ABSTRACT

Lockless concurrent programming brings new challenges to the field of program verification. These lockless programs require methods such as compare-and-swap and memory fences to ensure correctness. However, their unpredictable behaviour in combination with these methods complicates verifying such algorithms. We use linearisation points[3], i.e. the points in time when the state of the system changes, to abstract these methods. By deducing the possible ordering of these linearisation points we can predict the possible states of the system and draw conclusions about the scrutinised algorithms. This paper uses linearisation points and the control flow of the program to create an informal proof of the Lace[10] algorithm, which implements a work-stealing method for concurrent programs.

Keywords

Lace, linearisation points, concurrent programming, informal proof

1. INTRODUCTION

Modern day processors’ multi-core architecture has increased the demand for concurrent programming to fully utilise the processing power. These concurrent programs need to be proven correct to ensure the desired execution. Several methods for proving programs correct exist, such as separation logic[7] and first-order logic[8].

However, due to the unpredictable nature of lockless concurrent algorithms, results might be unexpected. Thus lockless concurrent algorithms use atomic methods such as compare-and-swap and structures such as memory fences to ensure intended execution. The properties of these operations play a vital role in proving concurrent programs correct. As demonstrated in this paper, concurrent algorithms such as Lace depend on these properties to ensure correctness. Other examples include implementations of lock-free linked lists[9] and nonblocking algorithms for shared queues[6]. We show Lace uses its compare-and-swap and memory fences to ensure correct working, without them the proof would be incomplete.

This paper tries to prove several properties of the Lace algorithm[10] to illustrate a method of proving concurrent programs. Lace is a concurrent algorithm for thread scheduling among processors. Its use of a memory fence, compare-and-swap system call and both concurrent and sequential components make it an ideal algorithm to prove correct.

In this paper, we will prove the Lace algorithm correct by giving an informal proof based on linearisation points and the order in which they appear. First we define what properties of Lace need to be proven to establish correctness. Thereafter, the linearisation points are manually constructed and a control flow diagram of Lace is created. All Lemmas and Invariants used in the proof are based on these linearisation points and the control flow of these points. With the Invariants and Lemmas a comprehensive proof of the Lace algorithm is constructed.

1.1 Lace

Lace is a work stealing algorithm[2] which uses a compare-and-swap operation to steal tasks[10]. Work stealing algorithms dynamically execute multi-threaded computations. In this context, each thread is called a worker. These workers are able to spawn new computational tasks and execute them. When a worker has got no work of its own it attempts to steal a task from other workers, thereby becoming a thieving thread. Each worker in Lace has its own deque, i.e. double-ended queue. A deque can be accessed from both its head and its tail, the process itself operates on the head, whereas stealing threads operate on the tail. Besides the deque, a worker also holds pointers for the head and tail of the deque, as well as a pointer for a split point. This split point indicates whether a thieving thread can steal a task or not, i.e. what part of the deque is shared and what part is private. When the tail is still lower than the split point, thief processes may not steal the task. Conversely, when the tail increases beyond the split point thief processes may not steal the task. Figure 1 illustrates the workings of this deque with its head, tail, and split variables. The deque is presented as an array of slots which may be empty or contain tasks. Slots

![Figure 1: Example on manipulation of deque variables.](image)
containing a task are represented with a •. We say a task is stolen when it is located left of the tail, i.e. tail points to the next task to be stolen. The bar before the split point indicates that tasks can only be stolen left of the bar or split point. In the first deque, tail is still to the left of split, we say tail < split, and thus tasks can be stolen. In the second deque, tail = split, meaning that tasks may not be stolen. However, in Lace a thieving process can request to move the split point so more tasks can be stolen. This grow_shared request issues the deque process to increase the split point if possible, which is shown in the bottom deque. When the head wants to execute a task, it will pop it off the head of the deque and point its head pointer to the task before that. However, if head = split it cannot pop a task, therefore it will try to shrink the shared part of the deque. If this is possible, i.e. if tail < split, split will be moved to the left and head can pop more tasks. The full algorithm of Lace, as used in the paper can be found in the appendix.

2. PRELIMINARIES

2.1 System

We specify the assumed memory model to illustrate the possibility of reordering loads before stores within the system. This introduces several difficulties in proving the algorithm, but is necessary to ensure correct execution on real systems. This paper assumes the algorithm runs on a shared memory system with the x86 memory model. This memory model allows the reordering of loads before stores, i.e., write operations are buffered before they are stored in memory. Thus threads might read old values, even though they have been written by other threads. These writes may not have become globally visible yet, because they are still buffered. Memory fences are used to flush these write buffers and make changes globally visible. Memory stores are immediately visible to threads that wrote them, hence they do not require memory fences.

2.2 Compare-and-swap

A compare-and-swap (cas) ensures thread safety by simulating an atomic memory operation. The cas operation takes three parameters as input, namely a variable, an expected value, and a replacement value. The cas operation compares the value of variable to the expected value. When those values are equal, the replacement value replaces the current value of variable and cas returns true. If they are not equal, variable retains its current value and cas returns false. Because of its atomicity, cas ensures only one thread executes successfully when multiple threads invoke cas with the same valid parameters. cas is used for concurrent programming as it ensures thread safety and provides feedback to executing threads on whether their operation succeeded or not.

2.3 Linearisation points

An important aspect of proving concurrent algorithms is defining linearisation points[3]. These linearisation points are points in time where the state of the system changes. Note that linearisation points are defined as the points in time when changes are globally visible instead of when they are visible to the thread that modified a variable. In this case it is important since we assume shared memory system with the x86 memory model. When reasoning about concurrent programs, information about the state of the system is needed to predict its expected behaviour. By defining linearisation points, the order in which they occur can be varied and thereby all possible states can be derived. That is, by varying the order of linearisation points, all possible equivalent sequential programs can be derived, which in turn can be reasoned about.

2.4 Assumed properties

First of all, we have four ways of classifying a task.

**Definition 1.** A task x exists ⇔ x < head

**Definition 2.** A task x is stolen ⇔ x < tail

**Definition 3.** A task x is shared ⇔ x < split

**Definition 4.** A task x is private ⇔ x ≥ split

In proving Lace correct, this paper assumes there are N workers, each executing a single thread. Initially, all deques, as well as their head, tail, and split variables are set to 0. One worker starts with a single task in its deque, i.e. head = 1, whereas the other N-1 workers have no tasks to execute and are therefore stealing threads. This worker thread is able to spawn tasks using the spawn method in Figure 2. When a worker thread requires the result of a task, it synchronises using the sync method in Figure 2, thereby popping the method from its deque. Since a task spawns other tasks it has to synchronise all tasks before returning, therefore we assume that the number of spawns ≥ number of syncs and at the end of each task the number of spawns = number of syncs. Conversely, stealing threads try to steal tasks from working threads using the forever loop on top of Figure 2. When stealing threads have successfully stolen a task, they become working threads as well, spawning their own tasks. If a thread runs out of tasks to complete, it becomes a thieving thread executing the thieving loop. The assumptions below follow from the

```
1 thief: forever: {  
2   status , Task = steal_from_queue()  
3   if status == stolen  
4      Task.result = Task.call()  
5      Task.done = true  
6 }  
7 spawn(function , parameters) {  
8    push(Task(function , parameters))  
9 }  

10 sync() {  
11   status , Task = peek()  
12   if status is Stolen:  
13      wait until Task.done == true  
14      res = Task.result  
15      pop()  
16   else :  
17      pop()  
18      res = Task.call()  
19      return res  
20 }  
```

Figure 2: The assumed algorithm threads use to steal, spawn, and synchronise tasks respectively.

statement that methods of Lace are called by the program described above and illustrated in Figure 2.

1. One owner thread per deque.
2. 0 to N-1 stealing threads per deque.
3. **steal** is only called by thieving threads.
4. **push** and **pop** are only called by owner thread.
5. Number of \texttt{pop} calls $\leq$ number of \texttt{push} calls.
6. When finished, number of \texttt{pop} calls = number of \texttt{push} calls.
7. On initialisation all deque variables are set to zero.

2.5 Required properties
Lace requires some properties to work correctly. First of all, the variable \texttt{head} should stay within bounds so that the deque does not overflow. Figure 3 expresses this require-
ment as invariants. Furthermore, a task must be stolen only once, and executed exactly once. To prove these prop-
erties, they are written as invariants. When every task is stolen only once, it means thieving threads can only steal

tasks that are \texttt{STOLEN} and that when \texttt{tail} reduces, a task cannot
be stolen again. The former part of the property is trivial, if \texttt{tail} increases it steals an unstolen task which is
consistent with the requirement. The latter part is less
trivial, if \texttt{tail} reduces, a stolen task cannot be executed
again, therefore the stolen tasks need to be removed be-
tail is reduced. This is written as an invariant in
Figure 4. To elaborate on this invariant: either \texttt{tail in-}
tail.in > tail.old || (tail.new < tail.old &&
tail.old - tail.new <= #pop() == STOLEN)

Figure 4: Invariant: every task is stolen only once.

4. MODELING LACE
A model for the Lace algorithm is constructed from its
specifications as described in [10] and displayed in Figure
8 which can be found in the Appendix. Note the that Fig-
ure 8 projects a slightly different form of the algorithm.
However, this alternate form is used to construct a model
for Lace, since it displays the working of lace more com-
prehensively.

4.1 Linearisation Points
In Lace, the state of the system depends on variables \texttt{head},
\texttt{split}, and \texttt{tail}. The linearisation points are defined on
these variables and can be found in Table 1a, 1b, and 1d
respectively. This table is attached in the appendix. Note
that the variables used at linearisation points to change
the state of the system are read at a point before they
are used. Thus the value might have changed when the
program reaches the linearisation point, in which case it
still uses the old value. In addition to these three main
variables, the algorithm also uses a \texttt{moveSplit} and all-

stolen variable. These variables influence the behaviour
of the system by indicating whether the split point must
be moved or all public tasks are stolen respectively. There-
fore, these variables are also included in the linearisation
points, which can be found in Table 1e, 1f, and 1g.

4.1.1 Head
Variable \texttt{head} is modified in both \texttt{push} and \texttt{pop} methods.
Table 1a in the appendix gives an overview of the lineariza-
tion points and the places where the variables used in
the linearisation points are initialised. The first linearisa-
tion point is at line 13 of Lace as shown in Figure 8, where
\texttt{head} is incremented by 1. In this case the variable \texttt{head}
is read in the same line of code but does not occur simul-
taneously with the write operation. The second and third
linearisation points are at line 45 and 47 of Lace as shown
in Figure 8 respectively. In both cases, \texttt{head} is decreased
by 1 and the variable is read in the same line of code as
it is written. As with the first point, the read and write
operation do not occur simultaneously.

4.1.2 Split
In Lace, the variable \texttt{split} has two instances, namely
\texttt{split} and \texttt{o_split}. The latter instance of \texttt{split} is pri-

vate, and thus visible to the owner thread but not to thiev-
ing threads. To avoid confusion, \texttt{split} and \texttt{o_split} are
displayed in Table 1b and Table 1c in the appendix respectively. Just as explained in section 4.1.1 these tables indicate the methods, linearisation points, operations, initialisation points, and variables for the variable scrutinised, in this case the split and o_split variables. As opposed to the linearisation points of the variable head, split has linearisation points in which variables are used that are read in a different line of the algorithm. This might indicate a greater chance of the variables being altered before they are used.

4.1.3 Tail
The final variable of the deque that is important to include in the model is tail, its linearisation points can be found in Table 1d in the appendix. This variable is modified in the steal and push functions at lines 5 and 15 respectively. In steal, tail is read at line 3 whilst incremented at line 5. Because of this gap tail can be modified in the meantime by other threads. The same goes for the modification of tail in push, since read and write do not occur simultaneously.

4.1.4 Movesplit
movesplit is not part of the deque, but it indicates whether the owner thread should grow the shared part of the deque. This boolean variable is set to true at line 8 of the algorithm and is set to false at lines 16 and 25. These linearisation points can be found in Table 1e in the appendix.

4.1.5 Allstolen
As with split, the variable allstolen has two instances, namely allstolen and o_allstolen. The global boolean variable allstolen is set to false at line 17 of the algorithm and to true at line 22. The private o_allstolen is set to false and true at lines 19 and 23 respectively. The linearisation points can be found in Table 1f and 1g in the appendix.

4.2 Order of linearisation points
By assuming only one owner thread exclusively calls methods push and pop, all variables manipulated only at those methods will occur in order. Subsequently, this is true for all methods grow_shared and shrink_shared which are invoked only through push and pop. The natural order of linearisation points called by push and pop can be derived by creating a flow diagram. In this section, the order of linearisation points for push and pop are derived respectively. Note that since push and pop are exclusively called by the owner thread the order in which the linearisation points execute is not reordered. If multiple threads would execute the same linearisation points, the order can alter.

First of all, the method push has a flow diagram as depicted in Figure 6. Here the first linearisation point is at line 13, where head is increased. If o_allstolen is set, the program continues with linearisation point 15. Then, if movesplit is true, it is set to false, otherwise it continues flowing through all linearisation points down to linearisation point 19 where o_allstolen is set to false. Alternatively, when o_allstolen is false at line 14 and movesplit is true at line 20, it invokes the method grow_shared which is depicted as the right part of the flow diagram.

Second, the method pop has a flow diagram as depicted in Figure 7. This diagram is less trivial than the push flow diagram. Since the method pop first checks whether head equals 0 and returns when it does, the linearisation points are not reached, thus the first condition of head ≠ 0 is introduced. After this, it checks whether o_allstolen is set, if this is the case, it continues to decrease head and returns. Alternatively, if o_allstolen is false, it checks whether o_split == head, if so, it calls the method shrink_shared following the rhombus’ upper arrow. The method shrink_shared first checks whether the variables t and s that it read from tail and split respectively are equal. If this condition is true, it sets allstolen and o_allstolen after which it returns true and continues at pop. Subsequently, if the condition is false, it modifies split and sets a memory fence, depicted with the dashed line. After reading the tail variable into t again, the algorithm checks whether t equals s. When they are equal, the method sets the allstolen variables, and returns true, after which it continues at pop. However, when they are not equal, t and the temporary variable new_s are compared. If t > new_s, split is modified, otherwise only o_split is set to new_s. Thereafter, the method returns false and continues at pop.

From these flow diagrams, the possible sequences of linearisation points are clear. Note that the linearisation points modified by thieving threads through the method steal are not included. Since these occur at arbitrary points in the algorithm, it does not make sense to include them in a flow chart.

5. PROVING LACE

**Lemma 1.** \( o\text{-allstolen} = allstolen \) at every point \( o\text{-allstolen} \) is used.

**Proof.** Table 1f and 1g indicate that both allstolen variables only change through methods push and shrink_shared. Figure 6 states that private variable o_split is the only variable to change in between the modifications to false of the allstolen variables. Subsequently, Figure 7 states there are no linearisation points between the
modifications to true at all. No load operations of allstolen or o_allstolen occur in between either modification of the variables. The assumed memory model where loads can be reordered before stores does not influence this Lemma, since Table 1f states stealing threads only use allstolen. The owner thread does not reorder both allstolen variables, because the thread itself is the only thread modifying them.

**Lemma 2.** o_split = split at every point o_split is used.

**Proof.** Table 1b and 1c affirm that both split variables only change through methods push, grow_shared, and shrink_shared. Figure 6 indicates that private variable move_split and allstolen are modified between the linearisation points of both split variables. Furthermore, Figure 7 states that between linearisation points 23 and 24 there are no read operations for either split variable. In addition, between linearisation point 30 of split and 37 of o_split there are no read operations on split. The x86 memory model where loads can be reordered before stores is not important for this Lemma, since the method steal only uses split. Whereas the owner thread uses o_split, but loads cannot be reordered before stores for the thread that executes the stores.

**Invariant 1.** 0 ≤ head ≤ size.

**Proof.** Table 1a indicates head is only increased at method push and decreased at the method pop. This indicates there is no need to take the reordering of loads before stores into account, since only one thread operates on this variable. Also, Figure 7 affirms that this invariant holds. Between linearisation point 45 and 47 cannot decrease head to a value lower than 0. Linearisation point 13 increases head by 1. By inspection of line 11 of Lace we find head ≠ size, in combination with the assumption that head ≤ size, head < size holds. This implies head+1 ≤ size for which this invariant holds. Both linearisation point 45 and 47 decrease head by 1, and upon inspection of Figure 7 we find line 43 of head = 0 precedes both points. Thus head ≥ 0 and head ≠ 0 implies head > 0. Therefore 0 ≤ head, proving the invariant.

**Lemma 3.** tail can only increase by exactly 1 if ¬allstolen and t < s and t = tail and s = split.

**Proof.** The only linearisation point where tail is increased is at the cas operation of line 5 as stated in Table 1d. This modification only succeeds if cas succeeds, i.e. if the compare-and-swap method returns true. That is, t = tail and s = split. Furthermore, line 4 states t < s, for which neither variable can change at any point since they are local to the method. Opposed to t and s not changing, allstolen can be modified after it is checked at line 2. However, Lemma 15 explains a change in allstolen is caught before the cas operation is executed, ensuring this Lemma holds.

**Lemma 4.** The modification split = new_s = (t+s)/2 at line 30 of Lace ⇒ split.new ≤ split.old.

**Proof.** According to Figure 7, split cannot be modified in between the read operation of split. In addition, tail ≤ split for when tail is increased at line 5 it is increased to t+1 where t < split. Table 1b indicates split can only be decreased to head at line 18 of Lace. However, tail is simultaneously set to head-1, i.e. lower than tail. Moreover, Lemma 8 proves the modification at line 24 increases split. Therefore we conclude tail ≤ split. Thus line 30 states split.new = new_s = (t+s)/2 = tail+split.old/2 ≤ 2*split.old/2 ≤ split.old, since t = tail and s = split. This proves Lemma 4.

**Lemma 5.** The modification split = new_s = (t+s)/2 at line 36 of Lace ⇒ split.new ≤ split.old, where split.old is before line 30 of Lace.

**Proof.** Figure 7 states line 30 precedes line 36 in Lace. Lemma 4 indicates split decreases at line 30, however, the memory fence of line 31 making this modification globally visible might be too late so that tail has increased beyond the new split. As described in the proof of Lemma 4, tail cannot increase further than split.old. Therefore the newly read value of tail at line 32 ensures tail.new ≤ split, where split is the modification after line 30. Now the modification of line 36 can be interpreted as split.new = (t+s)/2 ≤ 2*split.old/2 leq split.old, proving Lemma 5.

**Lemma 6.** The lowest value of split, as modified by lines 30 and 36 is globally visible after the modification of line 36.

**Proof.** Figure 7 affirms line 30 is reached before line 36 and a memory fence is executed in between the executions of both lines. It is left to prove line 36 does not decrease the value of tail, with respect to the modification of line 30. The operations of both linearisation points are equal, they are set to new_s where new_s = (t+s)/2. In the latter modification the value of split is read from tail at line 32, i.e. after the memory fence, all other variables are equal. Table 1d states, tail is only modified at lines 5 and 15 by steal and push respectively. The only modification is the increase of tail, since push is only called from within the owner thread, and not in between lines 30 and 36. Therefore, t.new ≥ t.old which means split.30 = (t.old+split.old)/2 ≤ (t.new+split.old)/2.

**Lemma 7.** shrink_shared returns false and tail ≤ head ⇒ tail ≤ split < head.

**Proof.** Figure 7 affirms shrink_shared (represented by linearisation points 30, 36, and 37) is only called if o_split = split = head (Lemma 2). Furthermore, the Figure states shrink_shared only returns false if t != s, i.e. tail ≠ head, since split = head and there occur no modifications to either variables in the meantime. Lemmas 4 and 5 state tail can only decrease at these points. Combined with the assumption that split ≠ head it follows that tail ≤ split < head.

**Lemma 8.** grow_shared is only called if ¬allstolen.

**Proof.** Figure 7 affirms grow_shared, represented by linearisation point 23, can only be reached if ¬allstolen. Furthermore, it shows no linearisation points modifying o_allstolen occur after o_allstolen is checked. Table 1f and 1g state both variables can only be changed by the owner thread. In combination with Lemma 1 we conclude that grow_shared is only called if ¬allstolen.

**Lemma 9.** ¬allstolen ⇒ tail ≤ split ≤ head.

**Proof.** This paper assumes that the deque is initialised with tail = split = head = 0 and ¬allstolen, hence,
the lemma holds at at initialisation. Table 1f indicates allstolen is only reset at linearisation point 17. Figure 6 shows linearisation points 15 of tail and split must precede the reset of allstolen, which set tail and split to head-1 and head respectively. At this point the lemma still holds, and modifications of head, tail, and split need be scrutinised. First of all we note that tail might be modified in between linearisation points 15 and linearisation point 17. However, tail will never increase beyond split because of the condition t < s described in Lemma 3.

Table 1d demonstrates variable tail is modified at linearisation point 5 and 15. Lemma 3 states that for linearisation point 5 to execute t < a and t == tail and s = split. Therefore, t < split and tail is set to t+1, i.e. tail.new = t+1 ≤ split. Figure 6 expounds linearisation point 15 cannot be reached if ¬ allstolen, because Lemma 1 states o_allstolen = allstolen at load operations, this linearisation point cannot be reached when ¬allstolen.

Table 1b indicates split is modified at linearisation points 15, 23, 30, and 36. As with linearisation point 15 of tail, linearisation point 15 is unreachable when ¬allstolen. Linearisation point 23 can be reached, Lemma 10 states that after this linearisation point split ≤ head still holds, using the assumption of this Lemma, that ¬allstolen ⇒ split ≤ head. Linearisation point 30 may violate the lemma, since tail can grow to the value of split.old and Lemma 4 expounds split.new ≤ split.old after the modification of linearisation point 30. However linearisation point 36 restores the value split so that tail ≤ split holds. If tail grows beyond the value of split.tmp as split is set to at line 30, the newly read variable t, which we call tail.new, is larger than (tail.old+split.old)/2, i.e. the value of split.tmp. If the newly read variable t, which we call tail.new, is larger than (tail.old + split.old)/2, i.e. the value of split after linearisation point 30, tail violates this lemma. According to Figure 7, this initiates linearisation point 36 which ensures tail.new < (tail.new + split.old)/2. The statement tail.new < (tail.new + split.old)/2, for line 33 ensures t != s, i.e. tail.new ≠ split.old, and linearisation point 5 cannot increase tail beyond split. Therefore, tail.new < (tail.new+split.old)/2.

Table 1a states linearisation points 12, 45, and 47 modify head. Of these linearisation points 45 and 47 decrease head and may violate the Lemma, whereas linearisation point 13 increases head posing no danger of violation. Figure 7 states linearisation point 45 is reached, only if o_allstolen is set, since o_allstolen cannot be altered by any other than the owner thread according to Table 1f. Combined with Lemma 1, this linearisation point cannot be reached unless allstolen is set, thus this Lemma is not applicable to the linearisation point. Conversely, linearisation point 47 can still be reached. In this case, shrink_shared must have returned false, indicating that split < head (Lemma 7). This linearisation point decreases head by 1, therefore split < head implies split ≤ head-1 and the lemma holds. This proves that for all linearisation points modifying tail, split, and head the Lemma still holds.

LEMMA 10. Linearisation point 23 does not increase split beyond head.

Proof. Lemma 8 affirms ¬ allstolen holds when grow_shared is invoked. Table 1f shows allstolen is not altered between the invocation of grow_shared and linearisation point 23. From Lemma 9 we now conclude split ≤ head. Furthermore, Lemma 2 states o_split = split at each point where o_split is used. This implies the modification split.new = (o_split.old+head+1)/2 = (split.old+head+1)/2. Where split.old ≤ head implying split.new ≤ head for the equation is a floor function and the +1 cannot increase the numerator to 2*head+2 since split.new ≤ head. Therefore linearisation point 23 does not increase split beyond head.

LEMMA 11. Invoking pop decreases head by exactly 1.

Proof. Table 1a shows head = head-1 can only be invoked from within the method pop, hence head.new = head.old-1 ⇒ pop() holds. Figure 7 shows it is impossible to go from linearisation point 45 to 47 or vice versa. Furthermore, the Figure shows both linearisation points are the first possible end states to reach. However, the Figure does state that neither linearisation points are reached when head == 0. Though this is not applicable to this situation since we assume the number of pop calls never exceed the number of push calls. Therefore pop() ⇒ head.new = head.old - 1 holds.

LEMMA 12. shrink_shared returns True ⇒ allstolen and o_allstolen.

Proof. Table 1f and 1g indicate both allstolen variables can only be modified within the methods push and shrink_shared. Since both can only be called by the owner thread, they cannot be run simultaneously. Thus, only shrink_shared needs to be inspected. The return true statement of shrink_shared corresponds to linearisation point 40 in Figure 7, which is preceded by linearisation point 39. Since no more linearisation points occur between that point and the return true statement, the allstolen variables are true. Note that stealing threads might not receive this modification of allstolen before they read the value since this paper assumes a system with the x86 memory model.

LEMMA 13. If head decreases and pop returns STOLEN this implies allstolen is set.

Proof. Figure 7 illustrates pop = STOLEN in combination with the decrement of head as linearisation point 45, which can only be reached if o_allstolen or (o_split == head and shrink_shared = true). The latter implies o_allstolen is true accordingly to Lemma 12. Either proposition of the if-statement requires o_allstolen to be set and therefore allstolen to be set (Lemma 1).

LEMMA 14. Linearisation point 23 does not decrease split.

Proof. Table 1b states grow_shared invokes linearisation point 23. In addition, Lemma 8 implies ¬allstolen when grow_shared is invoked. Table 7 indicates allstolen is not altered within this method, therefore we can assume ¬allstolen at linearisation point 23. Lemma 9 suggests that ¬allstolen ⇒ tail ≤ split ≤ head. Therefore we can assume head ≥ split. Linearisation point 23 sets split to (split+head+1)/2, from Lemma 9 we know head ≥ split implying split.new ≥ split.old, proving this Lemma.

LEMMA 15. If allstolen is true, tail cannot increase.
Proof. Table 1d affirms tail only increases at linearisation point 5 of Lace and Lemma 3 states that allstolen must be false to increase tail. Table 1f shows that allstolen can only be set to true at line 39 of Lace. However, the x86 memory model might not have globalised the modification of allstolen, or has globalised the variable after the stealing method checked it. Subsequently, the stealing thread reads tail and split into t and s respectively, and checks whether t < s, a requirement which is also stated in Lemma 3. Figure 7 exposes allstolen can be modified if t == s, either at line 28 or line 33. This means tail = split since they are read from the memory. Stealing threads read the same variables tail and split into their t and s respectively. By Lemma 3, tail cannot increase, endorsing this Lemma. These tail and split variables are globally visible, for linearisation point 23 increases split (Lemma 14) and a decrease in split is made globally visible according to Lemma 6. The update of split is globally visible before the update of allstolen therefore stealing threads which have missed the update of allstolen might still be stopped at line 5 where they check whether t < s. Now two scenarios can occur, either the update of split is made globally visible between line 4 and linearisation point 5, or it is made globally visible after linearisation point 5 has occured. In the former case, cas ensures the operation of increasing tail fails, since split ≠ s, for split is updated whereas the local variable s is equal to the one read at line 3. In the latter case, tail has either grown beyond split, in which case linearisation point 36 ensures split is restored to a valid value as stated in Lemma 5, or tail ≤ new value of split. Figure 7 affirms that in this case allstolen will not be set and this Lemma is not applicable. □

Invariant 2. tail.new ≥ tail.old or
(tail.new < tail.old and tail.old-tail.new ≤ #pop returns STOLEN)

Proof. The first part of the lemma states that tail should increase, this is true for all linearisation points except at line 15 of Lace, as described in Table 1d. The second part states, that if tail is decreased, it decreases by the same amount as the number of calls to pop returning STOLEN. Upon scrutinisation of linearisation point 15, we find tail is set to head-1. We find tail = head.old, where head.old is the value of head before push is called. Since head is only modified by the owner thread (Table 1a) and due to the linear nature of push, linearisation point 13 came before linearisation point 15 (Figure 6). Thus the decrease of head when tail is set to head-1 cancels out the increase of head at linearisation point 13.

Now, we must establish that once pop returns STOLEN, it is not possible for tail to increase. Note that this is not necessarily true for the invariant to hold, but it is convenient for the lemma to be proven. Lemma 13 states that allstolen is true in the described scenario, i.e. head is decreased and pop returns STOLEN. Along with Lemma 15, we find it impossible for tail to increase once pop returned stolen. Because Lemma 11 states pop decreases head with exactly 1, the number of pop() = STOLEN corresponds to the number of decreases of head. According to Table 1f and Figure 6 allstolen cannot be reset unless preceded by linearisation point 15. Therefore, x consecutive calls to pop returning STOLEN correspond to x decreases of head by 1. Thus, when linearisation point 15 is executed, tail.old-tail.new = head.old-head.new, where head.old is the value of head before the first pop() = STOLEN and head.new is the value of head after the last pop() = STOLEN. This proves Lemma 2 correct. □

Invariant 3. (pop returns STOLEN and head.new < head.old) || (pop returns WORK and head.new < head.old)

Proof. Lemma 11 expounds invoking pop always decreases head. Furthermore, we assumed the number of calls to pop never exceed the number of calls to push, meaning pop never returns EMPTY. Upon inspection of pop we find this method either returns STOLEN or WORK, thereby proving the invariant. □

5.1 Correctness

As the required properties stated two invariants needed to be proven to establish correctness of Lace. Invariant tail.new ≥ tail.old or (tail.new < tail.old and tail.old-tail.new ≤ #pop == STOLEN) was proven by Invariant 2, using Lemmas 13, 15, and 11, as well as Table 1 and flow diagrams 6 and 7. This proves each stolen task is only stolen once. The invariant (pop returns STOLEN and head.new < head.old)||(pop returns WORK and head.new < head.old) was proven by Invariant 3 using Lemma 11 and the assumed property that the number of calls to pop never exceed the number of calls to push. From this invariant we conclude each task is executed exactly once, since the previous Invariant states each task is stolen only once. Thus, the synchronisation method from the assumed algorithm invoking the deque methods sees the task was stolen, waits until it finished and uses the result without computing it again. Conversely, when the synchronisation method finds a task is not yet executed, it executes the task itself and then uses the result. Hereby we have proven each task is executed exactly once as stated as a required property.

6. CONCLUSION

This study focused on proving the algorithm Lace correct using linearisation points and their control flow. Hereby we constructed an informal proof establishing correctness in both required invariants. I.e. we found tail either increases or stays equal or it is decreased by the same amount of pops that return STOLEN, proving that each task is only stolen once. Furthermore we have proven each task is executed exactly once by proving Invariant 5. By these proofs we have shown it is possible to prove the required properties of concurrent algorithms using linearisation points and their control flow. Concurrent operations such as compare-and-swap and memory fences have properties that are sufficient to prove the correctness of a concurrent program. This paper gains further insights in the correlation between these methods and their effect on lockless concurrent algorithms such as Lace. It shows the correctness of the algorithm depends on the atomic properties of cas and the assurance of memory fences that each buffered write is made globally visible. However the constructed proof is an informal proof, i.e. it is not generated or checked by any theorem prover tool. Hence, the presented proof is intended as a basis for creating a formal proof that ensures Lace's correctness.

6.1 Future work

This paper suggests to use theorem prover tools such as VerCors[1], PVS[4] or Isabelle[5] to generate a formal proof for Lace. A formal proof can be generated on the basis of this paper using these theorem prover tools. Such proof creates certainty in the correctness of Lace allowing the algorithm to be used more widespread. In addition
to creating a formal proof, other variations of Lace can be scrutinised, e.g. variations where stealing threads can steal multiple tasks instead of one. These might increase performance of Lace, but need to be proven correct separately since this paper limits itself to thieving threads stealing one task at the time.

7. REFERENCES
APPENDIX

A. LINEARISATION POINTS
The linearisation points used to generate the informal proof are stated in Table 1 below.

Table 1: The method column indicates the method where the variable is modified. Lin pt column refers to the linearisation point in the Lace algorithm as shown in Figure 8. The op column explains the executed operation. The final column init point refers to the point in the Lace algorithm where the variables are read. Note that in 1b and 1c o_split and head are abbreviated to o, h respectively. Tail and split are abbreviated to t and s since these is a local variables as in the algorithm.

(a) Linearisation points of head variable.

<table>
<thead>
<tr>
<th>method</th>
<th>lin pt</th>
<th>new value</th>
<th>init point</th>
</tr>
</thead>
<tbody>
<tr>
<td>push</td>
<td>13</td>
<td>head+1</td>
<td>13</td>
</tr>
<tr>
<td>pop</td>
<td>45</td>
<td>head-1</td>
<td>45</td>
</tr>
<tr>
<td>pop</td>
<td>47</td>
<td>head-1</td>
<td>47</td>
</tr>
</tbody>
</table>

(b) Linearisation points of split variable.

<table>
<thead>
<tr>
<th>method</th>
<th>lin pt</th>
<th>new value</th>
<th>init point</th>
</tr>
</thead>
<tbody>
<tr>
<td>push</td>
<td>15</td>
<td>head</td>
<td>15</td>
</tr>
<tr>
<td>gr_shared</td>
<td>23</td>
<td>new_s=(o+h+1)/2</td>
<td>22</td>
</tr>
<tr>
<td>shr_shared</td>
<td>30</td>
<td>new_s=(t+s)/2</td>
<td>27</td>
</tr>
<tr>
<td>shr_shared</td>
<td>36</td>
<td>new_s=(t+s)/2</td>
<td>t:32,s:27</td>
</tr>
</tbody>
</table>

(c) Linearisation points of o_split variable.

<table>
<thead>
<tr>
<th>method</th>
<th>lin point</th>
<th>new value</th>
<th>init point</th>
</tr>
</thead>
<tbody>
<tr>
<td>push</td>
<td>18</td>
<td>head</td>
<td>18</td>
</tr>
<tr>
<td>gr_shared</td>
<td>24</td>
<td>new_s=(o+h+1)/2</td>
<td>22</td>
</tr>
<tr>
<td>shr_shared</td>
<td>37</td>
<td>new_s=(t+s)/2</td>
<td>27</td>
</tr>
<tr>
<td>shr_shared</td>
<td>37</td>
<td>new_s=(t+s)/2</td>
<td>t:32,s:27</td>
</tr>
</tbody>
</table>

(d) Linearisation points of tail variable

<table>
<thead>
<tr>
<th>method</th>
<th>lin point</th>
<th>new value</th>
<th>init point</th>
</tr>
</thead>
<tbody>
<tr>
<td>steal</td>
<td>5</td>
<td>t+1</td>
<td>3</td>
</tr>
<tr>
<td>push</td>
<td>15</td>
<td>head-1</td>
<td>15</td>
</tr>
</tbody>
</table>

(e) Linearisation points of movesplit variable

<table>
<thead>
<tr>
<th>method</th>
<th>lin point</th>
<th>new value</th>
<th>init point</th>
</tr>
</thead>
<tbody>
<tr>
<td>steal</td>
<td>8</td>
<td>true</td>
<td>8</td>
</tr>
<tr>
<td>push</td>
<td>16</td>
<td>false</td>
<td>16</td>
</tr>
<tr>
<td>grow_shared</td>
<td>25</td>
<td>false</td>
<td>25</td>
</tr>
</tbody>
</table>

(f) Linearisation points of allstolen variable

<table>
<thead>
<tr>
<th>method</th>
<th>lin point</th>
<th>new value</th>
<th>init point</th>
</tr>
</thead>
<tbody>
<tr>
<td>push</td>
<td>17</td>
<td>false</td>
<td>17</td>
</tr>
<tr>
<td>shrink_shared</td>
<td>39</td>
<td>false</td>
<td>39</td>
</tr>
</tbody>
</table>

(g) Linearisation points of o_allstolen variable

<table>
<thead>
<tr>
<th>method</th>
<th>lin point</th>
<th>new value</th>
<th>init point</th>
</tr>
</thead>
<tbody>
<tr>
<td>push</td>
<td>19</td>
<td>false</td>
<td>19</td>
</tr>
<tr>
<td>shrink_shared</td>
<td>40</td>
<td>true</td>
<td>40</td>
</tr>
</tbody>
</table>

B. LACE ALGORITHM
This paper uses the Lace algorithm as given in Figure 8. All assumptions, linearisation points, flow diagrams and lemma’s are based on this algorithm as described in "Lace: Non-blocking Split Deque for Work-Stealing" [10].

```python
1 def steal():
2     if allstolen: return NOWORK
3     (t, s) = (tail, split)  #o, s are local
4     if t < s:
5         if cas((tail, split), (t, s), (t+1, s)):
6             return WORK(t)
7         else: return NONE  #busy
8     elif movesplit: movesplit = 1
9         return NONE  #no work

10 def push(data):
11     if head == size: return FULL
12     head = head + 1  #write task data at deque head
13     if o_allstolen:
14         (tail, split) = (head-1, head)
15         if movesplit: movesplit = 0
16         allstolen = 0
17         o_split = head
18         o_allstolen = 0
19     elif movesplit: grow_shared()

21 def grow_shared():
22     new_s = (o_split+head+1)/2
23     split = new_s
24     movesplit = 0

26 def shrink_shared():
27     (t, s) = (tail, split)
28     if t != s:
29         new_s = (t+s)/2
30         split = new_s
31         MFENCE
32         t = tail  #read again
33     if t != s:
34         if t > new_s:
35             new_s = (t+s)/2
36             split = new_s
37             o_split = new_s
38             return FALSE
39             allstolen = 1
40             o_allstolen = 1
41             return TRUE

42 def pop():
43     if head == 0: return EMPTY, None
44     if o_allstolen or (o_split == head and
45         shrink_shared()):
46         head = head-1
47         return STOLEN, head
48     if movesplit: grow_shared()
49         return WORK, head
```

Figure 8: Lace algorithm as described in "Lace: Non-blocking Split Deque for Work-Stealing"[10]