Taming TSO Memory Consistency with Models

Michael N. Christoff
University of Toronto
27 King's College Circle
Toronto, Ontario, Canada
christoff@cs.toronto.edu

ABSTRACT
From cell phones to desktops, the average number of CPUs found in new computer systems continues to grow. With this growth comes the need for multi-threaded shared memory algorithms that can exploit the many CPUs found in these systems. However, in systems with TSO memory consistency, there exists an unknown delay between when a thread executes a 'write' instruction, and when the value of that write is actually written to shared memory (where it can be read by other threads). This can make reasoning about communication between threads difficult, since it can be complicated to determine when the messages a thread writes will become visible to other threads. In this paper we describe the TSO Helper system in which the sequence of reads and writes made by a multi-threaded algorithm can be visually modeled. TSO Helper can then transform this model of reads and writes into one that visually illustrates the points at which each write instruction executed by a thread is guaranteed to have become visible to other threads. With this knowledge, a developer can ensure her algorithm is not making decisions that assume previous writes are visible to other threads before these writes have actually been written to shared memory.

Categories and Subject Descriptors
D.2.2 [**Software Engineering**]: Design Tools and Techniques – c*omputer-aided software engineering (CASE).*

General Terms
Algorithms, Design, Reliability, Theory, Verification.

Keywords
TSO, Total Store Order, Modeling, Concurrency, Relaxed Memory Consistency, Shared Memory Multiprocessing, Distributed Computing.

1. INTRODUCTION

1.1 Overview
The current trend in computer systems design is to develop machines with increasingly large numbers of CPUs. In order to take advantage of this increase in computing power, multi-threaded algorithms are needed that can make use of all available CPUs. However—for performance reasons—many modern systems employ what is called TSO memory consistency. Under TSO memory consistency, the value of an executed write instruction is not immediately written to shared memory. Instead, there is a finite, nondeterministic delay between when the write instruction is executed, and when the value of the write is actually written to shared memory. When the value of a write instruction is actually written to shared memory, we say that the write has been ***committed***. Until a write is committed, other threads cannot read the value of the write. Because of the delay between when a write is executed and when that write is committed, a thread must be careful about making decisions that assume the values of its previous writes are visible to other threads, because if it is wrong it can leave the system in an undefined state. To be certain that the values of previous writes have been committed (and are hence visible to other threads), a thread must execute a ***memory barrier*** instruction. A memory barrier instruction guarantees that all write instructions executed before it are committed by the time the memory barrier instruction completes. However, memory barriers are expensive operations that circumvent many system performance optimizations.

This is where TSO Helper comes in. Using TSO Helper, an algorithm designer can create a visual model of the read, write, and memory barrier instructions that her algorithm executes. The model can also include nodes representing the functions that the algorithm calls, as well as synchronizations (explained later) in the algorithm. Given this model, TSO Helper generates a new model that contains the points at which writes made by the algorithm are guaranteed to have been committed. With this knowledge, the designer can ensure her algorithm is not making decisions that assume previous writes are visible to other threads before these writes have actually been committed.

1.2 Organization of the rest of the paper
In section 2 we begin by discussing the motivation behind the need for multi-threaded algorithms. We then delve into the details of TSO memory consistency. In section 3, we describe two algorithms that we will use to illustrate the use of TSO Helper. In section 4, we introduce TSO Helper and—by using it to analyze the algorithms of section 3—show how it can aid developers in writing high performance algorithms for systems with TSO memory consistency. In section 5 we outline the implementation of TSO Helper. In section 6 we conclude with future directions.

2. BACKGROUND AND MOTIVATION

2.1 The Need for Multi-threaded Algorithms
Up until the mid-2000s, the history of CPU performance had in large part, been due to ever-increasing CPU clock speeds. Faster CPU clock frequencies allow a processor to execute more instructions per second, and hence execute algorithms faster than older CPUs with slower clock frequencies. This constant increase in the number of instructions executed per second allowed programmers the luxury of writing sequential algorithms with the knowledge that these algorithms would see increased performance with each new generation of processor. However, due to physical and practical limitations, CPU clock frequency has remained stalled at between 3 to 4 GHz for the past 7 years. Despite this,

there remains an expectation that software performance will continue to improve over time.

As a response to this 'frequency barrier', the computer industry has seen a dramatic shift away from single CPU systems to the point where, today, consumer systems with 2 or more independent CPUs are now the norm rather than the exception. However, since the clock speeds of individual CPUs in these multi-CPU systems are also subject to the frequency barrier, continued increases in application performance requires that software developers code parallel and distributed algorithms that spread work across all available processors.

2.2 Sequential and TSO Memory Consistency

A system's memory consistency model determines the order in which the read and write instructions executed by a thread are committed to memory. The two main memory consistency models that exist today are ***sequential consistency*** (SC herein), and ***TSO*** (total store order***) consistency***.

TSO is of interest, since it is the native memory model of all modern x86 and x64 compatible CPUs (both from Intel and AMD).

2.2.1 Sequential Consistency

This is the simpler of the two consistency models. Under sequential consistency, when a write is executed, the processor waits until it is committed to memory before executing the following instruction. This makes proving the correctness of algorithms in a system with an SC memory model somewhat simpler than for algorithms running in a TSO system. This is due to the fact that one can be certain that the value of each write instruction an algorithm executes is visible to all other threads immediately after the instruction is executed.

2.2.2 TSO Consistency

Under TSO consistency, the value of write instructions are not immediately committed to memory. Instead, they are stored in a per-thread FIFO queue called the thread's **write buffer**. At some unknown, but finite time after the write is buffered, the CPU will dequeue each write in the buffer and commit it to memory. All writes are dequeued in the same order they were enqueued (i.e.: they are dequeued in FIFO order).

2.2.3 Why TSO?

The reason for buffering writes is to increase system performance. A thread cannot make forward progress if the read instruction it is currently executing is delayed, since the next step the thread needs to take may depend on the value returned from the read (we will ignore the possibility of speculative branching in this paper). For example: consider the statement `if(x < 0) doThis(); else doThat();`. Without being able to read the current value of x from main memory, the thread cannot know whether to execute doThis() or doThat(); However, a thread ***can*** make forward progress if its writes are buffered since its branching logic does not depend on the values it writes (except in one special case which we discuss below). Buffering writes can increase performance by allowing a thread to continue processing while cache lines are loaded from main memory, and by increasing system bus utilization. However, the technical details regarding why TSO increases system performance are outside the scope of this paper. Suffice it to say, write buffering helps multiprocessor systems run faster.

2.2.4 Write Buffer Bypassing

If a thread *writes* to a variable in shared memory, say x, and then subsequently *reads* from x ***before its write to x is committed***, then the value of x returned to it is ***the value of the last write to x in its write buffer***, not the value of x in main memory. This is referred to as write buffer bypassing, since main memory is bypassed in favour of a returning a value for the read from the write buffer.

2.2.5 TSO Operational Model

To make the last few sections more concrete, we provide here an operational model for TSO.

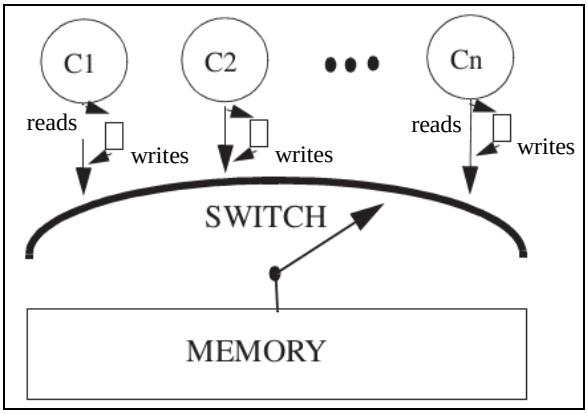

Figure 1. TSO operational model using a memory switch.

The model is composed of a set of threads C_i, a single switch, and memory, as depicted in figure 1. Assume that each thread presents memory operations to the switch one at a time in its program order (e.g.: in the order that its operations appear in its source code).

The Switch is Fair. The switch may select processes by any method that does not starve a process with either a waiting operation, or a non-empty write buffer.

Write. When a thread p executes a write operation of the form ***a = u***, where ***a*** is the location of a variable in main memory and ***u*** is a value, the tuple ***(a,u)*** is immediately enqueued in p's write buffer without p needing to wait to be selected by the switch.

Memory Barrier. If a thread p executes a memory barrier instruction, **it cannot execute any subsequent instructions until its write buffer is empty**. Once p is selected, the switch may dequeue and commit one or more buffered writes in p's write buffer before servicing another thread. Depending on the number of writes in p's write buffer, p may need to be selected multiple times before all of its buffered writes are dequeued and committed by the switch.

Read. Assume p executes a read operation from memory location ***a***. If p's write buffer contains any buffered write of the form ***(a,u)***, then p takes the value ***u*** of the most recently buffered write ***(a,u)*** as the value of ***a***. It can do this without having to wait to be selected by the switch. *(Note: This is write buffer bypassing as described in section 2.2.4)*

Otherwise, one of two things may occur:

1. If p's write buffer is non-empty, the switch may dequeue a write ***(a,u)*** from the buffer, and commit the operation ***a=u*** to shared memory.

2. If there are no tuples of the form ***(a,u)*** in p's write buffer, the switch returns the value of variable ***a*** from memory.

We note that the switch may—at any time—dequeue a write from p's write buffer. It is not restricted to doing so only when p executes a read operation or a memory barrier instruction.

2.2.6 SC from TSO

It is possible to simulate an SC memory model in a TSO system, simply by placing a memory barrier operation after each write in an algorithm's source code. This is what most high-level programming languages do in order to provide programmers the semantics of an SC system when running on Intel or AMD based machines. However, this imposes a performance hit on programs that could be reduced if unnecessary memory barriers can be detected and removed. TSO Helper can help a developer determine places where memory barriers may be unnecessary.

2.2.7 The Peculiarity of TSO

In the code of listing 1, we show an example of the non-intuitive behaviour that can occur in TSO systems. The listing consists of two short programs executed by two separate threads.

Variables x and y are ***shared*** amongst threads t1 and t2, while variables a and b are used ***only*** by thread t1 and thread t2, respectively.

`x = y = 0 // the values of x and y in main memory`	
t1	t2
1. x = **1** 2. a = y	3. y = 1 4. b = x

Listing 1. An example of write buffer bypassing.

Surprisingly, there exists an execution of these two programs after which a = 0, b = 0 while x = 1 and y = 1. We illustrate how this occurs in listing 2, where WB1 is thread t1's write buffer, and WB2 is t2's. The column 'Main Memory' contains the values of x, y, a, and b in main memory. Variables whose value do not change in a step are left blank.

			Main Memory					
	t1	**WB1**	**x**	**y**	**a**	**b**	**t2**	**WB2**
1		[ ]	**0**	**0**	**?**	**?**		[ ]
2	*x = 1*	*[(x,1)]*						[ ]
3		[(x,1)]					*y = 1*	*[(y,1)]*
4	*a = y*	*[(x,1)(a,0)]*						[(y,1)]
5		[(x,1)(a,0)]					*b = x*	*[(y,1),(b,0)]*
6		*[(a,0)]*	*1*					
7		*[ ]*			*0*			
8				*1*				*[(b,0)]*
9						*0*		*[ ]*
10			1	*1*	*0*	*0*		

Listing 2. Writes to x and y are buffered

In steps 2-4, thread t1 has its writes to x and a buffered, while thread t2 has it writes to y and b buffered. Then in steps 6 and 7, thread t1's write buffer is flushed. Note how the order that writes are dequeued and committed is the same order in which they entered. Similarly, in steps 8 and 9, thread t1's write buffer is emptied and committed in FIFO order. This execution results in x = 1, y = 1, while a = 0 and b = 0. This result is not possible for these programs if run in an SC system.

3. Motivating Examples

To help ground the preceding material and introduce TSO Helper, we provide the following concrete examples as an illustrative aid.

```
Shared variables:
    X_WANTS_TO_UPDATE = false
    Y_WANTS_TO_UPDATE = false
    acct = 0

Local variables:
    x_temp = 0   // local to thread x
    y_temp = 0   // local to thread y
```

thread X :	**thread Y :**
1. X_OTHER_BANKING_WORK();	8. Y_OTHER_BANKING_WORK();
2. X_WANTS_TO_UPDATE = **true**	9. Y_WANTS_TO_UPDATE = **true**
3. **await**(¬Y_WANTS_TO_UPDATE)	10. **await**(¬X_WANTS_TO_UPDATE)
4. *x_temp = acct + 1*	11. *y_temp = acct + 1*
5. *acct = x_temp*	12. *acct = y_temp*
6. X_WANTS_TO_UPDATE = **false**	13. Y_WANTS_TO_UPDATE = **false**
7. **goto 1**	14. **goto 8**

Listing 3. Simple two thread mutual exclusion algorithm. Note that this algorithm is *not* deadlock-free.

Listing 3 is code for a simple 2-thread distributed ***mutual exclusion*** algorithm, however, it is only correct under SC, not TSO. A mutual exclusion algorithm is one that serializes access to a shared variable. We will note that this algorithm is ***not*** deadlock-free under SC or TSO. However, deadlocks and deadlock-freedom are outside the scope of this paper so we shall ignore this fact.

In this contrived example, threads X and Y perform deposit updates on a single bank account, where the current value of the account is stored in the shared variable acct. We imagine that X and Y monitor two different ATM machines, and when an account owner deposits some money, either X or Y (depending on which ATM is used) will update the account. We also imagine that these ATMs only allow one dollar to be deposited at a time. If the account has two owners (for example, consider a married couple that jointly own the account), then we must ensure that their deposits are made one at a time. Otherwise, we can have the following situation:

acct = 0	
thread X **(account owner 1):**	**thread Y** **(account owner 2):**
1. x_temp = acct	
2	y_temp = acct
3. acct = x_temp + 1	
4.	acct = y_temp + 1

Listing 4. An interleaving where two $1 deposits to acct result in acct having an with incorrect value of $1.

This ordering of X's and Y's instructions is called an ***interleaving***. In this particular interleaving, we see that X and Y both read and increment the same account value, and then store the result back into acct. This results in the final value of acct being 1, rather than the correct value of 2.

To avoid this problem, we must ensure that only one thread has access to the shared variable acct at given point in time. In other words, we must ensure that access to the variable acct is mutually exclusive; i.e.: if thread X has access to acct, then thread Y does not, and vice. versa.

To ensure mutually exclusive access to acct, thread X first flags its intention to write to acct by setting X_WANTS_TO_UPDATE to true. It then checks to see if Y has flagged its intention to update acct. If Y has not flagged its intent to enter the account, then X updates it, and—after doing so—resets X_WANTS_TO_UPDATE to false. However, if Y has set Y_WANTS_TO_UPDATE to true (indicating Y wants to, or is already in the process of updating the account), then X waits until Y_WANTS_TO_UPDATE becomes false. Once Y_WANTS_TO_UPDATE becomes false, X performs its update.

The situation described above is symmetric for Y.

```
Shared variables

X[1],...,X[n] = DONE
SOMEONE_IS_WAITING = false

client thread i ∈ { 1,...,n} :

1.  OTHER_WORK(i);
2.  X[i] = WAITING
3.  SOMEONE_IS_WAITING = true
4.  await(X[i] == YOUR_TURN)
5.  processSharedVariables(i);
6.  X[i] = DONE
7.  goto 1


administrator thread :

8.  await(SOMEONE_IS_WAITING == true)
9.  SOMEONE_IS_WAITING = false
10. for(i in 1...n) :
11.       if(X[i] == WAITING) :
12.             X[i] = YOUR_TURN
13.             await(X[i] == DONE)
14. goto 7
```

Listing 5. A simple centralized mutual exclusion algorithm.

Listing 5 is a centralized mutual exclusion algorithm. We describe the entire algorithm only for completeness, as we will only focus on the 'client' code of the algorithm in our examples.

The algorithm is composed of a set of n > 1 'client' threads, and a single dedicated daemon thread called the 'administrator' that grants clients exclusive access to a set of shared variables (not shown here) that are processed in the function processSharedVariables(). Threads communicate with the administrator via elements in the n-element array X, and via the shared variable SOMEONE_IS_WAITING.

The algorithm works as follows. The administrator continuously reads the variable SOMEONE_IS_WAITING, waiting for it to become equal to true (line 8). When a client i needs to process the shared variables, it exits the OTHER_WORK() function (line 1). It then sets X[i] to WAITING (line 2), and, finally, sets SOMEONE_IS_WAITING to true (line 3). When the administrator sees SOMEONE_IS_WAITING is true, it resets SOMEONE_IS_WAITING to false (line 9), and then iterates through X (lines 10-13), looking for elements X[j] in X that are equal to WAITING (e.g.: it looks for a client that is waiting for access to the shared variables). Once the administrator finds such an element, X[i] for example, it grants thread i access to the shared variables by setting X[i] to the value YOUR_TURN (line 12), and then waits for X[i] to become equal to DONE (which indicates that thread i is done processing the shared variables). Since client i has been reading X[i] in a loop (line 3), it will eventually see X[i] == YOUR_TURN and execute processSharedVariables(i). When it is done, it sets X[i] to DONE and continues with its other work. After the administrator sees X[i] has been changed to DONE, it continues iterating through the array X looking for any other threads that want access to the shared variables.

4. TSO HELPER

To describe and illustrate TSO Helper, we will first use it to analyze the algorithm of listing 3.

4.1 Creating the Read/Write graph

The first step in using TSO Helper is to convert the algorithm one wishes analyzed into what we refer to as a read/write graph. To illustrate what a read/write is, we will now create one from the code for thread X in listing 3. To conserve space, we have renamed some of the functions and variables: X_OTHER_BANKING_WORK() is now X_OTHR_WRK() and X_WANTS_TO_UPDATE is now X_UPDT.

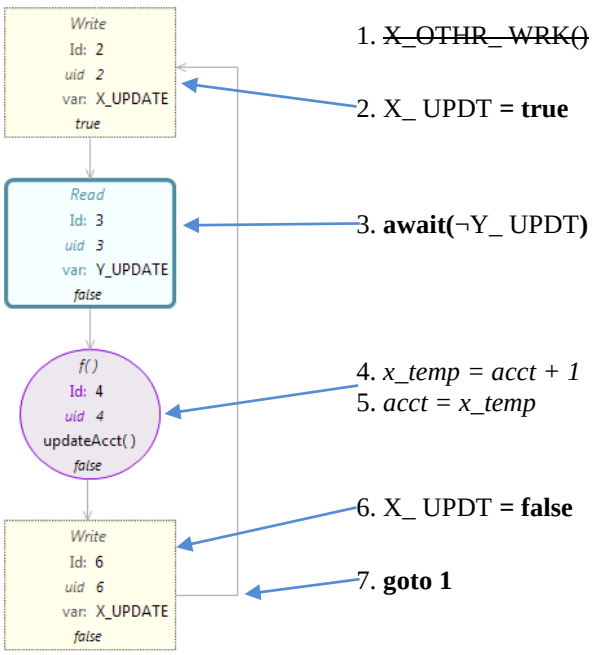

Figure 2. Read/Write graph for thread X of listing 3

4.1.1 First Attempt

For our first attempt, we have removed the call to the X_OTHR_WRK() function in line 1, for simplicity.

For the read/write graph, we begin by creating a Write node to model the write in line 2, and set its id value to 2. id's are arbitrary values, but must be unique in a read/write graph. We select the 2 for its id since the code associated with it appears on line 2. We also set this node's uid to 2. uids must also be unique within a read/write graph. Importantly, we set the bottom boolean attribute of the node to *true*. This tells TSO Helper that this node is the ***start node*** of the algorithm. Only one node in the read/write graph may be designated a start node.

Note that Read and Write nodes have a ***var*** attribute. This attribute is meant to hold information about the variable being written or read from, however its purpose is purely to make the read/write graph more readable for the developer, and is not materially used by TSO Helper.

To represent the busy-loop in line 3, we create a single Read node. For lines 4 and 5, we use a function node to represent the two lines. For the write in line 6, we use another Write node. Finally, we attach each of the nodes together with Transition links. Note that we have not added a Transition between the Read with id 3 and itself, even though this read is part of a busy-loop (line 3). This is because TSO Helper does not allow back-edges in the read/write graph, except to the start state. To represent a loop, users must simply manually expand 1 or more iterations of it. However, this simplification does not affect the correctness of the analysis TSO Helper makes.

4.1.2 The Function Node

While Read nodes, Write nodes, and Transitions are relatively self-explanatory, function nodes require marginally more explanation. A function node is used to represent a block of code to help simplify a read/write graph. The first step TSO Helper performs when it processes a read/write graph is to convert function nodes into plain Read and Writes nodes.

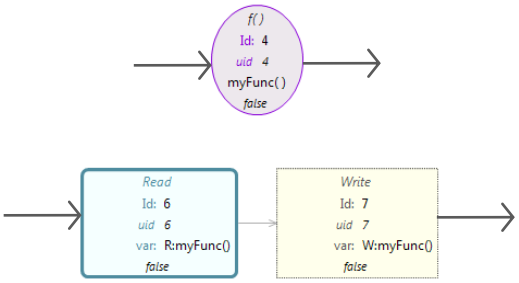

Figure 3. Function node converted to a Read followed by a Write node

In figure 3 we see a function node for the function myFunc() converted into a Read node R:myFunc() and a write node W:myFunc(). The Read node represents all read operations that may occur in myFunc(), while the node W:myFunc() represents all the writes that may occur.

4.1.3 Second Attempt

To further simplify this example, we will replace the function node with more descriptively named Read and Write nodes (see figure 4). So instead of the function node, we have instead a Read node followed by a Write node whose var attributes are set to accept. We do not need to worry about the x_temp variable of lines 4 and 5 since it is local to thread X (i.e.: it is not shared with thread Y).

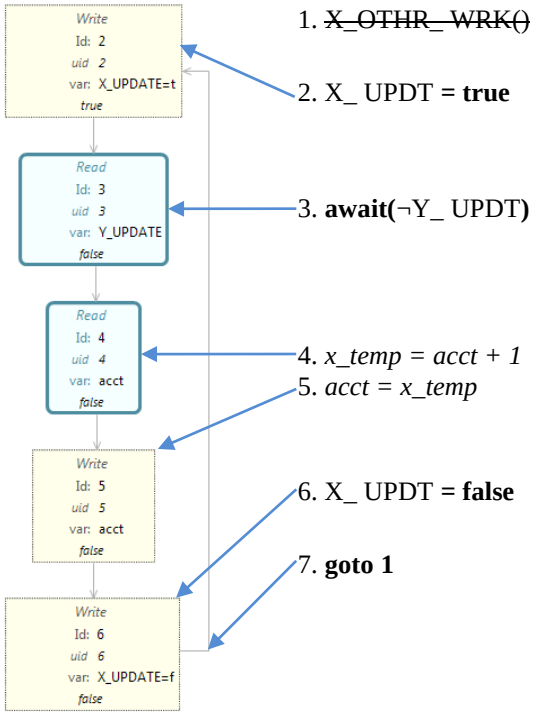

Figure 4. Second Version of Read/Write graph for thread X of listing 3

4.2 Running The Analysis

Read/Write graphs are designed using the ***TSO Helper Designer***, which uses the GMF framework.

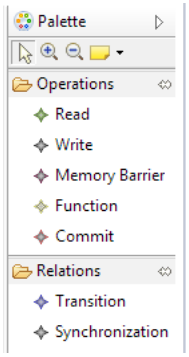

Figure 5. The TSO Helper Designer palette

Once a read/write graph is designed, it is stored as an XMI file with the extension ***tso***. To process a read/write graph, the developer selects the input graph, and selects a name for the output graph using the ***TSO Helper Control Panel*** (figure 6). Once the input and output file names have been set, the developer clicks the Run button to start the transformation.

4.3 A First Result

In this case, our transformation fails. In other words, TSO Helper concludes that there is no place in any execution of the algorithm where any write can be guaranteed to have been committed. However, this should be clear, since at no point in the algorithm

are any writes forced to commit. Put another way, if we consider the algorithm being executed in the 'switch' model of section 2.2.5, there is no way for us to predict when the switch may decide to dequeue any buffered write operation. So while we know the switch will ***eventually*** dequeue all buffered writes (since the switch is fair), we cannot—for any write—deduce any single point where the switch is guaranteed to have dequeued and committed any write.

This result means we will be unable to prove that the algorithm satisfies mutual exclusion. If we cannot make any assumptions about when our write in line 2 (X_UPDT = true) becomes visible to Y, we cannot update the account (lines 4-5), ***even*** if we have read that Y_UPDT is false in line 3.

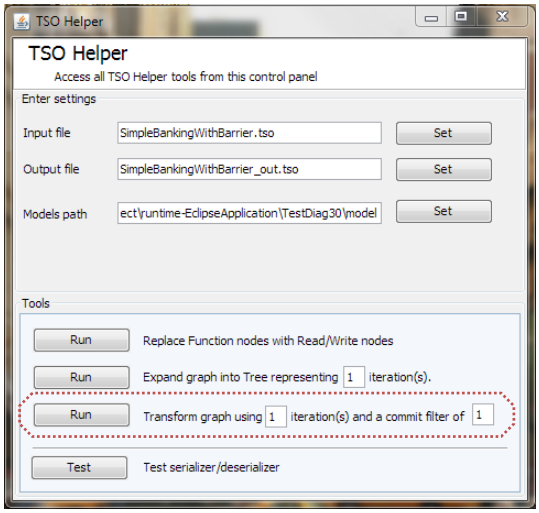

Figure 6. TSO Helper Control Panel. The enclosed section (surrounded by red dots) is where parameters for the full transformation are set.

4.4 Revising the Algorithm

A simple solution is to add memory barrier instructions after every write we make. Recall that—after a memory barrier instruction completes—we are guaranteed all our buffered writes have been committed. However, perhaps we can get away with fewer memory barrier instructions. As a first attempt at a revised algorithm, we will insert a memory barrier after the write in line 2 (figure 7). We will then process the new model and see if this change is enough to allow the algorithm to satisfy mutual exclusion.

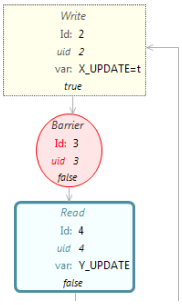

Figure 7. A Memory Barrier node is inserted after the write X_UPDT = true

4.5 Second Result

This time TSO Helper is able to successfully determine points in the execution of the algorithm where each write is guaranteed to have been committed. The output, called a ***commit graph***, is shown in figure 8.

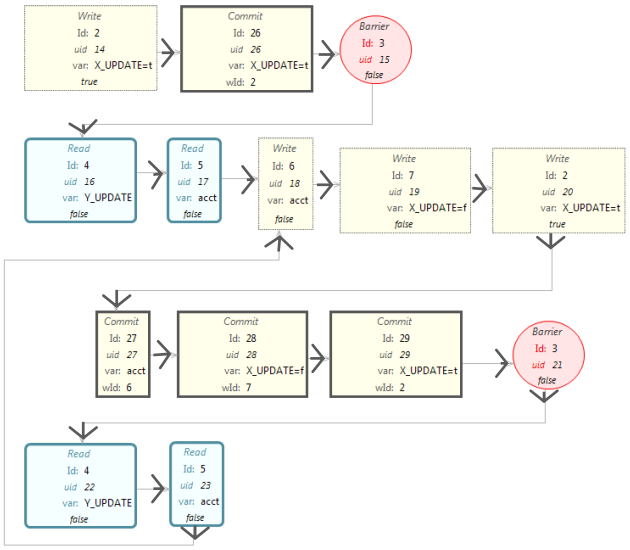

Figure 8. Write nodes with thick dark edges represent places where writes have been guaranteed to have been committed.

Recall that the write of line was given was represented by a Write node with id 2. In row 1 of the output graph (figure 8), we see that the Write node with id 2 is followed by a **Commit** node. A Commit node looks like a Read node but with darker edges. This particular Commit node has a **wId** (write id) attribute of 2. This means that this Commit node represents the commission of a write represented by a Write node with id 2. In our model, the Write node with id 2 is the write X_UPDT = true of line 2 of the algorithm. The way row 1 is to be interpreted is: After the execution of the memory barrier in iteration 1 of the algorithm, we are guaranteed that the write of line 2 (from iteration 1) has been committed.

In row 2 of the output graph, we see nodes representing the read of Y_UPDT (await(¬Y_UPDATE)), and the read of acct (x_temp = acct) of iteration 1. This is followed by the node representing the write to acct (acct = x_temp + 1) and the write to X_UPDT (X_UPDT = false) of iteration 1. The third Write node in row 2 represents the write X_UPDT = true of the ***next iteration*** of the algorithm.

Row 3 is composed three Commit nodes followed by the ***next*** execution of the memory barrier. The three Commit nodes—in order—represent the commission of writes acct = x_temp + 1 and X_UPDT = false, of iteration 1. The third Commit node represents the commission of the write X_UPDT = true of the ***next*** iteration. This behaviour is exactly what we expect: The first write (X_UPDT = true) is committed due to the barrier in the first iteration. The remaining writes of iteration 1, plus, the next write X_UPDT = true from the second iteration are committed due to the barrier in the next iteration. This pattern then repeats as is illustrated by the loop in the commit diagram.

4.6 Synchronization

To illustrate the concept of synchronization and how TSO Helper makes use of it, we will examine the algorithm of listing 5. It may be prudent to review the description of this algorithm now.

To understand synchronization, let us take first examine lines 2 and 4 of listing 5. In line 2, client i sets X[i] to WAITING. It does this so that when the admin thread scans the array X, it will see that X[i] == WAITING and know that client i wishes to obtain exclusive access to the shared variables. Then, in line 4, client i enters a tight loop where it reads X[i] and waits for the admin thread to change its value from WAITING to YOUR_TURN.

There are two important points to note here. First: Immediately before client i sets the X[i] to WAITING, the value of X[i] in shared memory is equal to DONE. Second, the admin thread will only set X[i] to YOUR_TURN ***if it sees that X[i] is equal to WAITING***. Therefore, if client i sees X[i] change from WAITING to YOUR_TURN, then its write in line 2 (X[i] = WAITING) ***must have been committed***, since otherwise, the admin thread would not have seen X[i] == WAITING (i.e.: it would still see X[i] == DONE), and would not have changed X[i] to YOUR_TURN. Therefore, we can infer that X[i]'s write in line 2 has been committed when it exits the loop at line 4. In addition to this, we can also infer that all writes made prior to the write at line 2 have also been committed due to the fact that buffered writes are dequeued in FIFO order.

When termination of a read-loop on a variable v allows us to infer that a previous write to v has been committed, we say that the read-loop and the previous write form a ***synchronization***. So in our example, the write in line 2 to X[i], and the read-loop of X[i] in line 4, together form a synchronization.

4.6.1 Modeling Synchronization

To model the fact that the write of line 2, and the read-loop of line 4 form a synchronization, we use dashed-blue connector to link them together (see figure 9).

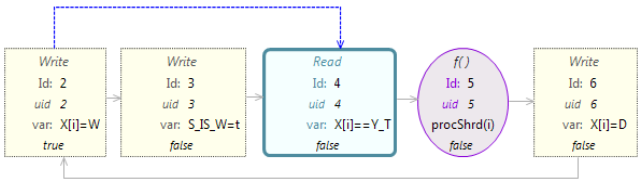

Figure 9. The write of line 2 and the reads of line 4 form a synchronization. This is modeled by the blue dashed Synchronization connector that links the nodes that represent them.

In figure 10, we see the commit graph that is generated from the read/write graph of figure 9. Note that TSO Helper is able to calculate guaranteed commit locations without there being any memory barriers in the input model.

4.7 Branching

None of the algorithms modeled thus far contain any branching. Branching in an algorithm occurs based on the results of read operations on variables. For this reason, branches in a read/write graph are modeled by having multiple outgoing transitions on Read nodes.

To illustrate this, we use the completely contrived read/write model of figure 11 as our example.

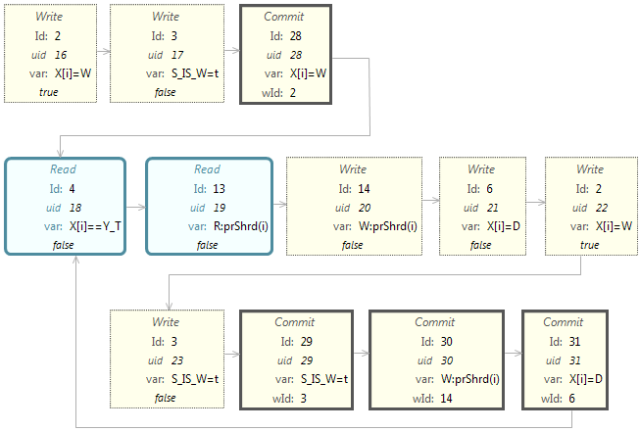

Figure 10. Output graph for listing 5. Commit points for all writes are guaranteed without any memory barriers in the code.

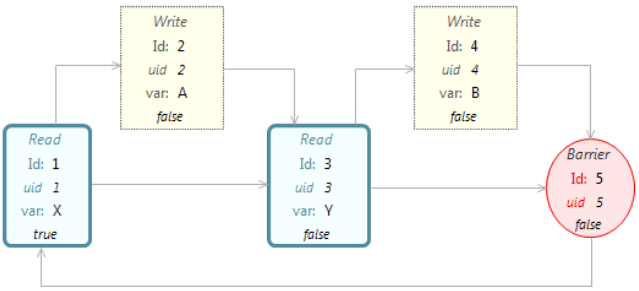

Figure 11. A simple example of a read/write graph with branching.

The output of this example is shown in figure 12.

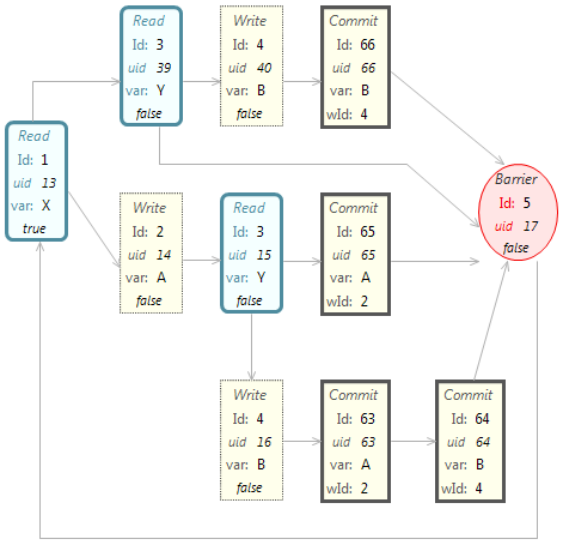

Figure 12. The output graph of the model of figure 11.

5. IMPLEMENTATION

In this section, we outline how TSO Helper analyzes and transforms read/write graphs to produce the output 'commit graphs' of the type we examined in section 4. We also explain the criteria under which TSO Helper determines that no commit graph exists.

5.1 The Ecore Model

The Ecore model for read/write and commit graphs can be seen in figure 13. All nodes inherit from the interface Operation with attributes id, uid, (both EInts), and isStart (an EBoolean). The Transition class is used is used to model connections between Operations that represent the order Operation nodes are executed in the read/write graph. Transition objects take objects of type Operation for the source and target of their connection. The Synch class models connections between Write and Read nodes to model synchronizations, and takes an object of type Write as its source, and type Read as its target. All classes are contained by the TSOReadWriteSynch class.

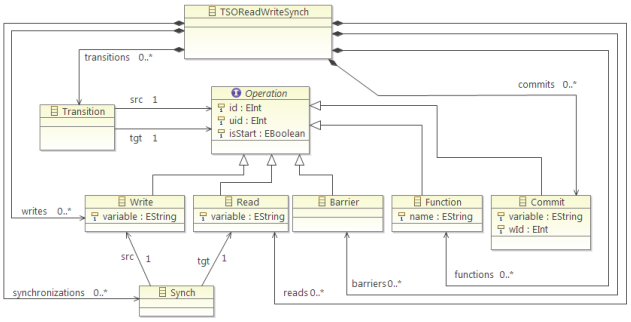

Figure 13. The Ecore model read/write and commit graphs.

5.2 The Transformation Algorithm

Stage 1. In the first step of the transformation, the input graph T1 is transformed into a graph T2 that is identical to T1 except with each Function node in T1 replaced with a Read and Write node (see section 4.1.2).

Stage 2. Graph T2 is then 'unrolled' n times (where n is a user-specified value) to produce a directed acyclic graph T3 rooted at a copy of T2's start node. Figure 14 shows a picture of the result of unrolling the read/write graph of figure 12 two times. Since each node in T2 may appear multiple times in T3 (once for each possible execution it may occur in), the uid attribute is used to differentiate different instances of the same node from T2 in T3. However, note that ***the id of each instance of a node from T2 in T3 remains unchanged***. Hence—***ids are not unique in T3*** (as they were in T1 and T2), but ***uids remain unique in all graphs***. For example, in figures 14 and 15, all red nodes have the id 5, but they all have unique uids.

Stage 3. Once the unrolled 'execution tree' T3 is built, TSO Helper begins transforming T3 into the final commit graph. Beginning at the root node of T3, it performs a recursive DFS traversal of T3.

For each node x of T3 encountered during the DFS traversal, the algorithm augments x with the state that the write buffer will be in immediately after the execution of the operation represented by it.

The write buffer is initially empty. If x is a Read node that is ***not part of a synchronization pair***, it leaves the buffer in its current state. If x is a Write node with id y, it enqueues the number y into the buffer (recall, the buffer is a FIFO queue).

If x is a Barrier, it adds Commit nodes (in the FIFO order they appear in the buffer) between x and x's parent. It then empties the buffer. If x is a Read node that ***is synchronized with a Write node*** with id y, the algorithm dequeues the first Write in the write buffer that has id y, along with all Writes that were enqueued after it. It then adds one Commit node for each Write it just dequeued (in the FIFO order they appeared in the buffer) between x and x's parent.

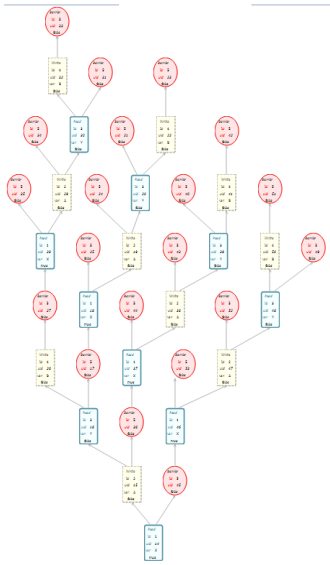

Figure 14. The result of unrolling the read/write graph of figure 12 two times.

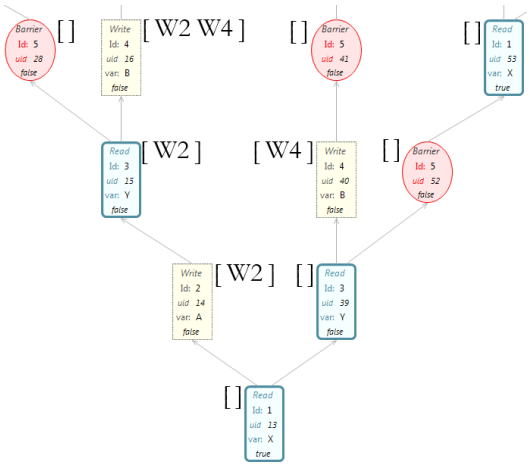

Figure 15. Picture of an early stage of the construction of a commit graph, annotated with each node's write buffer state.

Once x's write buffer has been calculated, and any Commits have been inserted, it then checks to see if any node it has previously encountered during its traversal of T3 is a ***match*** for it.

We say that nodes x and z ***match*** if: 1) x.id = z.id, and, 2) x and z have write buffers with the same sequence of writes. We note that if x and z have the same id—then by definition—they have the same type (e.g.: are both Read nodes or both Write nodes, etc...). Previously-encountered nodes are stored in a hashtable named 'discovered' that is keyed by node id. Each entry in the discovered hashtable is a list of previously encountered nodes with the same id.

If the algorithm finds a matching node z, then it points all of x's incoming transitions to z, and then deletes the subtree rooted at x from the tree and returns.

If the algorithm cannot find a match for x, it stores x in discovered, and continues its recursion.

If x is a leaf node, then the algorithm throws an exception. We justify throwing an exception in this case, because of the lemma of section 5.3.

5.3 TSO Helper Lemma

In the following lemma, let wb(x) be the contents of the write buffer at node x. E.g.: wb(x) is an ordered sequence of write node ids. Let A be an algorithm, and let T be the read/write graph for A (with Function nodes expanded into Read and Write nodes). Finally, let T^{∞} be an infinite unrolling of T.

Lemma. If there exists any infinite path P through T^{∞}, where P starts at the root of T^{∞} and contains no pair of distinct nodes x and z with x.id = z.id and wb(x) = wb(z), then there is an execution of A in which the write buffer grows without bound.

Proof.

Let P be an infinite path through T^{∞}, starting from the root, that contains no pair of distinct nodes x and z with x.id = z.id and wb(x) = wb(z).

For the purpose of obtaining a contradiction, let us assume that write buffer sizes for nodes in P ***do not*** grow without bound. Then it follows that there must be some maximum write buffer size r such that, for each node x in P, $|wb(x)| \leq r$. Hence, the number of unique write buffers is $M = 1 + k + k^2 + ... + k^r$.

Let k be the number ids (NOT uids) used by Write nodes in T^{∞}, and let q be the total number of ids used in T^{∞}. We note that $k \leq q$, and that q must be finite since T^{∞} is an unrolling of T, and T contains a finite number of ids, so T^{∞} also contains a finite number of ids (though it contains an infinite number of uids).

Let z be the $(qM + 1)^{th}$ node in path P. Since nodes in P can have only one of M unique write buffers, and only one of q different ids, there must be some node x amongst the first qM nodes of P with both the same id and same write buffer as z. But this contradicts our initial assumption that that P contains no pair of distinct nodes x and z with x.id = z.id and wb(x) = wb(z). □

6. FUTURE DIRECTIONS

In the future, we hope to continue developing the visual modeling interface for the program. In particular, we would like to develop tools that would better layout generated diagrams. We would also like to better handle self-loops in read/write graphs. Another goal, is to use the functionality of MMTF to enable interconnections between read/write graphs for different threads—we hope that this may lead to additional information that can be used to infer when writes are committed.

7. REFERENCES

[1] Daniel J Sorin, Mark D Hill, and David A Wood, "A Primer on Memory Consistency and Cache Coherence," *Synthesis Lectures on Computer Architecture*, no. 16, 2011.

[2] Sewell Peter, Susmit Sarkar, Scott Owens, Nardelli Zappa Francesco, and Magnus O Myreen, "x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors," *Communications of the ACM*, vol. 53, no. 7, pp. 89-97, July 2010.