Brief Announcement: Left-Right - A Concurrency Control Technique with Wait-Free Population Oblivious Reads

Pedro Ramalhete, Andreia Correia

Brief Announcement: Left-Right - A Concurrency Control Technique with Wait-Free Population Oblivious Reads

Pedro Ramalhete[1] and Andreia Correia[2]

[1] Cisco Systems, pramalhe@gmail.com
[2] ConcurrencyFreaks andreiacraveiroramalhete@gmail.com

We present a new concurrency control algorithm with Blocking Starvation-Free write operations and Wait-Free Population Oblivious read operations, which we named the Left-Right technique. This technique requires using two instances of a given resource, and can be used on any data structure, allowing concurrent access to it, similar to a Reader-Writer lock, but in a non-blocking manner for reads, and it does not need an automatic Garbage Collector (GC).

To allow concurrent read and write access to a data structure or object written for single threaded execution, a common approach is to use a Reader-Writer lock. Another alternative is Copy-On-Write (COW), which consists of replacing the instance by a copy of that instance with the applied modification. Peterson [3] has presented several solutions to the *Concurrent Reading While Writing* problem. One of them guarantees wait-free progress for both reads and writes, allowing Readers and Writer to access simultaneously *buff1* and *buff2* instances, which compromises memory reclamation.

The Left-Right is a concurrency control technique with two identical objects or data structures, that allows an unlimited number of threads (Readers) to access one instance in read-only mode, while a single thread (Writer) modifies the other instance. The Writer starts by working on the right-side instance (`rightInst`) while the Readers read the left-side instance (`leftInst`), and once the Writer completes the modification, the two instances are *switched* and new Readers will read from the `rightInst`. The Writer will wait for all the Readers still running on the `leftInst` instance to finish, and then repeat the modification on the `leftInst`, leaving both instances up-to-date. It us up to the Writer to ensure that Readers are always running on the data structure that is currently *not* being modified. The synchronization between Writers is achieved with an exclusive lock that is used to protect write-access (`writersMutex`).

The components ensuring a Writer performs in exclusivity are the following: a `leftRight` variable which is toggled by the Writer between `LEFT` and `RIGHT`, that indicates which instance the Readers should go into; a `versionIndex` variable, which is modified by the Writer, functioning like a *timestamp*; and a Reader's indicator [1], `readIndic`, for each Reader to *publish* the `versionIndex` it read. The `readIndic` is a data structure that allows Readers to publish their state through `arrive()` and `depart()`, and for the Writer to determine the presence of ongoing Readers with `isEmpty()`. A simple implementation of the `readIndic` is to use two single atomic synchronized counters, one per `versionIndex`.

```
const int arrive(void) {
  int vi = _versionIndex.load();
  _readIndic[vi]->arrive();
  return vi;
}


void depart(const int vi) {
  _readIndic[vi]->depart();
}


void toggleVersionAndWait(void)
{
  int vi = _versionIndex.load();
  int p = vi & 0x1;
  int n = (vi+1) & 0x1;
  while(!_readIndic[n]->isEmpty())
  {
    this_thread::yield();
  }
  _versionIndex.store(n);
  while(!_readIndic[p]->isEmpty())
  {
    this_thread::yield();
  }
}
```

```
template<typename R, typename A>
R applyRead(A& arg1,
            function<R(T*,A)>& f) {
  const int vi = arrive();
  T* inst = _leftRight.load() == LEFT
          ? _leftInst : _rightInst;
  R ret = f(inst, arg1);
  depart(vi);
  return ret;
}


template<typename R, typename A>
R applyMut(A& arg1,
           function<R(T*,A)>& f) {
  lock_guard<mutex> m(_writersMutex);
  if (_leftRight.load() == LEFT) {
    f(_rightInst, arg1);
    _leftRight.store(RIGHT);
    toggleVersionAndWait();
    return f(_leftInst, arg1);
  } else {
    f(_leftInst, arg1);
    _leftRight.store(LEFT);
    toggleVersionAndWait();
    return f(_rightInst, arg1);
  }
}
```

As shown on the C++ code above, the Writer calling `applyMut()` will acquire the lock on `writersMutex` to guarantee mutual exclusivity between Writers, and proceed to modify the instance opposite to the one currently referenced by `leftRight`. Then, it toggles the `leftRight`, making the modification visible to new Readers. The final step is to modify the other instance, but first, it is necessary to guarantee that no Reader is accessing the intance, and this guarantee is provided by `toggleVersionAndWait()`. The method `applyRead()` shown above has no loops, and always executes in a constant number of steps, thus ensuring that read operations are wait-free population oblivious.

In summary, read operations can run concurrently with all operations, and will *never have to wait* for a Writer or for other Readers. Moreover, new Readers have no impact on the Writer's progress, making its progress starvation free relative to Readers. In addition, Writers will be starvation free if a starvation free `writersMutex` lock is used. We believe that due to its performance, low latency, and flexibility of usage, in practice, this technique can be used to wrap any data structure or object, as an alternative to other synchronization techniques, such as Reader-Writer locks, or COW with RCU [2] memory reclamation.

References

1. Y. Lev, V. Luchangco, and M. Olszewski. Scalable reader-writer locks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 101–110. ACM, 2009.
2. P. E. McKenney and J. Walpole. What is rcu, fundamentally? 2007.
3. G. L. Peterson. Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(1):46–55, 1983.