Supplementary Material for: Parallelization of Dynamic Languages: Synchronizing Built-in Collections

BENOIT DALOZE, Johannes Kepler University Linz, Austria
ARIE TAL, Technion, Israel
STEFAN MARR, University of Kent, United Kingdom
HANSPETER MÖSSENBÖCK, Johannes Kepler University Linz, Austria
EREZ PETRANK, Technion, Israel

This is supplementary material for the paper *Parallelization of Dynamic Languages: Synchronizing Built-in Collections* by Daloze et al. [2018].

ACM Reference Format:
Benoit Daloze, Arie Tal, Stefan Marr, Hanspeter Mössenböck, and Erez Petrank. 2018. Supplementary Material for: Parallelization of Dynamic Languages: Synchronizing Built-in Collections. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 108 (November 2018), 6 pages. https://doi.org/10.1145/3276478

1 REASONING ABOUT CORRECTNESS FOR THE LIGHTWEIGHT LAYOUT LOCK

Similarly to the Layout Lock [Cohen et al. 2017], the Lightweight Layout Lock must satisfy the following properties: (a) layout change operations are mutually exclusive, (b) write operations are not concurrent with a layout change operation, and (c) optimistic read operations detect a concurrent layout change operation and fail, allowing for recovery.

In this section, we sketch the correctness of the Lightweight Layout Lock protocol by describing invariants, which correspond to the above correctness properties. We argue that if these invariants hold, the lock is providing the necessary synchronization for concurrent strategies.

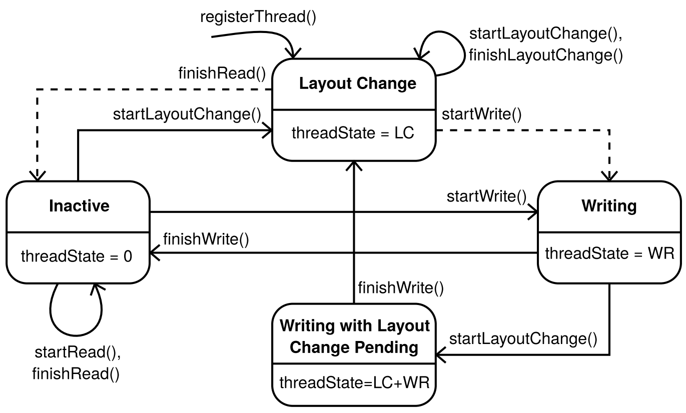
The correctness of the Lightweight Layout Lock invariants depends on a `baseLock` that correctly satisfies shared vs. exclusive locking invariants, such as a Reader-Writer Lock [Hillebrand and Leinenbach 2009].

The Lightweight Layout Lock does not support nesting API calls, nor does it support upgrading (i.e. atomically turning a read operation into a write operation, or a write operation into a layout change operation). Similar to other locks, an operation is comprised of the following steps: (a) a *prologue*, usually an API call (a call to a `start` method in the case of the Lightweight Layout Lock), (b) a *body*, the user code that performs the actions that are synchronized by the lock, and finally (c) an *epilogue*, a symmetric API to the *prologue* that performs synchronization actions related to finishing the operation (a call to a `finish` method in the case of the Lightweight Layout Lock).

A `threadState` is an `AtomicInteger` that associates a thread and a Lightweight Layout Lock instance. The thread states associated with a specific lock instance are maintained in that lock's `threadStates` list. The value of a `threadState` is a combination of two bits: The `LC` bit and `WR` bit,

Authors' addresses: Benoit Daloze, SSW, Johannes Kepler University Linz, Austria, benoit.daloze@jku.at; Arie Tal, Computer Science, Technion, Israel, arietal@cs.technion.ac.il; Stefan Marr, School of Computing, University of Kent, United Kingdom, s.marr@kent.ac.uk; Hanspeter Mössenböck, SSW, Johannes Kepler University Linz, Austria, moessenboeck@ssw.jku.at; Erez Petrank, Computer Science, Technion, Israel, erez@cs.technion.ac.il.

2475-1421/2018/11-ART108
https://doi.org/10.1145/3276478



Supp. Figure 1. This diagram shows all the state transitions of a `threadState`, as they are triggered by the different API calls of the Lightweight Layout Lock. The dashed arrows indicate the reader and writer slow paths (i.e. a call to `slowSet()` in Listing 2).

representing four states: `0` (*Inactive*), `WR` (*Writing*), `LC` (*Layout Change*), and `LC` + `WR` (*Writing with Layout Change Pending*). The `threadState` value is modified by the various API calls, as shown in Supp. Figure 1. Each transition of the `threadState` is performed atomically either by actions of the owning thread or by a different thread that starts a layout change.

Invariant 1. *For some* `threadState` *of thread T and lock lll, if T clears the* `LC` *bit, then*

- *T called either* `finishRead()` *or* `startWrite()`.
- *the* `rescan` *flag was set to* `true` *prior to clearing of the* `LC` *bit.*
- *clearing of the* `LC` *bit was performed while the baseLock was held in shared mode.*

The `slowSet()` method is used to set `rescan` to `true` and clear the `LC` bit. It is not invoked directly by user code, but rather as part of the read operation epilogue and the write operation prologue. It is the only method provided by the Lightweight Layout Lock that can clear the `LC` bit. The `slowSet()` method locks the `baseLock` in shared mode prior to modifying `rescan` and the `threadState`, and releases the `baseLock` when done. Note that although `startWrite()` and `finishWrite()` update the `threadState` directly (at lines 22 and 30 in Listing 2, respectively), they only modify the `WR` bit.

Invariant 2. *For a given Lightweight Layout Lock instance lll,*

$$lll.\texttt{rescan} = \texttt{false} \implies \forall ts \in lll.\texttt{threadStates},\ ts = \texttt{LC}\ \textit{(Layout Change)}$$

When the Lightweight Layout Lock is initialized, the `rescan` flag is set to `false` (at line 4 in Listing 2) and there are no threads registered with the lock (i.e. `threadStates = {}`). So this invariant holds when creating the lock since there are no `threadStates`.

When a new `threadState` is registered with the lock, it is first set to `LC` (*Layout Change*, at line 68), thus thread registration preserves this invariant.

When a reader calls `finishRead()` and detects (at line 13) that the `threadState` is not 0 (not *Inactive*), it calls `slowSet()` (at line 16) to set the `threadState` to `0` (*Inactive*). When a writer calls `startWrite()` and the CAS (at line 22) fails (i.e. the `threadState` value is `LC` (*Layout Change*)), it calls `slowSet()` (at line 25) to set the `threadState` to `WR` (*Writing*). By Invariant 1, `slowSet()` sets `rescan` to `true` prior to modifying the `threadState`. Therefore, in the only two cases that clear the `LC` bit, Invariant 2 is maintained.

The `rescan` flag can only be reset by `startLayoutChange()` when it detects (at line 46) that the `rescan` flag is `true`. It then scans all the registered `threadStates` and either verifies that they are already equal to `LC` (*Layout Change*), or turns on the `LC` bit, thus transitioning the `threadState` into either `LC` (*Layout Change*) or `LC` + `WR` (*Writing with Layout Change Pending*) (cf. Supp. Figure 1). If the thread that owns the `threadState` is concurrently performing a write operation, i.e. `threadState` equals `LC` + `WR` (*Writing with Layout Change Pending*), `startLayoutChange()` will busy-wait until the the thread calls `finishWrite()` and clears the `WR` bit.

After all `threadStates` have been set to `LC` (*Layout Change*), `startLayoutChange()` sets `rescan` to `false`. The scanning is performed while the `baseLock` is held in exclusive mode, and by Invariant 1, the `LC` bit is only cleared while holding the `baseLock` in shared mode (by `slowSet()`), thus clearing of the `LC` flag and setting the `LC` flag are mutually exclusive. The `LC` bits can only be set and the `rescan` flag can only be reset in `startLayoutChange()`. Therefore, when `startLayoutChange()` sets `rescan` to `false`, all registered `threadStates` must equal `LC` (*Layout Change*), satisfying Invariant 2.

Invariant 3. *For a given Lightweight Layout Lock instance lll,*

$$\exists ts \in lll.\texttt{threadStates},\ ts \neq \texttt{LC}\ (\textit{Layout Change}) \implies lll.\texttt{rescan} = \texttt{true}$$

This is trivially true by a contraposition of Invariant 2.

Invariant 4. *The body of a write operation and the body of a layout change operations are mutually exclusive.*

For this invariant, we need to look at the two cases of a write operation: the fast path and the slow path. The write operation fast path begins by a successful CAS of the `threadState` from `0` (*Inactive*) to `WR` (*Writing*). If a layout change prologue runs concurrently, then by Invariant 3, the `rescan` flag must be `true`, and thus it loops over the registered `threadStates` and tries to set them to `LC` (*Layout Change*). However, since the CAS of `startWrite` succeeded for `threadState`, then the comparison at line 49 fails for that same `threadState` (either before or after the CAS), thus the value `LC` is atomically added to the `threadState` at line 51 resulting in `LC` + `WR` (*Writing with Layout Change Pending*). `startLayoutChange()` will then busy-wait until the thread calls `finishWrite()` and clears the `WR` bit, letting the `threadState` become `LC` (*Layout Change*) and releasing the busy-wait loop. Thus the layout change operation body cannot start until all write operations fast paths completed.

The write operation slow path begins by a failing CAS of the `threadState` from `0` (*Inactive*) to `WR` (*Writing*). The slow path proceeds by calling `slowSet()` with the value `WR` (*Writing*), which requires locking the `baseLock` in shared mode. However, since a layout change prologue begins by locking the `baseLock` in exclusive mode (`startLayoutChange()` at line 45), and releases the lock only after the layout change operation completed (at line 63), the write operation is suspended until the layout change operation is completed. Thus a write operation body cannot start before a layout change body completes.

Since `startWrite()` always sets the `WR` bit (either via a successful CAS or `slowSet()`), `finishWrite()` is the only operation clearing the `WR` bit, and `startLayoutChange()` waits until the `WR` bit is removed, the layout change operation body cannot start until all write operation bodies completed.

Invariant 5. *During a layout change operation body*

- `baseLock` *is held in exclusive mode.*
- `rescan` *is* `false`*.*
- *all registered* `threadStates` *equal* `LC` *(Layout Change).*

The layout change prologue in `startLayoutChange()` begins by acquiring the `baseLock` in exclusive mode. The lock is released at `finishLayoutChange()`, thus it is held in exclusive mode during the layout change operation. When `startLayoutChange()` completes, `rescan` must equal `false`, since `startLayoutChange()` resets the `rescan` flag, and the flag can only be set by `slowSet()` which requires holding the `baseLock` in shared mode. However, the `baseLock` is held in exclusive mode during a layout change operation, thus `rescan` remains false at least until the end of the layout change operation. By Invariant 2 it is implied that all registered `threadStates` equal `LC` (*Layout Change*) when `rescan` is `false`.

Invariant 6. *For a given lock instance lll and a thread T, a call to lll.*`finishRead()` *by T returns* `false` *if a call to lll.*`startLayoutChange()` *completed since the previous call by T to lll.*`finishRead()` *or to lll.*`startWrite()`.

In other words, a read operation is made aware of a concurrent or previous layout change at `finishRead()`, and returns `false` to initiate a recovery (i.e. restart of the read operation).

By Invariant 5, all `threadStates` are set to `LC` (*Layout Change*) when `startLayoutChange()` completes and the `baseLock` is held in exclusive mode. Therefore, if `startLayoutChange()` completed since the previous call by `T` to `finishRead()` or `startWrite()`, then a subsequent call by `T` to `finishRead()` must go through the slow path (i.e. call `slowSet()`) since `threadState` equals `LC` (*Layout Change*). After `slowSet()`, `finishRead()` will return `false` (at line 17).

Conclusion. These invariants ensure that writing and layout changes are mutually exclusive, and that reading of a value after a layout change started is aware of the layout change and can perform the needed recovery. Thus, we argue that the Lightweight Layout Lock provides the necessary synchronization to ensure correctness of concurrent strategies.

2 SAFE PUBLICATION OF SHARED COLLECTIONS

An interesting semantic example arises when sharing a newly-created collection, and reading its contents from another thread. We discuss this example here, because it needs low-level implementation details to be explained. In our implementation, it is guaranteed that the other thread will observe values at least as recent as set just before sharing the collection. In other words, safe publication [Shipilëv 2014] is guaranteed for newly-created collections shared to other threads. Supp. Table 1 illustrates an example where safe publication ensures the same outcome as a global lock.

Supp. Table 1. An example illustrating safe publication.

	Example		GIL	Goal	Unsafe
7	Initial: `array = [1]`		1	1	1
	`array = [2]`	`print array[0]`	2	2	2
					0

Unsafe has the additional outcome `0`, because it does not guarantee safe publication and as a result might read the element before it is initialized to `2`. In Java, memory is guaranteed to be zero-initialized for allocations, so only zero values can be observed (including `null` for an `Object[]`). We consider `0` an out-of-thin-air value as it is never written by the program. The program only sets the first array element to `1` and `2`, but never to `0`. Therefore, we want to prevent this outcome.

Our implementation prevents observing `0` by having safe publication semantics for shared objects and collections. This is guaranteed by the process of sharing an object or collection, which involves having a memory barrier between marking the object or collection as shared and the

```
// assuming array is a variable shared between threads,
// the Ruby code array = [2] becomes:

RubyArray tmp = new RubyArray();
tmp.storage = new int[] { 2 };

tmp.shared = true; // mark as shared
Unsafe.storeFence(); // or just a StoreStore barrier
array = tmp; // publish the array to other threads
```

Supp. Figure 2. Pseudo-code illustrating the implementation of the write barrier sharing a collection.

actual assignment making the object or collection reachable from other threads. This process is detailed in Supp. Figure 2. Daloze et al. detail the write barrier but do not mention the memory barrier, which was an omission in that paper. The barrier is also required to ensure other threads correctly see this object or collection as shared.

3 RAW DATA TABLE FOR FIGURE 5

Supp. Table 2. Raw data table for Figure 5, showing the throughput relative to `Local` with 1 and 44 threads, as well as the scalability with 44 threads normalized to the 1-thread performance of the configuration.

Benchmark	100% reads			90% reads, 10% writes			50% reads, 50% writes		
	Throughput		Scaling	Throughput		Scaling	Throughput		Scaling
Threads	1	44	44	1	44	44	1	44	44
`Local`	1.000	1.000	45.1	1.000	1.000	39.2	1.000	1.000	42.4
`SharedFixedStorage`	1.030	0.997	43.7	1.036	1.079	40.8	0.997	0.897	38.1
`VolatileFixedStorage`	0.351	0.330	42.4	0.255	0.300	46.1	0.137	0.154	47.6
`LightweightLayoutLock`	0.259	0.247	43.0	0.162	0.181	43.9	0.079	0.081	43.7
`LayoutLock`	0.230	0.223	43.6	0.164	0.185	44.4	0.078	0.080	43.7
`StampedLock`	0.189	0.181	43.4	0.136	0.001	0.4	0.062	0.000	0.1
`ReentrantLock`	0.045	0.001	0.9	0.038	0.001	0.9	0.039	0.001	0.9

Figure 5 shows how each configuration scales from 1 to 44 threads but it is hard to observe some data points due to overlapping lines. Therefore we provide the most relevant data in Supp. Table 2. As the table shows, the throughput of `ReentrantLock` is very low even on a single thread. `StampedLock` does not scale when there are concurrent writes, and is even significantly slower on 44 threads than on 1 thread. Other locks scale well on these 3 benchmarks, but the throughput of `Local` and `SharedFixedStorage` is many times faster than other configurations.

REFERENCES

Nachshon Cohen, Arie Tal, and Erez Petrank. 2017. Layout Lock: A Scalable Locking Paradigm for Concurrent Data Layout Modifications. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '17)*. ACM, 17–29. https://doi.org/10.1145/3018743.3018753

Benoit Daloze, Stefan Marr, Daniele Bonetta, and Hanspeter Mössenböck. 2016. Efficient and Thread-Safe Objects for Dynamically-Typed Languages. In *Proceedings of the 2016 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '16)*. ACM, 642–659. https://doi.org/10.1145/2983990.2984001

Benoit Daloze, Arie Tal, Stefan Marr, Hanspeter Mössenböck, and Erez Petrank. 2018. Parallelization of Dynamic Languages: Synchronizing Built-in Collections. In *Proceedings of the 2018 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA'18)*. https://doi.org/10.1145/3276478

Mark A. Hillebrand and Dirk C. Leinenbach. 2009. Formal Verification of a Reader-Writer Lock Implementation in C. *Electronic Notes in Theoretical Computer Science* 254 (2009), 123–141.

Aleksey Shipilëv. 2014. Safe Publication and Safe Initialization in Java. https://shipilev.net/blog/2014/safe-public-construction/