École Polytechnique

BACHELOR THESIS IN COMPUTER SCIENCE

Linearization of Concurrent Data Structures

Author:

Mark Daychman, École Polytechnique

Advisor:

Prof. Constantin Enea, LIX & CNRS

Academic year 2022/2023

Abstract

Linearizability is a commonly accepted correctness criterion for concurrent data structures. [8] It is a strong consistency model that constrains how an object is accessed by multiple processes. Checking linearizability of an execution is a well-known NP-problem in concurrent computing. However, for some data structures, for instance a concurent queue [1], this problem can be solved in polynomial time. In this paper, we look at the restriction of this problem to a specific class of operations: writes, reads and compare-and-swaps. We will show that certain data structures, like shared logs can be reduced to these simple operations and thus be linearized in polynomial time.

Python implementation of the algorithm can be found here.

Contents

1 Introduction

1.1 Introduction of Concurrency and Consistency Models

Concurrency is an ability of a system to execute parts of the program in a different order without affecting the final result. These parts can be physically executed at the same time on different cores of a CPU (parallel computing) or they can execute on the same core by, for example, interleaving their execution, but at all times only one computation takes place. The main goal of concurrent algorithms is to achieve the same result as the sequential algorithm, but in a shorter time. The problem that arises when two concurrently executing threads attempt to communicate and share some common resource. Potential problems include race conditions (when a thread reads a value that is then updated by another thread) or deadlocks (when two threads are waiting for each other to release a resource). On top of that the number of execution paths in a concurrent program grows exponentially with the number of threads and operations, and the result can be indeterminant.

The problems that may arise during concurrent calculations motivate the creation of consistency models that specify how different threads can interact with these shared resources. One of the fundamental consistency models is sequential consistency. This model informally states that all operations within a single thread or core are executed in the order they appear in the program, but the order of the threads themselves is undefined. The main benefit of sequential consistency is that it provides a simple and intuitive model for understanding the behavior of concurrent programs. The programmer can reason about the program's execution as if it were a single sequential program, which makes it easier to develop and debug concurrent programs. Additionally, sequential consistency ensures that all threads observe the same order of operations, which can prevent errors such as race conditions and deadlocks. However, a critical flaw of sequential consistency model is non-compositionality. [4] If independent operations on two different resources are sequentially consistent, when placed in the context of one program, the result is not guranteed to be sequentially consistent.

A stronger consistency model that resolves this problem is linearizability, defined by Herlihy and Wing. [5] "Linearizability provides the illusion that each operation applied by concurrent processes takes effect instantaneously at some point between its invocation and its response, implying that the meaning of a concurrent object's operations can be given by pre- and post-conditions." (Herlihy and Wing, 1987) Two of the greatest advantages of linearizability over a sequential model is that it is compositional and non-blocking. Compositionality as explained earlier means that if linearizability of operations is proved for several programs separately, then the programs together will also be linearizable. In a linearizable program the launched operations do not require other operations to be launched for their termination. This is the property of non-blockability.

1.2 Non-blocking Algorithms and Atomic Operations

There are some classic approaches to combating the problems of concurrency. One of them is to use locks. Locks are a mechanism for controlling access to a shared resource. A lock can be in one of two states: locked or unlocked. When a thread wants to access a shared resource, it first acquires the lock for that resource and only then reads from or modifies the memory. This prevents other threads from accessing the resource until the lock is released. The main problem with locks is that they can cause a thread to block, waiting for the lock to be released. This can cause a thread to wait indefinitely if another thread holding the lock crashes or is otherwise unable to release it. This is known as a deadlock. Additionally, locks can cause significant performance overheads because of the need for synchronization between threads. We do not want our concurrent algorithms to block when one of the threads fails and never releases the lock. Algorithms that have this property are called non-blocking. With few exceptions, non-blocking algorithms use atomic read-modify-write primitives that the hardware must provide, the most notable of which is compare and swap (CAS). [3]. CAS is an atomic version of the pseudocode shown in figure 2. Atomic operations are operations that are guaranteed to be executed atomically by the hardware. This means that the operation will either complete in its entirety or not at all, and no other thread will be processed in parallel. This property is useful when implementing concurrent algorithms, because without the inefficiency of locks, we can be certain to avoid race conditions. Two other most commonly used atomic operations are atomic writes and reads, and some architectures even include more complex operations, such as atomic test-and-set and fetch-and-add, as well as store conditionals. [7] In reality, on a hardware level, the mutex locks themselves, as well as other synchronization mechanisms like semaphores, are implemented using atomic operations. [2]

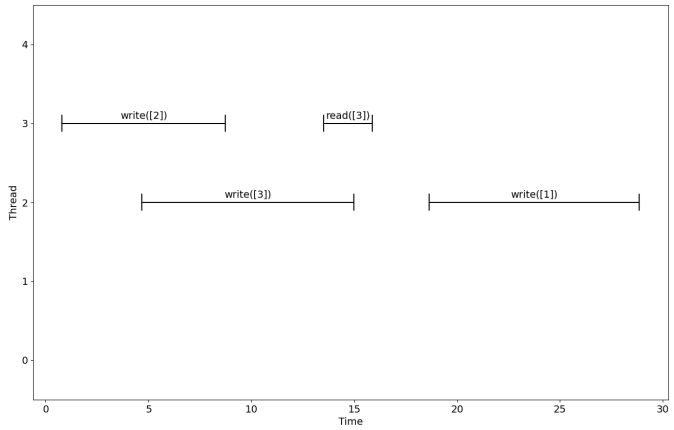

Figure 1: An example of a 2-thread history

function CAS(ptr, old, new)
  if $^*ptr \neq old$ **then**
    return False
  $^*ptr \leftarrow new$
  return True

Figure 2: Atomic CAS pseudocode

2 Checking Linearizability

2.1 Problem Introduction

As stated before, linearizability of an execution (also known as a history), assumes that the effect of an operation takes place instantaneously at some moment between the call of the operation and its return. An execution is a sequence of operations. Each operation has a thread number, a type (read, write, etc.), a list of arguments and a start/return time. To linearize an execution means to find a point in time for each operation in the history, such that if the effect of an operation takes place instantaneously at that point of time, the execution is still valid.

In general, the problem of checking whether a specification is linearizable is NP-complete. [6]. However, for some sets of operations and under certain constraints this problem can be solved in polynomial time. A common example is a concurrent stack with queues and dequeues on unique values. [1]. In this paper, we will focus on linearizing the following operations:

- *write(addr, v)* - Writes a value v to a memory location $addr$.
- *read(addr)* - Reads a value from a memory location $addr$. Does not modify the value.
- *true_cas(addr, old, new)* - Expects the value old to be at $addr$ and replaces it with new.
- *false_cas(addr, v)* - Expects the value at $addr$ to be anything, except v. Does not modify the value.

Furthermore, since linearizability is compositional, we can individually verify linearizability on each address. The execution as a whole is linearizable if and only if the operations on each address are. Because of this, we will only reason about the linearization of operations on an individual memory cell, as the algorithm can be easily extended to the entire memory array.

There also is a list of assumptions and conditions under which the algorithm works:

1. Each write on a value is unique, i.e. the same value cannot be written to the same address twice, both via an explicit write or a true CAS.
2. The initial value is undefined. All reads, true and false CASes will fail if they encounter an undefined value.
3. No operation can begin or return at the **exact** same time as another operation on a different thread.
4. **No false CAS can overlap with any write on any value on any thread.**

Condition 1 is useful, so we could reason about a "lifetime" of a value. In a practical context, if the same value is written many times, we can apply versioning and replace future writes with a dummy variant of a value. This does not have any impacts on the overall execution. Condition 2 and 3 are needed to avoid ambiguity. In real life, no two calls can truly

happen at the same time, so for purpose of demonstration and testing, a negligible amount of noise is added to the begin and return times of each operation to avoid collision. Finally, condition 4 is needed because otherwise reasoning about the order of the execution becomes significantly more complicated. This condition can potentially be replaced by a looser condition on the false CASes, but this requires further investigation.

For the ease of notation, we will occasionally use the following shorthands for CASes:

- *true_cas(old, new)* is denoted `old->new`
- *false_cas(v)* is denoted `!v`

In the figures, the duration of each operation is going to be denoted as an interval between the starting and return times on a corresponding thread.

Lastly, to provide an example, we will linearize the execution shown in figure 1.

- The linearization point of `read(3)` must happen after that of `write(3)`, as otherwise the read fails.
- The intervals of `write(2)` and `write(3)` intersect, but since linearizing `write(2)` after `write(3)` would fail the read, we must linearize it before.
- Finally, the linearization point of `write(1)` in all cases happens last.

This forces a unique linearization: `[write(2), write(3), read(3), write(1)]`. Notice how we did not specify the exact times of linearization points, only their relative positions to each other. In reality, it does not matter where the points are, as we only need to properly order the operations. From now, we will only reason directly about the order of operations, rather than the exact position of their linearization points. Lastly, it must be noted that in general, the linearization order need not be unique.

2.2 Brute Force Approach

```
1:  function DFS(threads, state)
2:      res ← []
3:      first_op_per_thread ← list of first operations for each thread in threads
4:      if first_op_per_thread is empty then
5:          return res
6:      ref ← operation with the earliest return in first_op_per_thread
7:      candidates ← all operations in first_op_per_thread that intersect ref
8:      for c in candidates do
9:          new_state, status ← Execute c on a copy of state
10:         if status is fail then                    ▷ Unexpected value in memory, etc.
11:             continue
12:         threads.remove(c)
13:         sol ← dfs(threads, new_state)             ▷ Recursive step
14:         threads.append(c)
15:         lin ← [c] + sol
16:         res.append(lin)
17:     return res
18:
19: function LINEARIZE_GENERIC(spec, state)
20:     Sort spec by thread and sort operations within each thread by the starting point
21:     return DFS(sorted spec, state)
```

Figure 3: Brute force algorithm for linearizing an execution

The obvious brute force approach to linearize a specification is to try all possible linearizations and check which ones are valid. This approach is infeasible for large specifications, because of the exponential sample space, but it can be used to check small specifications. Programmatically, we sort the operations by thread number and within each thread by the starting point of each operation. We keep track some internal state depending on which kind of operations we are trying to linearize (a stack, a memory array, etc.). We pick the operation that starts the earliest, and consider the list of "candidates" to be executed next. The candidates are all operations intersection the earliest operation. Any operation in the list of candidates can be executed next without violating the real-time order, but this does not mean that it will not violate the specification. For each candidate we make a branching, we execute it and run the algorithm recursively on the remaining operation. If the execution of the candidate fails, we backtrack. If no operations are left, that means we fully linearized the history, and we propagate the result up the recursion attack. The algorithm is shown in figure 3.

2.3 Write and Read

2.3.1 Introduction of Blocks and Intervals

Atomic read and write are the simplest operations to linearize. First observation, we can make is that if we group all writes and reads on the same value together into a dictionary `sort_by_val` (key: value, value: list of operations), then the write will always be linearized first and the reads will always be linearized directly after a corresponding write in any way that does not break the real-time order. In other words, to determine the linearization of the entire history, we only need to order the values, as the order of operations on a value is pre-determined. To reason about the order of values, we can introduce **value blocks**, a concept that will be used throughout this paper. Value blocks are a list of lists of values that defines some partial order on them. We traverse the blocks from left to right, and within each block, we can traverse the operations in any order. For example, the blocks `[[1, 2], [3, 4]]` mean we would first linearize 1 and 2 together in any order, and only then linearize 3 and 4 again any order. Blocks are an efficient way to store everything we know about the linearization order at any given moment. As we add new types of operations, the blocks can be gradually refined. The exact algorithm to build the blocks will be explained later, when the exact linearization order becomes more important. With only writes and reads, it is enough to only determine whether the history is linearizable at all.

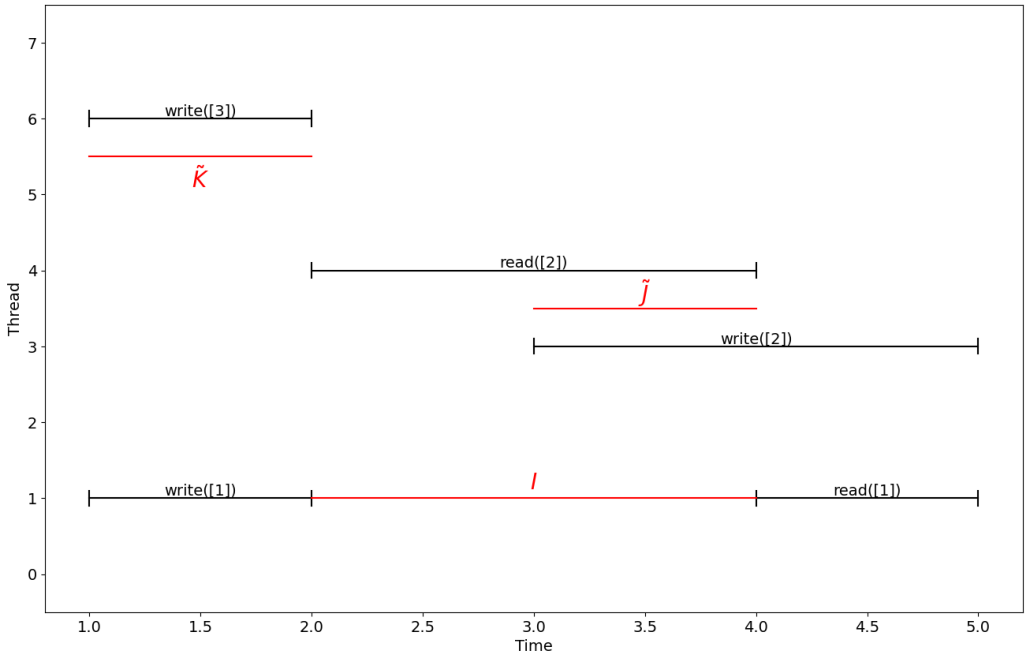

Figure 4: Examples of forward and reverse intervals

To make the blocks we need to understand what forces a certain order of values. First, if there are two operations on two different values, and the first operation returns before the second operation starts, then all operations on the first value must be linearized first. Second, if a write returns, but the read on the same value happens later, during the interval between the return and the call, we are certain which value is written in the memory. To further reason about this, we can introduce

value intervals. A value interval starts at the earliest return of an operation on that value and ends on the latest call.

```
1:  function MAKE_INTERVALS(sort_by_val)
2:      intervals ← { }
3:      for val, ops_on_val in sort_by_val do
4:          i1 ← min(c.end | c ∈ ops_on_val)
5:          i2 ← max(c.start | c ∈ ops_on_val)
6:          if i1 < i2 then
7:              intervals[val] ← I(i1, i2)
8:          else
9:              intervals[val] ← I(i2, i1, reversed = True)
10:     return intervals
```

Figure 5: Interval construction algorithm

The intervals can be broadly divided into two types: forward and reverse, based on whether the latest call happens after or before the earliest return. One can see the forward interval I and the reverse intervals $\tilde{J}$ and $\tilde{K}$ in figure 4. We will always denote a reverse interval by a tilde on the top. Notice that the read can start before a read starts, as long as they intersect, or there can be no read at all, but the intervals are still well-defined.

Each interval type has its own interpretation. The forward intervals force the value to be in memory during the interval. It does not mean that the value is only written to memory during it, it only means that during the interval, we are sure to have it written. The interpretation of reverse intervals is more vague, but we will roughly assume that all linearization points on the given value happen during the reverse interval. In reality, this does not need to be true, because we can occasionally linearize operations outside the interval, but this is never required and furthermore, we always want the linearization of a value to be as short as possible, keeping the linearization points close to each other. So in practice, there is never a need to linearize operations outside the interval.

2.3.2 Interval Check

```
1:  function INTERVAL_CHECK(intervals)
2:      for forward_interval in forward intervals of intervals do
3:          for interval in intervals do
4:              last_call ← time of the last call on the value of interval
5:              first_return ← time of the first return on the value of interval
6:              if first_return < forward_interval.end and
                   last_call > forward_interval.start then
7:                  return False
8:      return True
```

Figure 6: Writes/reads linearization verifier

We are now linearize writes and reads. We first check that no read returns before a corresponding write starts and that there is exactly one value per value. Then we build the corresponding reverse and forward intervals for each value. We then consider each pair of forward intervals and each pair of forward and reverse intervals. We apply the following checks:

- **Forward + Forward Intervals** - the intervals cannot intersect at all. Since each forward interval forces the value to be in memory, any intersection would mean that two different values are written to the same memory at the same time.
- **Forward + Reverse Intervals** - the reverse interval cannot be contained in the forward interval. There is always at least one linearization point in the reverse interval, but if the forward interval contains it, then we are certain that a different value is written to memory, so only operations on that value can be linearized.

2.4 True CAS

2.4.1 Basic Checks and Sorting

True CAS is a little more complicated. We cannot simply replace it by a write and a read, since we need to additionally make sure that these operations execute strictly one after another. We will incorporate true CASes in the model of blocks and intervals to be able to reason about their impact on the order. First, let us consider some trivial cases that prevent the history from being linearizable. We cannot have a linearization if we have more than one CAS with the same compare argument, i.e. we cannot have `1->2` and `1->3` anywhere in the execution, because once either of these CASes is linearized, the value of 1 is lost and cannot be used again. We also cannot have loops of any length, e.g. `1->2`, `2->3`, `3->1` is not allowed. Having loops would make it difficult to reason about any kind of order, and since any loop makes the history non-linearizable, we can exclude this case directly. To find a loop we could consider each true CAS as two connected nodes of two values, and use any of the well-known cycle detection algorithms.

After these two checks, reasoning about ordering becomes easier, but we need to first understand in which order true CASes will be linearized among themselves. For this we can once again rely on the graph abstraction and use a topological search shown in figure 7.

function TOPOLOGICAL_TRUE_CAS_SORT($true\_cases$)
    $graph \leftarrow \{c.compare : c.swap \mid c \in true\_cases\}$
    $val\_order \leftarrow$ `[]`
    $visited \leftarrow$ `{}`
    function DFS($node$)
        if $node$ **in** $visited$ **then**
            return
        $visited.add(node)$
        if $node$ **in** $graph$ **then**
            $dfs(graph[node])$
        $val\_order.add(node)$
    for $node$ **in** $graph$ **do**
        $dfs(node)$
    Sort $true\_cases$ in-place in the reversed order of appearance in val_order

Figure 7: Topological True CAS sort

2.4.2 Block Intergration

With true CASes sorted, we can observe that they form "chains" forcing a certain order between the values. Unfortunately, we cannot just take these chains and convert them into blocks. Consider the history given in figure 8. Two possible linearizations are `write(1), 1->3, write(2), 2->4` and `write(2), 2->4, write(1), 1->3`. However, defining the block is problematic, because 3 must follow directly after 1, but the group $(1, 3)$ as a whole can be linearized before or after the group $(2, 4)$. This is why when including true CASes in block, they are always given in a tuple specifying the whole chain. Therefore, the history in figure 8 corresponds to the blocks `[[(1,3), (2,4)]]`.

2.4.3 Intra- and inter- Group Checks

To check whether the entire history is linearizable, we can first check if the the group alone is linearizable (intragroup check) and then if we consider the entire chain to be a single value, whether it can fit it with other operations with an interval check (intergroup check). The pseudocode for these checks is shown in figure 9.

Notice that unlike in the inter-check, we cannot simply run the interval check on the intragroup intervals, because it checks that the intervals can be linearized in any order, but we need to check that they can be linearized in the order given by the true CASes. Instead, we can use a more efficient linear check that traverses each consecutive pair of true CASes, i.e. 1st and 2nd, 2nd and 3rd, etc. and checks whether the earliest time when we can fully linearize the first value is before the latest time when we can linearize the second value.

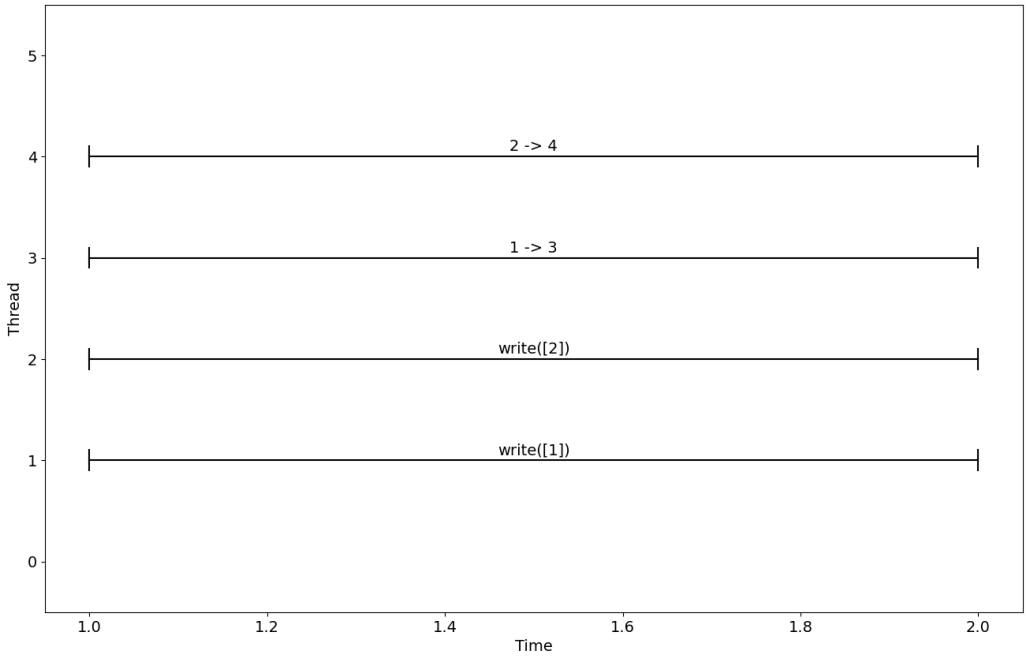

Figure 8: History corresponding to blocks `[[(1,3), (2,4)]]`

```
1:  function IS_VALID_ORDER(intervals, order)
2:      for (i1, i2) in consecutive pairs of order do
3:          earliest_lin_i1 ← if intervals.reversed then intervals[i1].start else intervals[i1].end
4:          latest_lin_i2 ← if intervals.reversed then intervals[i2].end else intervals[i2].start
5:          if earliest_lin_i1 > latest_lin_i2 then
6:              return False
7:      return True
8:
9:  function INTRA_GROUP_CHECK(sort_by_val, true_cas_val_groups)
10:     for group in true_cas_val_groups do
11:         ops_in_group ← {sort_by_val[val] | val ∈ group}
12:         intra_group_intervals ← make_intervals(ops_in_group)
13:         if is_valid_order(intra_group_intervals, order) is not True then
14:             return False
15:     return True
16:
17: function INTER_GROUP_CHECK(sort_by_val, true_cas_val_groups)
18:     sort_by_val ← deep copy of sort_by_val
19:     for val_group ← true_cas_val_groups do
20:         val ← first element of val_group
21:         for other_val ← other elements of val_group do
22:             sort_by_val[val].extend(sort_by_val[other_val])
23:             sort_by_val.delete(other_val)
24:     intervals ← make_intervals(sort_by_val)
25:     return interval_check(intervals)
```

Figure 9: Intra- and inter-group checks

2.5 False CAS

2.5.1 Block definition

The false CAS is the hardest operation to linearize. It can be linearized with all but one value, so the impact on the linearization order is not as strong as with the true CAS. Unlike the true CASes that we treated almost independently of each other, a false CAS can have a slight change the order of values at the beginning of the program, which will only have affects later on. It also breaks the logic of intervals, because we do not know ahead of time, which value will linearize it with, and so we cannot build the proper intervals. Somehow we need to find all available values that can linearize a false CAS, then add the false CAS to the corresponding value group, build the intervals and run our usual checks.

The values that can linearize a false CAS are called **resolvers**. To get a list of resolvers for each false CAS, we need to first understand how each false CAS affects the order of execution and encode this information in the blocks. Each block represents values that can be linearized in any order. First observation we can make is that two values with a forward interval can never be placed in the same block. This is because two forward intervals cannot intersect, so the one that starts first must also be linearized first. Secondly, a reverse and a forward interval can put in the same block if and only if, the reverse interval fully contains the forward interval. Otherwise, if the reverse interval does not contain the start of the forward, the forward must be linearized first and vice versa if the reverse interval does not contain the end of the forward, the reverse must be linearized first. This property is also transitive - if there is a forward interval and two reverse intervals that contain it, then not only can the order of linearization between the forward and reverse intervals be switched, but all three operations can be linearized in any order. Consider the example in figure 10. Interval 2 fully contains interval 1, so they get placed in the same block. The interval 2 does not need to contain all the operations - `read(1)` ends outside of interval 1, but the interval 2 ends at the call of the read, so the condition is satisfied. On the other hand, interval 4 does not contain the interval 3 fully, so they get placed in different blocks.

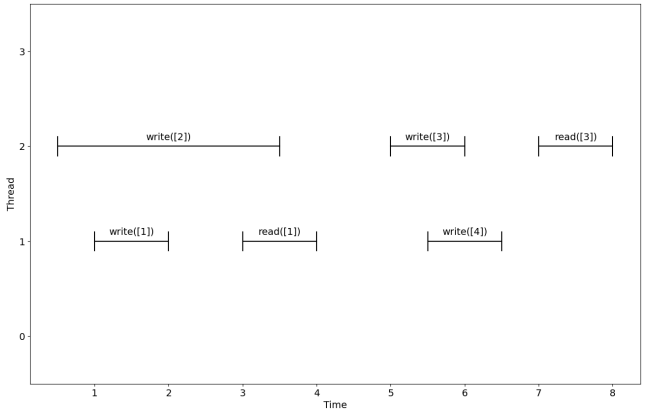

Figure 10: Blocks `[[1,2], [4], [3]]`

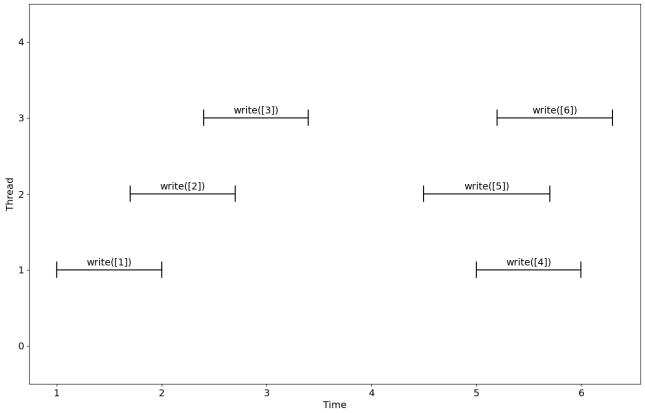

Figure 11: Blocks `[[1,2], [2,3], [4,5,6]]`

function FORWARD_BLOCKS($intervals$)
  $blocks \leftarrow$ `[]`
  for $forward\_val, forward\_interval \leftarrow$ forward intervals of $intervals$ **do**
    $block \leftarrow [forward\_val]$
    for $reverse\_val, reverse\_interval \leftarrow$ reverse intervals of $intervals$ **do**
      if $forward\_interval \subseteq reverse\_interval$ **then**
        $block.append(reverse\_val)$
        Mark $reverse\_val$ ▷ This means this value cannot form a block on its own. See below.
    $blocks.append(block)$
  return $blocks$

Figure 12: Forward blocks algorithm

Finally, two reverse intervals can be placed in the same block if and only if they intersect. On top of that, their intersection

cannot itself be contained inside another forward interval, because as we know no linearization points are possible inside a forward interval. Unfortunately, this property is not transitive (see figure 11). Interval 1 intersects with interval 2, and 2 intersects with 3, but 1 does not intersect with 3. This means that the order of linearization between 1 and 3 cannot be switched. On the other hand, intervals 4, 5, 6 all intersect among themselves, so they can be placed in the same block. Finding maximal groups of reverse intervals is similar to finding maximal cliques in a graph. Unfortunately, this problem is NP-complete [6], so we cannot use this approach. Instead, we pick an initial reverse interval with the earliest return time and pair it with all other reversed intervals in the order of their returns. We keep track of which of them our initial interval intersects. As before, the intersection cannot occur inside a forward interval. After we are done checking for intersections, we can make a block from the initial interval and all the intervals that intersect with it. We mark all the intervals in a block and delete the initial interval from the list of intervals as a whole. This is because we already found all the blocks the initial interval can potentially belong to, and there is no need to check it again. We repeat this process for a new initial interval, which ends after the previous one. Finally, we can never add a block if it is only made of marked intervals. Intervals get marked when they are added to the block in this step or in the previous step when they are joined with forward intervals. Furthermore, a reverse interval can be part of both forward and reverse blocks. The pseudocode for making reverse blocks is given in figure 13.

function REVERSE_BLOCKS($intervals$)
    $blocks \leftarrow$ [ ]
    $reverse\_intervals \leftarrow$ Sort reverse intervals by return time
    for rev_val1, rev_i1 **in** $reverse\_intervals$ **do**
        $block \leftarrow [rev\_val1]$
        for rev_val2, rev_i2 **in** $reverse\_intervals$ **do**
            $intersection \leftarrow rev\_i1 \cap rev\_i2$
            if $intersection \neq \emptyset$ **and** $intersection \not\subseteq$ any forward interval **then**
                $block.append(reverse\_val)$
        if $block \not\subseteq$ marked intervals **then**
            $blocks.append(block)$
            Mark all intervals in $block$
    return $blocks$

Figure 13: Reverse blocks algorithm

Before we make forward and reverse intervals, we need to incorporate true CAS chains shown in figure 8. We do it similarly to intergroup checks - by merging all values in a true CAS chain into a single interval. After that we can just treat the chain as a single value. After the blocks are made, we can sort them in order of the earliest end point of any intervals in the block.

function MAKE_BLOCKS($intervals, true\_cas\_val\_groups$)
    $blocks \leftarrow$ [ ]
    ▷ Step 1. Merge true CAS groups
    $merged\_intervals \leftarrow merge\_true\_cas\_groups(intervals, true\_cas\_val\_groups)$

    ▷ Step 2.1 Skeleton. All forward intervals are placed in different blocks
    ▷ Step 2.2 Reversed intervals that contain a forward interval get added to the block and get marked.
    $blocks.extend(forward\_blocks(merged\_intervals))$

    ▷ Step 3. "Clique problem" - find maximal groups of reverse intervals that intersect
    $blocks.extend(reverse\_blocks(merged\_intervals))$

    Sort $blocks$ by the return time of the earliest interval in the block
    return $blocks$

Figure 14: Block-making algorithm

2.5.2 False CAS Resolvers

Blocks encode all the branching points in the program. We now have to refine them further with false CASes. We first treat each false CAS independently and gather all writes that can resolve it. For that we traverse each block one by one and collect a list of writes that start before false CAS returns, as the writes that get called later cannot possibly resolve the CAS. All collected writes get added to the resolving pool of values. We also need a mechanism to remove values from the pool, when the write value gets overridden before reaching the CAS and cannot possibly resolve it. Since we are working under assumption that CASes and writes cannot overlap, an easy solution is once any of the operations on any of the values in the current block returns, we clear the pool of values entirely. We do this step before adding the writes from the current block, so we do not clear it as well. The idea is that if any operation returns, this value must be linearized thus overriding all the previous writes, so we must clear them. If we did not have the assumption of non-intersection, we could have the history shown in figure 15. The corresponding blocks are `[[1,2], [2,3]]`, but if we clear the pool after entering the second block, we would clear 1 from the first block, even though it is still possible to resolve the CAS with it.

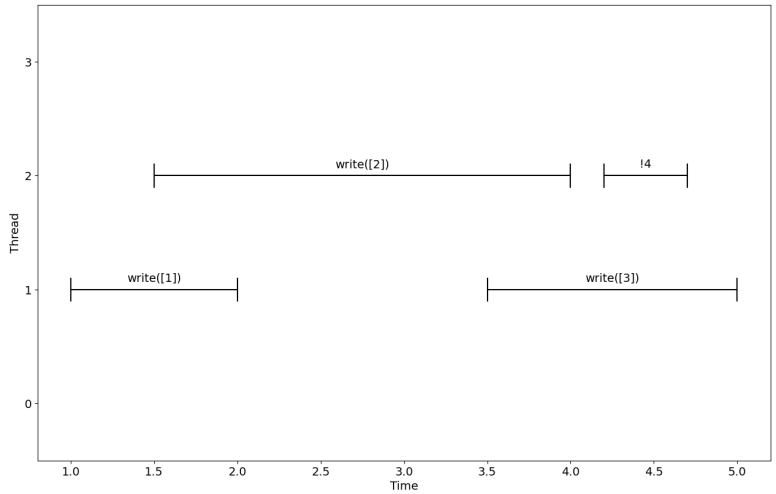

Figure 15: The clearing condition fails without the non-intersection assumption.

We also need to incorporate true CAS chains into the available writes. We cannot simply treat them as individual values, because at most one value per chain can be written at any moment of time. The false CAS can intersect multiple values from the chain. The solution is to traverse the true CAS values in order until we reach the first one that can theoretically reach the false CAS, i.e. the one that is not force-overridden by another value in the chain. From that value onwards, we keep adding values to the pool if again the corresponding true CAS begins before false CAS ends. From there on, the value is treated identically to a regular value and the same clearing condition applies.

```
function BLOCK_TRAVERSAL(blocks)
    resolvers ← { }
    for false_cas in false_cas_vals do
        available_writes ← []
        for block in blocks do
            if any operation in the current block returns before false CAS starts then
                available_writes ← []
            candidates ← All writes in the current block that start before false CAS returns
            for chain in true CAS chains in the current block do
                first_possible_true_cas ← latest value in the chain, such that the earliest operation
                                          on that value starts before false CAS returns
                candidates.extend(all values in the chain from first_possible_true_cas onwards,
                                  that start before false CAS returns)
            available_writes.extend(candidates)
            earliest_forward_lin_t ← latest end of a forward interval in the current block
            earliest_reverse_lin_t ← latest start of a reverse interval in the current block
            ▷ This is the earliest we can linearize the entire block
            earliest_block_lin_t ← max(earliest_forward_lin_t, earliest_reverse_lin_t)
            if false_cas ends before earliest_block_lin_t then
                break
        available_writes.discard(false_cas.compare)
        resolvers[false_cas] ← available_writes
    return resolvers
```

Figure 16: Initial false CAS resolvers

2.5.3 False CAS Propogation and Mutually-exclusive False CASes

After we have gathered the resolvers for each false CAS independently, we need to refine them further, because in reality false CASes can affect each other. Luckily, under the assumption that false CASes cannot overlap with writes, we only need to do two more checks to fix it. First, a false CAS can be placed either strictly before or strictly after its corresponding write. If it is placed before, it is trivially linearizable with any write and does not have any impact on other false CASes. Otherwise, if it is executed after its corresponding write, the value it compares against cannot be a resolver for any other false CAS that follows it. In a sense, the false CAS is propagated throughout the program, blocking all the values that it compares against. This is called false CAS propagation.

```
function PROPAGATE_FALSE_CASES(blocks, resolvers)
    blocked_val ← All values of false CASes that start after their corresponding write ends
    for false_cas in false_cases do
        for val in resolvers[false_cas] do
            if val in blocked_val and false_cas starts after the false cas blocking val ends then
                resolvers[false_cas].discard(val)
```

Figure 17: False CAS value propagation algorithm

Second, we can have a history shown in figure 18, where both false CASes can be linearized on their own, but not together. They are called mutually-exclusive false CASes. Since under our assumption, false CASes cannot overlap with writes, writes in some sense partition the program into groups where false CASes are contained. Within these groups there can only be one value written to memory, but do not know which one. However, since only one value is written to memory, there needs to exist a value that linearizes all false CASes. In figure 18, the false CASes resolvers are `[1]` and `[2]`, for `!2` and `!1`, respectively. Since their interaction is empty, the history is not linearizable. Finally, if the intersection is non-empty in every block, all false CASes and the history as a whole is linearizable.

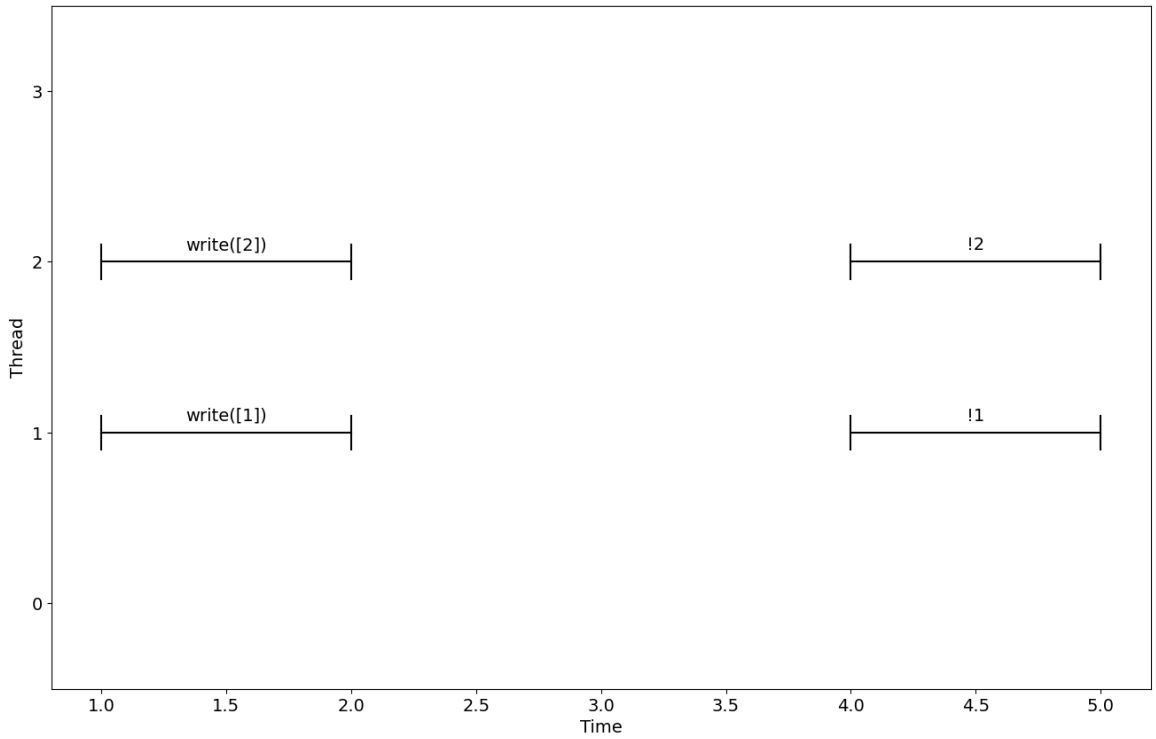

Figure 18: Mutually-exclusive false CASes.

function FALSE_CAS_EXCLUSION_CHECK($blocks, resolvers$)
    $false\_cas\_exclusion\_group\_no \leftarrow \{false\_cas :$ # writes that return before it$\}$
    $false_cas_exclusion_group \leftarrow \{$group number : all false CASes with that group number$\}$
    for $group\_no, false\_cases$ **in** $false\_cas\_exclusion\_group$ **do**
        for $false\_cas$ **in** $false\_cases$ **do**
            $resolver[false\_cas] \leftarrow$ intersection of all resolvers of false CASes in the same group
            if $resolver[false\_cas]$ is empty **then**
                return False
    return True

Figure 19: The check is based on the assumption that false CASes and writes do not intersect and prevents mutually-exclusive false CASes

2.6 Ordering

Finally, after passing all the checks, we need to order the operations. Each false CAS gets placed with other operations on one of its resolver values. We then have to regenerate the intervals and blocks. We traverse each block (from left to right) and each value in a block (in any order, unless we entered a true CAS chain, where we again traverse from left to right). Within each value we always place the write or true CAS first and all the reads in any legal order. If the value we linearize is part of a true CAS chain, we finish traversing the chain before moving on to the next value. We also make sure to not count a true CAS twice.

An example of what the final output of the algorithm on a more involved history with 5 threads and 16 operations looks like is given in figure 20 for demonstration.

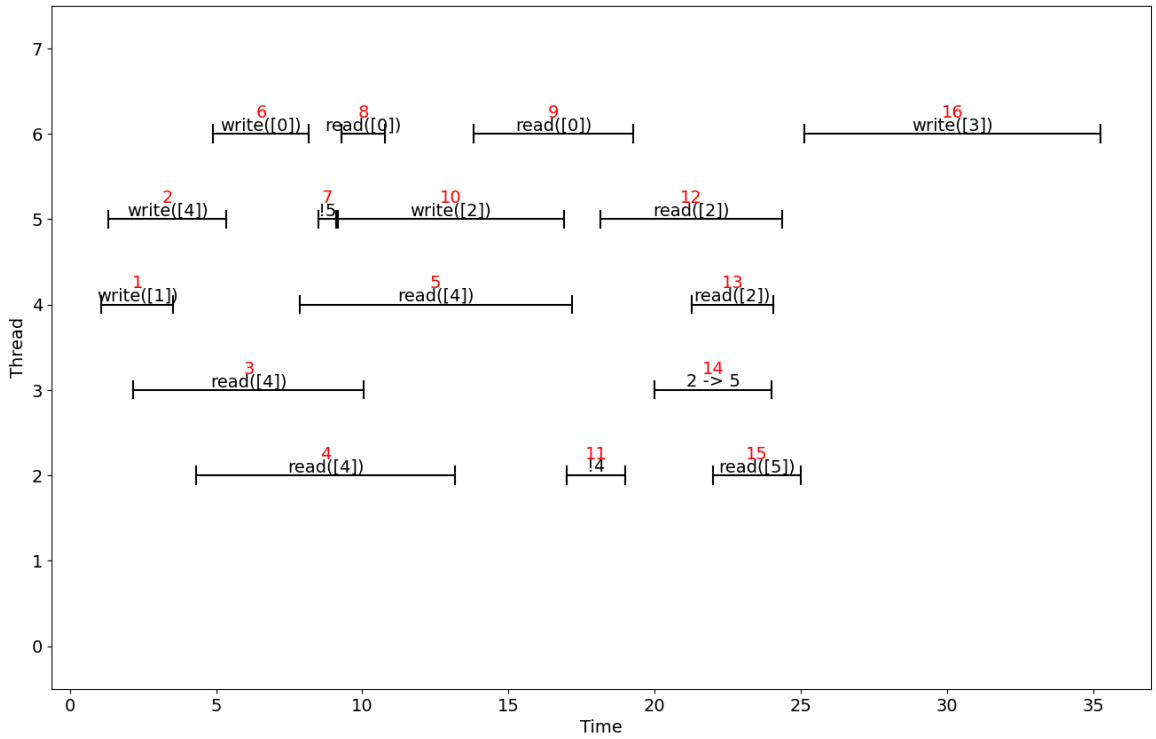

Figure 20: An example of ordering of a complicated involved history with 16 operations.

3 Evaluation and Results

3.1 Test Generation

Tests are generated automatically with a configurable set of parameters:

- $total$ - Total number of tests.
- n - Number of threads.
- m - Number of operations.
- k - Number of values.
- ops - Set of operations to be used.
- min/max_dur - Minimum and maximum duration of an operation.
- min/max_offset - Minimum and maximum offset between operations.
- $sucpercent$ - Percentage of successful cases.

For each test, we generate m random operations, assign each a random thread (between 1 and n) and a random set of arguments (each between 1 and k). The operations then get a random time offset from the previous operation on the same thread and a random duration. We also add random noise (between -0.1 and 0.1) to the call and return times of an operation to make sure that no two operations have the same start or end time, as that would create an ambiguity in the linearization. We also always add the write/true CAS before the read, as otherwise the algorithm almost instantly returns that the history is trivially non-linearizable. This does not mean that a write is always placed first, it only means that a write is added to the history first, it can be placed on a thread with more operations and a read added later can even return earlier than a write starts on a different thread. Lastly, we check if any false CASes intersect with any of the writes as that is one of the critical assumptions of the algorithm. The entire history gets rejected if this condition is not satisfied.

Each history is then checked by the brute force algorithm and stored along with a boolean indicating whether the history is linearizable or not. We repeat this process for $total$ times, making sure that exactly $sucpercent$ of the tests are linearizable. Usually between 15% and 30% of all histories are linearizable. Linearizable histories are much more computationally expensive to generate, especially with more operations, values and threads where they become exponentially rarer, however they are still included, as they "force" both algorithms to traverse the entire execution and are essential for an unbiased speed comparison. That said, some of the bigger tests take multiple days to generate, so they have both fewer linearizable histories and the total number of histories as a whole. The tests are then stored in a pickled python list of tuples (history, boolean).

3.2 Performance evaluation

The tests are then run on the Python implementation of the algorithm[1] and the results are compared to the expected results. The tests are run on a machine with an 11th Gen 8-core Intel® Core™ i5-1135G7 @ 2.40GHz CPU and 16GB of RAM. The speed of the algorithm is measured using the tqdm module in python, which displays how many tests are passed per second. The performance results are shown in figure 1.

Configuration	Brute-force Algorithm	Polynomial Algorithm	Configuration	Brute-force Algorithm	Polynomial Algorithm
Test group 1: writes and reads only					
2M tests. 30% success 1 thread. 7 ops. 3 vals.	4950 t/s	9634 t/s	1M tests. 30% success 3 threads. 10 ops. 4 vals.	687 t/s	8550 t/s
500k tests. 20% success 4 threads. 12 ops. 4 vals.	281 t/s	9201 t/s	200k tests. 20% success 5 threads. 12 ops. 4 vals.	175 t/s	10553 t/s
50k tests. 15% success 6 threads. 12 ops. 5 vals.	47 t/s	10550 t/s	25k tests. 15% success 7 threads. 15 ops. 5 vals.	19 t/s	9147 t/s
Test group 2: writes, reads and true CASes					
2M tests. 30% success 1 thread. 7 ops. 3 vals.	5552 t/s	11779 t/s	1M tests. 30% success 3 threads. 10 ops. 4 vals.	916 t/s	9523 t/s
500k tests. 20% success 4 threads. 12 ops. 4 vals.	783 t/s	22041 t/s	200k tests. 20% success 5 threads. 12 ops. 4 vals.	483 t/s	20808 t/s

[1] https://github.com/Pylyr/thesis-concurrency

Configuration	Brute-force Algorithm	Polynomial Algorithm	Configuration	Brute-force Algorithm	Polynomial Algorithm
50k tests. 15% success 6 threads. 12 ops. 5 vals.	238 t/s	36964 t/s	25k tests. 15% success 7 threads. 15 ops. 5 vals.	215 t/s	79588 t/s
Test group 3: writes, reads, true CASes and false CASes					
2M tests. 30% success 1 thread. 7 ops. 3 vals.	13735 t/s	39106 t/s	1M tests. 30% success 3 threads. 10 ops. 4 vals.	3799 t/s	111910 t/s
500k tests. 20% success 4 threads. 12 ops. 4 vals.	1546 t/s	13335 t/s	200k tests. 15% success 5 threads. 12 ops. 4 vals.	969 t/s	116712 t/s
50k tests. 15% success 6 threads. 12 ops. 5 vals.	218 t/s	130497 t/s	25k tests. 15% success 7 threads. 15 ops. 5 vals.	56 t/s	123499 t/s

Table 1: Performance comparison of the algorithm against the brute-force benchmark over different configurations (shown in tests per second)

Clearly, the algorithm is much more efficient than a brute-force solution. The difference is especially noticeable on histories with multiple threads and more operations. Interestingly, both algorithms perform almost an order of magnitude faster in test groups 2 and 3, as suppose to group 1 with only writes and reads. In case of a brute force algorithm, this can be explained by true CASes significantly reducing legal branching (i.e. branching that does not violate the real-time order) of the linearization. The brute-force algorithm does not have optimizations, however, if there is no way to linearize a particular true CAS (the needed value is unavailable by any means), the exploration will not continue past the true CAS, since all possible execution permutations till this point will fail. The polynomial algorithm has numerous, computationally cheap checks in place that reject a non-linearizable execution. On top of that, we can see that as the number of threads and operations goes up, the speed of the brute-force decays exponentially, but the polynomial algorithm becomes even faster! The number of operations does not grow linearly with the number of threads, so even though there are more threads, since it is very computationally expensive to generate tests, as they have to be generated with the brute force algorithm, each thread has fewer operations. This means the executions get shorter in terms of real time, and the value intervals inevitably become shorter. This means on average the intervals are shorter, and the algorithm has fewer checks to make.

4 Applications

As explained in the introduction, the problem of linearizing arbitrary operations is NP-complete. However, even with four operations we were able to process in this paper, we can still linearize some concurrent data structures. We can check linearizability of a lot of append-only data structures, or data structure with a bounded number of deletions. For example, consider a concurrent map supporting the following methods[2]:

- $put(k, v)$ - Put the value v in the map under the key k.

[2] A reduced version of Java's `ConcurrentHashMap`. See here for more details.

- $get(k)$ - Get the value under the key k.
- $replace(k, old, new)$ - Replace the value under the key k with new if the current value is old.
- $remove(k)$ - Remove the value under the key k.

Since linearizability is a compositional property, we can check linearizability of each key in the map separately. The put method is equivalent to a write, the get method is equivalent to a read, the $replace$ method is equivalent to a CAS. The deletion is more problematic. If we bound the number of deletions by a constant, we can assign a different special "null" value to each deletion on a given cell, as well as special value to the initial empty value. We can then append to every $get, remove$ and $replace$ a series of false CASes that will check that the value is neither deleted nor empty. We cannot re-use the same value across all deletions, because one of the assumptions of the algorithm is that each value is written once, so each "write".

An example of an append-only data structure is a concurrent shared log. A shared log is a data structure that only supports 2 operations:

- $append(v)$ - Appends the value v to the end of the log.
- $get()$ - Reads the entire log.

This kind of data structure is very similar to how blockchain transactions are kept track of. Likewise, $append$ here represents a write. Even though get reads the entire log, we can still convert it into a read. Since there is no way to somehow change the order of values, but only append more at the end, the values must have the same order across all get's and thus form prefixes of each other. We thus have to run a trivial check that all arguments of get form prefixes of one another. Alternatively, we can impose an order of writes using true CAS chains. After this check, we can substitute each get with a read on the last value. An example of a shared log history is shown in figure 21 and its conversion into

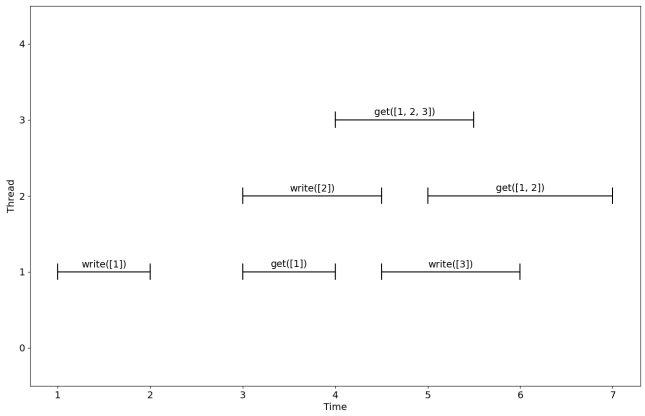

Figure 21: Shared log history

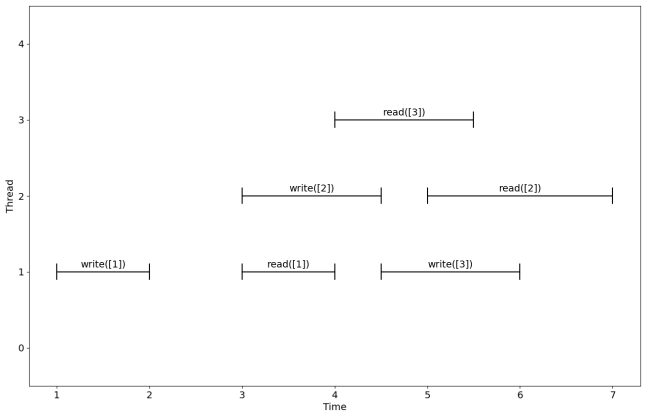

Figure 22: Shared log history after conversion

5 Conclusion

In conclusion, the algorithm successfully linearizes arbitrary histories of given operation under the given assumptions. The algorithm performs significantly better than a brute-force solution by all metrics. Further work on the algorithm could potentially allow for integration of more operations. For example, linearizing a binary switch, an operation that switches a value X by a value Y and vice versa could allow for linearizing concurrent maps and sets with unlimited deletion and replacement. Another possible extension is to adapt the algorithm for network context when operations need to be linearized not only over multiple threads, but also across different machines with a communication delay between them. Lastly, even though the algorithm can currently linearize tens of thousands of histories per second, the algorithm is still at least quadratic in the number of operations. The algorithm can theoretically be optimized to run in linear time allowing for linearization of much longer histories.

6 References

[1] Ahmed Bouajjani, Constantin Enea, and Chao Wang. Checking Linearizability of Concurrent Priority Queues. In Roland Meyer and Uwe Nestmann, editors, *28th International Conference on Concurrency Theory (CONCUR 2017)*, volume 85 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:16, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[2] Peter Druschel. Semaphore implementation. Operating Systems (COMP 421): Lecture Notes. Lecture 7.

[3] Danny Hendler. *Non-Blocking Algorithms*, pages 1321–1329. Springer US, Boston, MA, 2011.

[4] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[5] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, jul 1990.

[6] Gavin Lowe. Testing for linearizability. *Concurrency and Computation: Practice and Experience*, 29(4), 2017.

[7] Hermann Schweizer, Maciej Besta, and Torsten Hoefler. Evaluating the cost of atomic operations on modern architectures. 2020.

[8] Tangliu Wen. A simple way to verify linearizability of concurrent stacks, 2021.