Verification of Software Barriers

Alexander Malkis Anindya Banerjee
IMDEA Software Institute
{alexandermalkis,anindya.banerjee}@imdea.org

Abstract

This paper describes frontiers in verification of the software barrier synchronization primitive. So far most software barrier algorithms have not been mechanically verified. We show preliminary results in automatically proving the correctness of the major software barriers.

Categories and Subject Descriptors D.2.4 [*Software engineering*]: Software/program verification—Correctness proofs; Formal methods; Assertion checkers

General Terms verification, algorithms, reliability, theory

Keywords software, barrier, verification, invariant, safety, verifier, counting, central, dissemination, tournament, static, combining, implementation, client

1. Introduction

The software barrier is a standard concurrency primitive. It enables threads to synchronize in the following manner: if a thread calls the barrier function, the barrier function starts waiting until all other threads also call the barrier function. After all other threads have called the barrier function, the waiting stops, and all the threads are allowed to proceed with the instruction following the call to the barrier function.

The *barrier property* we would like to verify is: if one thread passes the barrier, all the other threads have already arrived at it. The paper is devoted to proving just this fact for the major barrier algorithms [1]. We present an overview of the results; the details and the related work are deferred to the full paper version.

2. The simplest central barrier

A trivial implementation of the barrier involves a counter. The counter is initially the number of (participating) threads and gets decreased when a thread enters the barrier. Only when the counter is zero are all the threads allowed to proceed. Each thread executes thus the following code:

`A:` $\mathtt{cnt}{--};$ `B:` **while**$(\mathtt{cnt}\neq 0);$ `C:`

We want to show that when one thread arrives at location `C`, all the others are no more at location `A`. To show that, we need the ability to express that at any time point in an execution of the program, the value of `cnt` is at least the number of threads at location `A`, i.e., $\mathtt{cnt}\geq|\{t\in Tid|pc_t=\mathtt{A}\}|$. As of year 2011, no usable automatic theorem prover can reason about such formulas.

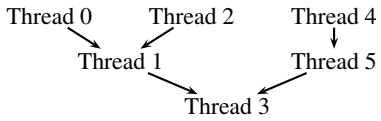
To overcome this restriction, we syntactically convert the multithreaded program into a nondeterministic one working on sets A, B, and C, where A (resp. B or C) is the set of threads whose current control flow location is `A` (resp. `B` or `C`). The problematic formula now turns into a benign $\mathtt{cnt}\geq|A|$, which lies in the QFBAPA logic.

We have verified the central barrier in the verification system Jahob.

3. Tree-based barriers

We will verify the static and combining tree barriers in Jahob.

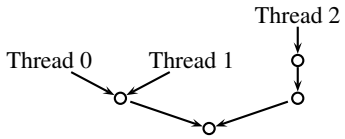
Static tree barrier. In the static tree barrier threads operate on a single tree, in which each thread is statically associated with a distinct tree node, e.g.:



A thread proceeds in two phases: synchronization and wake-up. A thread (say, number 1) starts in the synchronization phase by waiting until its children (0 and 2) have synchronized, then telling the parent (3) that the whole subtree (0,1, and 2) rooted at the thread has synchronized, and then waiting for the wake-up command from the parent (3). In the wake-up phase, once the parent (3) wakes up the thread (1), the thread wakes up its children (0 and 2) and proceeds with the instructions following the barrier call. The root (3) does not wait for a wake-up at the end of its synchronization phase; the static barrier algorithm guarantees that the root starts waking up only if all the threads have started computation.

The threads convey signals via toggling boolean variables in the tree nodes. The crucial part of the inductive invariant that is needed to verify the barrier property speaks about the parent-descendant relation as follows. If a node m (say, 0) is a descendant of a node n (say, 3) and n has received signals from all its children (1 and 5) in the synchronization phase, then m has already sent a signal to its parent (1). This property is encodable in the WS2S logic.

Combining tree barrier. In the combining tree barrier each thread is initially associated with a distinct leaf of a common tree, e.g.:



In the synchronization phase, threads start walking towards the root of the tree from their leaves such that each thread eventually begins waiting at a distinct node and such that exactly one thread reaches the root. In the wake-up phase, the thread pointing to the

PPoPP'12, February 25–29, 2012, New Orleans, Louisiana, USA.
ACM 978-1-4503-1160-1/12/02.

root initiates a wake-up process in which every thread stops waiting and walks towards a leaf of the tree.

Each node of the tree contains a field cnt that stores the number of threads that still have to arrive at the node; e.g., the common parent k of the leaves of threads 0 and 1 starts with $k{\rightarrow}\mathsf{cnt} = 2$. When some thread arrives at k first, it decrements $k \rightarrow \mathsf{cnt}$ to 1 and starts waiting for the wake-up. The next thread that arrives at k diminishes $k{\rightarrow}\mathsf{cnt}$ to 0, and, noticing the fact that no more threads are going to arrive at k, proceeds to the parent of k. Notice that in our picture, thread 2 decrements the counters of the nodes between the initial leaf of thread 2 and the root (excluding both) from 1 to 0 without waiting.

The crucial part of the inductive invariant for the synchronization phase is $J = (\forall\, m{\in}\mathrm{Node}\colon\, m{\rightarrow}\mathsf{cnt} \geq S_1{+}S_2{+}S_3)$, where $S_1 = |\{t{\in}\mathrm{Thread} \mid n_t{=}m \,\wedge\, t \text{ is about to decrement } m{\rightarrow}\mathsf{cnt}\}|$, $S_2 = |\{\overline{m}{\in}\mathrm{Node} \mid \overline{m}{\rightarrow}\mathsf{parent}{=}m \,\wedge\, \overline{m}{\rightarrow}\mathsf{cnt}{>}0\}|$, $S_3 = |\{t{\in}\mathrm{Thread} \mid n_t{\rightarrow}\mathsf{parent}{=}m \,\wedge\, t \text{ is about to go to } m\}|$, where n_t is the pointer into the tree of the thread t.

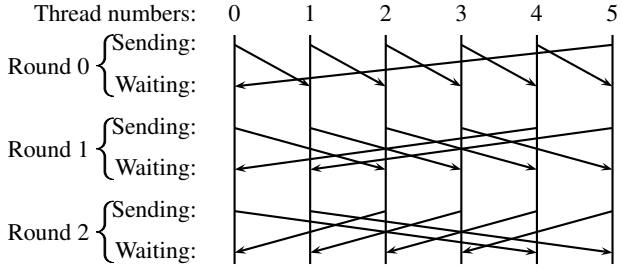
It is possible to encode J into WS2S for finitely many threads and bounded degree of the tree nodes.

4. Array-based barriers

Now we will verify the dissemination and tournament barriers in the verifier Boogie.

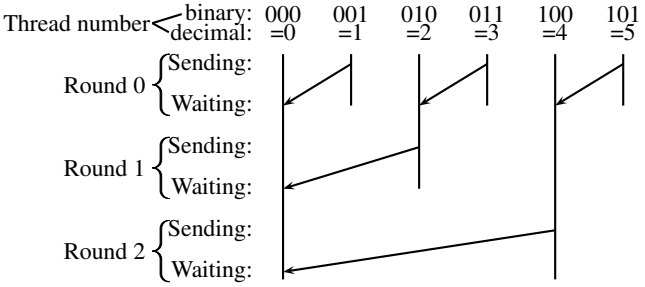
Dissemination barrier. In the dissemination barrier, a thread executes $L = \lceil \log_2 n \rceil$ synchronization rounds. In round i, thread number t sends a signal to thread number $(t + 2^i) \bmod n$ (where $\bmod n$ gives the smallest nonnegative remainder after division by n) and starts waiting for a signal from thread $(t - 2^i) \bmod n$. It turns out that after L rounds, each thread x has received a signal from every other thread y, either directly or transitively, i.e., some thread z has received a signal (directly or transitively) from y and sent its signal to x.

The threads send signals and wait for them as follows.



Signals are stored in an array A: ((thread number)$\times$(round number))$\rightarrow$bool. A crucial property that we had to prove automatically is that if a thread t has received a signal in round l, then all the threads which are less than $t - 2^l$ (including the wrapped ones) by a certain amount have already started. Such a property is a universally quantified property over the two-dimensional array. This property was approximated by a property in the AUFLIA logic and handed over to the SMT theorem prover Z3.

Tournament barrier. A computation of the tournament barrier is akin to a chess tournament: in each round, threads are partitioned into couples, in each couple one of the threads wins and proceeds to the next round, the other thread loses and starts waiting until the winner comes back and wakes the thread up. If the number of threads is not a power of two, threads without opponents skip certain rounds. The difference to the real tournament is that the winner and the loser, as well as the couplings, are statically determined by the bits of the thread identifiers (a nonnegative integer represented by a bitstring). For example, six threads will synchronize as follows:



After the champion (here: 000) has arrived at the last round, it will wake up every thread that has lost to the champion directly during the synchronization phase (here: 100, 010 and 001). Once a thread is awake, it will wake up all the threads that have lost to it directly.

The signals are stored in an array A: ((thread identifier)$\times$(round number))$\rightarrow$bool. The crucial property we needed to prove automatically is that if $A[t, l]$ is set and some thread $\hat{t}$ loses to t transitively till and including round l, then $\hat{t}$ is waiting for the wake-up.

5. Central barrier coded in C

We will verify a C implementation of a central barrier in VCC:

```
shared int cnt = num_threads;   shared bool sense = false;
local bool local_sense = false; // needed for wake−up and reinitialization.
void barrier() {
  local_sense^= true; // toggling, which means: no wake−up yet.
  if(cnt−−>1) // assume atomic fetch−and−decrement.
    while(sense!=local_sense); // wait for the wake−up.
  else{/*Reinit and wake−up*/ cnt=num_threads; sense=local_sense; }}
```

The necessary inductive invariant is similar to that of Section 2. Currently, VCC cannot reason about cardinalities of set comprehensions, so we get rid of cardinalities. Instead, the code is augmented with an auxiliary bijection between thread identifiers and local states (a local state is a valuation of the program counter and local_sense) such that the threads that have not yet fetched-and-decremented have identifiers below cnt. Namely, right after a thread decrements cnt we swap the thread's current identifier with the thread number cnt by updating the aforementioned bijection.

6. A central barrier with a client

If the number of threads is small, as in the following example, and if the implementation of a barrier uses only relatively few states, as the barrier from .NET 4.0, exhaustive search may succeed. The example is a simplification of a client from the MSDN library:

```
int x = 0; Barrier barrier = new Barrier(3);
barrier.AddParticipants(2); barrier.RemoveParticipant();
Action action =  () => { Interlocked.Increment(ref x); // atomic x++
  barrier.SignalAndWait(); Interlocked.Increment(ref x);
  barrier.SignalAndWait(); Interlocked.Increment(ref x);
  barrier.SignalAndWait(); Interlocked.Increment(ref x);
  barrier.SignalAndWait(); assert(x==16); };
Parallel.Invoke(action, action, action, action); barrier.Dispose();
```

Here, four threads increment a variable x four times, calling the barrier function after incrementing. The Spin tool has used automata encoding and partial order reduction to prove no use-after-disposal and that the value of x is 16 at the end of each thread.

Acknowledgments

We thank Ernie Cohen, Viktor Kuncak, Rustan Leino, Michał Moskal, and Thomas Wies for help on Boogie, Jahob, and VCC.

References

[1] M. Herlihy and N. Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.