cell and iteratively finds a neighboring grid cell with a lower number than its own and occupies it, until it reaches the source grid cell.

There is a workqueue which includes pair of cells as a route to be completed by workers. When the workqueue has no pair of cells to be completed then the program terminates.

It is obvious that there is transactional nature inside this application. Each route can be treated as independent transaction. Any of grid cell routed by a transaction is commited concurrently by another transaction then routing transaction is not successful. Lets see the transactional model of this algorithm more in detail.

2.2 Transactional Model

Transactional model of the algorithm is running transactionally as the following: Each transaction takes a snapshot of the global grid and starts its routing operation. When route is found then it wants to commit its updates to the global grid. The only thing that should prevent this is another route having committed and used a common grid point. The following psuedo code shows how the transactiona labyrinth code looks like.

```
Grid global;

forall routes {
 atomic{
  Grid local;
  Expand from source to the target;// Read from global, write to the local
  Backtrack from destination to the source; //Read from local, write to global
 }

}
```

To say from TM semantic model(Snapshot Isolation SI), we can just take the intersection of write-sets in this transactional model because anything in the read set is also in the write-set so checking the write-write conflicts.

2.3 Correctness Argument

In this section we formally state that correctness argument for Labyrinth example which we mention informally above. Then dive into the interleavings that forms violation of TM model (SI) and successful execution ot TM model (SI).

A transaction running for finding route which has expansion(shortest-path) and bactracking phase(adding-path-to-pathlist) is successful as long as a transaction's expansion phase is not written by another transactions bactracking phase. As a result, points read for shortest path running by transaction T are updated in on grid after T's adding-path-to-list where path is formed by the points read during finding shortest path.

Assume that we have a function shortestPath for expansion phase which takes destion-ation point, dst, and source point, src, as parameters. This function guarantees that path returned from shortestPath function is connecting (IsConnectingPath(src,dst)) src to dst, and all points forming this path is on path (IsValidPath) and lastly there is no overlap which means all points on path is valid, inside grid borders, and values of these points on grid are equal to the path's unique ID.

Second pahse. addPathToList funciton for backtracking, It is guaranteed that old paths before this path is added are preserved on grid, new path is added and all points' values read in shortestPath function are updated as path's unique ID. One last post condition guaranteed is all other points on grid do not changed.

2.4 Applying VCCTM to transactional Labyrinth

Applying our technique to Labyrinth is as the following:

1. First we take the C program's source code written for a model which is SI in our example. Convert every pointer to an address into Pointer type and primitive integer type into PInt type which are transaction aware types.⟨hlink | | http://msrc.ku.edu.tr/wp-content/media/tmlbr.c⟩
2. Convert every read and write operation with VCCTM's transaction aware trans_read and trans_write which are the functions. These functions have pre and post conditions that have to hold (all these are converted into assume and assert statements later on by VCC) in the execution context that it is being called.
3. Convert every construct that start execution enviroment of the model that program is running on which is atomic in our case, into the trans_begin.
4. Insert TM runtime encoding into the program's transformed source code (steps above) so that invariants of PInt and Pointer typed variables and the functions that are implemeted with trans_read and trans_write are checked under admissibility(see in paper) constraint.⟨hlink||http://msrc.ku.edu.tr/wp-content/media/tmlbr.c⟩
5. If any violation of invariant admissibility and assertion do not occur then the program running under SI is verified.

Lets see an example of transformation:

```
void addToPathList(Grid *grid, pathlist_t* pathlist, path_t* path)
is transformed into

void addToPathList(PTrans ptrans,
          Pointer globalGrid,
          Pointer globalPathList,
          PTM tm,
              path_t* pathToAdd _(ghost claim c))
```

You see Grid* grid and pathlist_t* pathlist are the globals that are updated transaction that are running under SI model. path_t* path variable's value is computed locally so it is transaction local variable. We transformed the globals into transaction aware Pointers globalGrid and globalPathList. Local path_t* pathToAdd variable is not effected by transformation.

Basically, the reason why we are using claim object is just ha

```
void RouteFind(Pointer globalGrid,
     Pointer globalPathList,
     PTM tm,
     int xsrc,int ysrc,int zsrc, int xdst , int ydst , int zdst
     _(ghost claim c))

PRE-POST CONDITIONS // ⟨hlink||http://msrc.ku.edu.tr/wp-content/media/tmlbr.c⟩
{
   path_t* foundPath; int i, j;
   PTrans ptrans = (PTrans)malloc(sizeof(Trans)); _(assume ptrans)
   //create transactional context See 1.4.3
   trans_begin(ptrans, tm _(ghost c));

int ***localGridSnapshot = trans_relaxed_read_grid(ptrans, globalGrid, tm
                              _(ghost c));
// See 1.3 paragraph 3
foundPath = shortestPath(localGridSnapshot, xsrc, ysrc, zsrc, xdst, ydst,
                         zdst);
// assert desired path properties See 1.3 paragraph 3
_(assert IsConnectingPath(xsrc, ysrc, zsrc, xdst, ydst, zdst) &&
                         isValidPath(localGridSnapshot, foundPath) &&
                         NoOverLap(localGridSnapshot, foundPath))

// acquire lock of the grid : Just to mimic that we are the only owners of the
// global variables are ptrans. Check the invariants of this in TM runtime
// 1.4.4 ref
Acquire(globalGrid->pointerLock, ptrans _(ghost c));
Acquire(globalPathList->pointerLock , ptrans _(ghost c));
// update grid and pathlist in transcation See 1.4.2
trans_write_ptr(ptrans, globalGrid, tm, localGridSnapshot _(ghost c));
trans_write_addToList(ptrans, globalGrid, globalPathList, tm, foundPath
                      _(ghost c));
// commit to TM : Update global concurrently accessed(volatile) value and
// version number maps atomically.
_(ghost_atomic tm, ptrans, c {
      _(ghost tm->val = (lambda PInt i; ptrans->lockedWritesInteger[i] ?
                         i->data : tm->val[i]))
      _(ghost tm->valP = (lambda Pointer pr; ptrans->lockedWritesPointer[pr]?
                         (pr->ptr):tm->valP[pr])))
      _(ghost tm->version_num= (lambda PInt pr;
                               ptrans->lockedWritesInteger[pr] ?
                               pr->version_num_int : tm->version_num[pr])))
      _(ghost tm->version_numP= (lambda  Pointer pr;
                                 ptrans->lockedWritesPointer[pr] ?
                                 pr->version_num_ptr : tm->version_numP[pr]))
// assert desired invariants and method post-condition See 1.3 paragraph 4
//deallocation of transactional execution context
//smoke test :)
_(assert false)
}
```

Chapter 3
Sorted Linked List

3.1 Algorithm

The algorithm is basically as the following: In order to insert a node into a sorted linked list , first traversing the list to find the proper place to preserve being sorted and insert the node by updating the next field of last node whose value is less then the value of the node to be inserted.

3.2 Transactional Model

Simply what a transaction is as an execution unit in this as example is as the following: Assume that we have so many transactions each wants to update a global linked list concurrently with preserving it being sorted. Assume that we have a transaction that wants to insert a node to the sorted linked list. In order to do that transaction includes two phases: First it reads the nodes, traverse the linked list, finds the proper place which comes from being sorted. Second phase, after finding the proper place then transaction updates the global linked list by inserting the node.

In the first phase, a transaction reads from the global linked list and fills its read set until it finds the node.

In the second phase, a transaction writes to the global linked list. It flushes the local update (its write set) to the global linked list.

In our sorted linked list example our transactional model ignores write-after-read conflicts (!WAR) which can be forme by the following interleaving:

i. Transaction X and Y can cocurrently read the global sorted linked list.

ii. First, transaction X finds the proper place and inserts the node into the global linked list (flushes its write set). In !WAR, it is suitable to write to the nodes in transaction Y's read set for updating list as long as any write-after-write(WAW) conflict does not occur.

3.3 Correctness Argument

The correctness argument of the linked list is being sorted and all nodes are reachable from head. Lets see how correctness argument is inserted in between and after the phases.

The first phase which is traversing the list for the proper place , findNodeToInsert function, includes a loop that has a condition which states that if the node being read is bigger than the value of new node to be inserted then break the loop due we find the node whose ownerships should be explicitly given to the transaction that is running.

The second phase, which is updating global sorted linked list has the pre condition of having ownership of the nodes on which the update occurs. It does the update on linked list and assert the correctness invariant in first paragraph on the updated linked list.

3.4 Applying VCCTM to transactional Sorted Linked List

Procedure in 1.4(1-5) is followed. Lets see the code transformation in detail.

1. Insertion of the specifications of the objects into strcuts.

   ```
   typedef struct Node { struct Node *next; int value; } Node;
   typedef struct List { Node* head} List;
   ```

   is transformed into the following with given specification.

   ```
   typedef _(dynamic_owns) struct List { Node *head;
    _(ghost int size;)
    _(ghost bool val[int];)// the int value exists in list or not
                           // val[Node->value]
    _(invariant head== NULL ==> size>=0)
    _(invariant head!=NULL ==> size>0)
    _(invariant forall Node *n; mine(n) && n->next ==> mine(n->next))
    ...
    }
   ```

   Here _(ghost) variables and mine constructs are auxiliary variables to help proving the reachabiltiy of nodes in the linked list. We add them as auxiliary functions in the specification for proving linked data structures.

2. Inserting pre-post conditions of the functions that belongs to the data structures. These are the specifications of the functions given by the programmer.

   ```
   // adds a newNode to the sorted list and maintains the order int
   int list_addNode(List *list, Node* newNode)
     _(requires wrapped(list))
     _(requires wrapped(newNode))
     _(requires (newNode \in list->owns) == \false)
     ...
     _(ensures \result == 0 ==> (forall int p; list->val[p] <==>
                  (\old(list->val[p]) || p == newNode->value ))
     _(ensures \result == 0 ==> list->find == (\lambda int p;
                  p == newNode->value ? newNode : list->find[p]))
     _(ensures \result == 0 ==> list->size == \old(list->size) +1 ))
    ...
   ```

   is the specification added function declarations which is part of code transformation under programmer control. owns construct is a set which holds the objects that are included by mine construct.

3. The next step is inlining the TM runtime specification which include transactional reads, writes, transaction aware types PInt, Pointer and TM runtime encoding.⟨hlink | |http://msrc.ku.edu.tr/wp-content/media/tm.c⟩

4. All integers and any type of address pointer that are shared concurrently accessed are converted to Pointer and PInt respectively.

   ```
   // take prev and current nodes whose locks are already acquired //
   in the same trancation (that the locks are acquired) insert new node
   between prev and current
   void list_addNode(PTM tm, PTrans ptrans,
                   Pointer prev,
                   Pointer newNode,
                   Pointer current
   ```

```
                    _(ghost claim c)
                    _(ghost claim current_ok))
```

The type transformation for function bodies can be seen ⟨hlink | | http://msrc.ku.edu.tr/wp-content/media/tm.c⟩. After doing type transformation, we handle the state of heap as giving the heap variable as a parameter to the function and state as a pre condition.

5. After doing type transformation, transactional contexts inside the functions are formed (trans_begin) and concurrent read and write operations inside these execution contexts are transformed (trans_write, trans_read) and the committing the local updates into the global memory, commit phase. You can see the body of the function before transformation in ⟨hlink| | http://msrc.ku.edu.tr/wp-content/media/seq_list.c⟩ and after transformation in ⟨hlink | | http://msrc.ku.edu.tr/wp-content/media/tmlnk.c⟩. However, It is useful to dive into details commit phase.

   i. After the read phase, it is ensured that the ownership of the nodes that are going to be updated are taken by the current transaction. End of the loop in the first phase, the adress of the node to be written (prev node) and to be inserted (newNode) are explicitly owned by the ptrans.

```
Acquire(prev->pointerLock, ptrans (ghost c))
Acquire(newNode->pointerLock, ptrans (ghost c))
```

   ii. The local update on the snapshot is done by the trans_write functions.

```
// Set: newNode->next == current
trans_write(ptrans, newNodel, tm, current, _(ghost c))
// Set: prev->next == newNode
trans_write(ptrans, prev, tm, newNode, _(ghost c))
```

   iii. Atomic update of global maps.

```
_(ghost_atomic tm, ptrans, c {
    _(ghost tm->val = (lambda PInt i;
                      ptrans->lcokedWritesInteger[i] ?
                      i->data : tm->val[i]))
    _(ghost tm->version_num = (lambda PInt pr;
                               ptrans->lockedWritesInteger[pr]
                               ? pr->version_num_int :
                                 tm->version_num[pr]))
...
```

   iv. Assertion of desired invariants. (See 2.3 paragraph 1)

   v. Deallocaltion of the transactional context

Part III

How to run benchmarks

Chapter 4
VCC Installation

4.1 Getting VCC installed

⟨hlink||http://vcc.codeplex.com/wikipage?title=Install&referringTitle=Home⟩

4.2 Downloading benchmarks

You can download benchmarks from the ⟨hlink | | http://msrc.ku.edu.tr/projects/vcctm/⟩. Sequential and TM version of the benchmarks can be found in this link.

4.3 Automatic code transformation

A tool which performs code transformation explained 1.4 and 2.4 will be implemented soon. Specification of objects and function predicates taken as an input from the user and code is transformed with this given input.

Will Be Available Soon !