

Proving correctness of Lace using interactive theorem prover tools

Thijs van Ede
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
t.s.vanede@student.utwente.nl

ABSTRACT

Interactive theorem prover tools

Keywords

interactive theorem prover tool, PVS, Lace, linearisation points, concurrent programming, proof

1. INTRODUCTION

Modern day processors' multi-core architecture has increased 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[9] and first-order logic[12].

Interactive theorem prover tools such as Isabelle[8], Coq[1], and PVS[7] use various methods of logical based proving check models. These tools are helpful in proving properties of both sequential[3] and concurrent[5] [10] algorithms. The theorem prover tool PVS uses several decision procedures and a symbolic model checker, in combination with a random tester to help generating formal proofs.

This paper tries to prove several properties of the Lace algorithm[13] 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 proof correct.

In this paper, we will prove the Lace algorithm correct using the interactive theorem prover tool PVS. The algorithm is modeled by constructing linearisation points where the state of the system changes, and using them to generate a flow diagram of the program. Subsequently, PVS assists in proving the required invariants using this model in combination with several assumptions about the system. With the invariants a comprehensive proof of the Lace algorithm is constructed.

1.1 Lace

Lace is a work stealing algorithm[4] which uses a compare-and-swap operation to steal tasks[13]. Each process 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 processes operate on the tail. Besides the deque, a process 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 thief process 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 are allowed to steal this task. Conversely, when the tail increases beyond the split point thief processes may not steal the task. Figure 1 illustrate the workings of this deque with its head, tail, and split variables. The deque is presented as an ar-

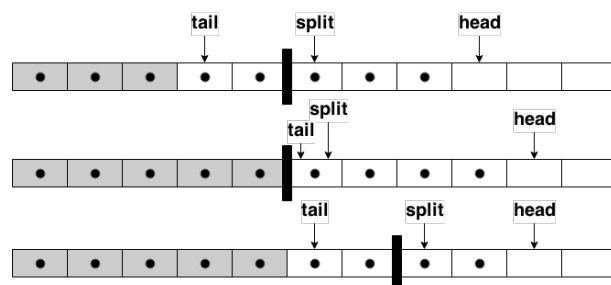


Figure 1: Example on manipulation of deque variables.

ray of slots which may be empty or contain tasks. Slots 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, $tail = split$ which means thief processes may not steal tasks. 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.

1.2 PVS

Concurrent programs such as Lace, can be proven in a variety of different ways. There exist tools for proving concurrent programs such as VerCors[2], but they give rather little feedback on the workings of the program. That is, these tools state whether a program works correct, it does

not give the reason it works correct. Since Lace is still being developed, it is desirable to develop understanding of the algorithm. Interactive theorem prover tools such as PVS are ideally suited for this purpose, since they give feedback on required properties of lemmas and require the user to check whether required properties hold. PVS in particular comes highly recommended for its abilities to proof concurrent programs[5] [10]. Therefore, this paper uses PVS as interactive theorem prover tool to prove Lace correct.

2. PRELIMINARIES

2.1 System

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, that is, 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[6]. These linearisation points are the points in time the state of the system changes. 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 $\iff x < \text{head}$

DEFINITION 2. A task x is stolen $\iff x < \text{tail}$

DEFINITION 3. A task x is shared $\iff x < \text{split}$

DEFINITION 4. A task x is private $\iff x \geq \text{split}$

In proving Lace correct, this paper assumes there is only one owner thread per deque and a possibly infinite number of thieving threads. In practice there are more than one

owner threads since there are multiple dequeues. However, the proof of correctness for one deque is equivalent to all other dequeues, i.e. we assume that owner threads are not influenced by each other, only by thieving threads. The second assumption is that thieving threads only call **steal** and the owner threads only call **push** and **pop**. This implies that the owner thread is also the only thread that has access to **grow_shared** and **shrink_shared**, since they are only called in **push** and **pop**. In addition, this paper assumes the number of calls to **pop** is equal to the number of calls to **push** when the algorithm finished executing. I.e. the deque is empty when the program finished. Finally, all deque values - i.e. **tail**, **split**, **head**, and the entries within the deque - are all set to zero when the deque is initialised.

1. One owner thread per deque.
2. Possibly infinite number of thieving threads per deque.
3. **steal** is only called by thieving threads.
4. **push** and **pop** are only called by owner thread.
5. Owner thread only calls **grow_shared** and **shrink_shared**.
6. Number of **pop** calls \leq number of **push** calls.
7. When finished, number of **pop** calls = number of **push** calls.
8. On initialisation all deque variables are set to zero.

Figure 2: Assumed properties of Lace

2.5 Required properties

Lace requires some properties to work correctly. First of all, the variables **head**, **split**, and **tail** should stay within bounds. Figure 3 expresses this requirement as invariants. Furthermore, a task must be stolen only once, and ex-

```
0 ≤ head ≤ size
0 ≤ split ≤ head
0 ≤ tail ≤ head
```

Figure 3: Invariants: variables stay within bounds.

ecuted exactly once. To prove these properties, they are written as invariants. When every task is stolen only once, it means thieving threads can only steal tasks $\geq \text{tail}$ and that when **tail** reduces, a task cannot be stolen again. The former part of the property is trivial, if **tail** increases it steals an unstolen task which is consistent with the requirement. The latter part is less trivial, if **tail** reduces, a stolen task cannot be executed again, therefore the stolen tasks need to be removed before **tail** is reduced. This is written as an invariant in Figure 4. To elaborate on this

```
tail.new > tail.old || (tail.new ≤ tail.old &&
tail.old == head && pop() == STOLEN, head)
```

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

invariant: either **tail** increases when a new task is stolen, or **tail** decreases but the task has to be removed from the deque so that it cannot be stolen again. When a task is executed once, this implies it is only stolen once and not executed by the owner thread, or it is executed by the owner thread and never stolen. This can be written as the invariant in Figure 5.

3. METHOD

```
(pop() == STOLEN && head.new < head.old) ||
(pop() == WORK && head.new < head.old)
```

Figure 5: Invariant: every task is executed only once.

In proving concurrent programs such as Lace using interactive theorem prover tools, linearisation points of the algorithm must be appointed. Each linearisation point modifies a variable of the program. For concurrent programs, each possible state of the system can be derived by altering the order in which the linearisation points occur. From which conclusions can be drawn about the desired behaviour of the system with respect to the possible behaviour of the system.

The first step of appointing linearisation points is trivial and could be done automatically. However, because the algorithm is relatively simple and to illustrate how these points can be identified it is done manually. First all global variables of the algorithm are identified, then each point in the algorithm where a variable is changed is appointed as a linearisation point with the operation that is performed at that point. This is done for all global variables, i.e. `head`, `split`, `tail`, `allstolen`, `movesplit` as well as private variables `o_allstolen` and `o_split`.

After the linearisation points are established, this paper deduces the order in which they occur. Therefore, we setup flow diagrams of the methods `pop` and `push` to establish the possible orders of linearisation points. These diagrams include methods invoked within the `pop` or `push` method. No flow diagram is setup for the method `steal`, since we assume numerous threads execute this method simultaneously which renders a diagram useless.

Finally, we prove the required properties of Lace using several lemmas which base themselves on the linearisation points of the system and the possible order of these points. For these proofs, the interactive theorem prover tool PVS[7] is used which correctly states whether lemmas hold. These lemmas, as well as axioms are introduced by the user into PVS. Within this paper, all lemmas and axioms are deduced from the assumptions about Lace, constructed linearisation points, and their order as described by the flow diagrams. In this paper, PVS’ grind option[11] is used extensively to proof lemmas within the theorem prover tool. This option attempts to solve given lemmas automatically using lemmas introduced by the user. Ultimately, the invariant `(pop() == STOLEN && head.new < head.old) || (pop() == WORK && head.new < head.old)` is proven which establishes correctness of Lace.

Various interactive theorem prover tools may be used for proving this algorithm correct. However, this paper uses the PVS 6.0 Allegro Lisp Binary for a Linux Intel 64-bit machine. The model of the algorithm is constructed from its description in the paper [13].

A proof constructed with an interactive theorem prover such as PVS holds in all cases and could also be constructed without using PVS. The tool only helps to construct the proof and formalise the method of proving algorithms correct. Therefore, PVS speeds up the process of constructing a proof and increasing confidence in that proof. It also increases the transferability of this research since the tool constructs equal proofs in an identical way.

4. MODELING LACE

A model for the Lace algorithm is constructed from its specifications as described in [13] and displayed in Figure

17. Note that Figure 17 projects a slightly different form of the algorithm. However, this alternate form is used to construct a model for Lace. First of all, a model is created from the algorithm without the `allstolen` flag.

4.1 Linearisation Points

In Lace, the state of the system depends on variables `head`, `split`, and `tail`. The linearisation points are defined on these variables and can be found in Table 1a, 1b, and 1d respectively. 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 `movesplit` and `allstolen` 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. Therefore, these variables are also included in the linearisation points, which can be found in Table 1e, 1f, and 1g.

4.1.1 Head

Variable `head` is modified in both `push` and `pop` methods. Table 1a gives an overview of the linearisation points and the places where the variables used in the linearisation points are initialised. The first linearisation point is at line 13 of Lace as shown in Figure 17, where `head` is incremented by 1. In this case the variable `head` is read in the same line of code but does not occur simultaneously with the write operation. The second and third linearisation points are at line 45 and 47 of Lace as shown in Figure 17 respectively. In both cases, `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 `split` has two instances, namely `split` and `o_split`. The latter instance of `split` is private, and thus visible to the owner thread but not to thieving threads. To avoid confusion, `split` and `o_split` are displayed in Table 1b and Table 1c 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. 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.

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 39. The private `o_allstolen` is set to false and true at lines 19 and 40 respectively. The linearisation points can be found in Table 1f and 1g.

method	lin pt	new value	init point
push	13	head+1	13
pop	45	head-1	45
pop	47	head-1	47

(a) Linearisation points of head variable.

method	lin pt	new value	init point
push	15	head	15
gr_shared	23	$\text{new_s} = (\text{o} + \text{h} + 1) / 2$	22
shr_shared	30	$\text{new_s} = (\text{t} + \text{s}) / 2$	27
shr_shared	36	$\text{new_s} = (\text{t} + \text{s}) / 2$	t:32,s:27

(b) Linearisation points of split variable.

method	lin pt	new value	init point
push	18	head	18
gr_shared	24	$\text{new_s} = (\text{o} + \text{h} + 1) / 2$	22
shr_shared	37	$\text{new_s} = (\text{t} + \text{s}) / 2$	27
shr_shared	37	$\text{new_s} = (\text{t} + \text{s}) / 2$	t:32,s:27

(c) Linearisation points of o_split variable.

method	lin point	new value	init point
steal	5	t+1	3
push	15	head-1	15

(d) Linearisation points of tail variable

method	lin point	new value	init point
steal	8	true	8
push	16	false	16
grow_shared	25	false	25

(e) Linearisation points of movesplit variable

method	lin point	new value	init point
push	17	false	17
shrink_shared	39	true	39

(f) Linearisation points of allstolen variable

method	lin point	new value	init point
push	19	false	19
shrink_shared	40	true	40

(g) Linearisation points of o_allstolen variable

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 17. 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.

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.

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.

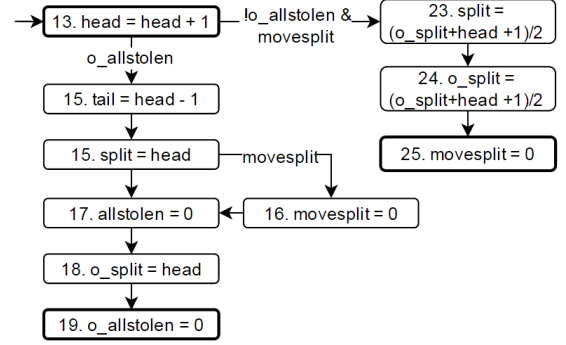


Figure 6: Flow diagram of method push.

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 lines. 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`.

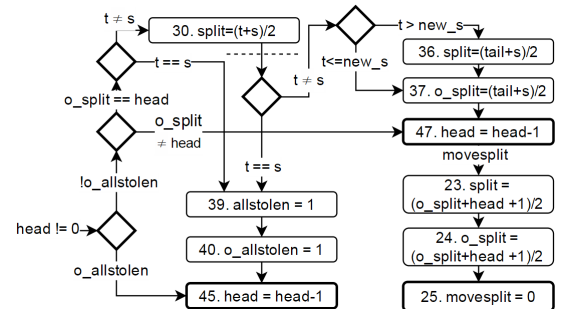


Figure 7: Flow diagram of method pop.

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. *allstolen = o_allstolen at load operations.*

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 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. \square

LEMMA 2. *split = o_split at load operations.*

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 `movesplit` and `allstolen` are modified between the linearisation points of the `split` variables. Furthermore, Figure 7 states that between linearisation points 23 and 24 no reads of either `split` variable occur. Between linearisation point 30 of `split` and 37 of `o_split` there are no read operations for either `split` variable. In addition, between linearisation point 36 of `split` and linearisation point 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. \square

LEMMA 3. $0 \leq \text{head} \leq \text{size}$.

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. Since we assume `head = 0` at the start of the algorithm, i.e. $0 \leq \text{head} < \text{size}$, it is left to prove linearisation point 13 cannot increase `head` beyond `size`. Subsequently, we prove linearisation points 45 and 47 cannot decrease `head` to a value lower than 0. In PVS we input the code as depicted in Figure 8. To prove lemma `lemma_size`, we use axioms `head_start`, stating `head < size` at the initialisation, and `head_size` stating `head \neq size`, i.e. line 11 of Lace. Then, we expand function `incr_head(head)`, which increases `head` by 1, i.e. line 13 of Lace. Then using PVS' grind option, it concludes that indeed the lemma holds. Proving `head` cannot decrease to a value lower than 0, we use axioms `head_start` and `head_zero` and prove `lemma_zero` using PVS' grind option. This concludes that the lemma holds. \square

LEMMA 4. *tail.new = tail.old+1 \Rightarrow !allstolen and $t < s$ and $t == \text{tail}$ and $s == \text{split}$*

```
head_start:  AXIOM head >= 0 AND head < size
head_zero:   AXIOM NOT (head = 0)
head_size:   AXIOM NOT (head = size)

incr_head(head):  posnat = head+1
decr_head(head):  posnat = head-1

lemma_size:  LEMMA incr_head(head) <= size
lemma_zero:  LEMMA decr_head(head) >= 0
```

Figure 8: PVS proof of $0 \leq \text{head} \leq \text{size}$

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. Thus this can be modeled in PVS as described in Figure 9. \square

```
cas(var1, var2, exp1, exp2, val1, val2: posnat)
: bool = IF (var1 = exp1 AND var2 = exp2 THEN
true ELSE false ENDIF
```

```
steal(t, s, tail, split, allstolen): bool =
IF allstolen THEN false ELSE (IF t < s THEN
cas(t,s,tail,split,(t+1),s) ELSE false ENDIF)
ENDIF
```

```
lem_incr_tail: LEMMA (NOT allstolen AND
t < s AND t = tail AND s = split) IMPLIES
steal(t,s,tail,split,allstolen) = true
lem_incr_tail: LEMMA NOT (NOT allstolen AND
t < s AND t = tail AND s = split) IMPLIES
steal(t,s,tail,split,allstolen) = false
```

Figure 9: PVS proof of incrementing tail

LEMMA 5. *Linearisation point 30 \Rightarrow split.new \leq split.old*

PROOF. Lemma `lem_decr_a` in Figure 10 corresponds to line 30 of Lace and states that `decr_tail \leq split`, under the assumptions that `tail \neq split` and `tail < split`. The former assumption is valid since Figure 7 states the variables `tail` and `split` are not equal when linearisation point 30 occurs and the read variables `tail` and `split` are not modified in between. The latter assumption is valid since Lemma 4 states `tail` cannot increase if `tail \geq split` and at the start of Lace `tail == split == 0`. Using PVS' grind option, this Lemma is proven correct. \square

LEMMA 6. *Linearisation point 36 \Rightarrow split.new \leq split.old where split.old is before Linearisation point 30.*

PROOF. Lemma `lem_decr_b` in Figure 10 corresponds to the linearisation point of line 36 of Lace and states that after the linearisation point of line 30, line 36 can increase `split`, but not beyond the point of `split.old`, i.e. `split` before the linearisation point of line 30. It assumes `tail \neq split` and `t > decr_tail(tail,split)` and `t < split`. The first of these assumptions is true according to Figure 7, it shows `t \neq s` is a requirement for reaching linearisation point 36. Since `tail` is reread at line 32, but `split` is equal to the value read at line 27 this assumption still holds. The second assumption is valid since Figure 7 expounds `t > new_s` for linearisation point 36 to be reached. That is, the newly read value of `tail` here

modeled as t is larger than the computed value of `split` at linearisation point 30. Note that the newly read value of `tail` cannot increase any further, since the memory fence of line 31 asserts the modification of `split` is globally visible. Any reordering of read and write operations as the x86 memory model allow do not apply in this part of the code since the memory fence forces global visibility. The final assumption of $t < \text{split}$ is true since Lemma 4 affirms `tail` cannot increase unless it is smaller than `split`. With these assumptions, PVS shows that indeed the newly computed value of `split.new` \leq `split.old`. \square

LEMMA 7. *min(linearisation point 30, linearisation point 36) is globally visible.*

PROOF. Lemma `lem_decr_c` in Figure 10 insists linearisation point 36 cannot decrease with respect to linearisation point 30. Here the assumptions of `lem_decr_a` and `lem_decr_b` are both needed to prove the lemma correct. PVS proves that the minimum value of `split` after linearisation point 30 and linearisation point 36 is linearisation point 30. This change is visible because of the memory fence of line 31. No memory fence is needed for linearisation point 36, since stealing threads cannot abuse this modification because `split.old` \leq `tail.new`. Therefore stealing threads cannot steal more because of the modification. \square

```
decr_tail(tail, split): posnat = (tail+split)/2
```

```
lem_decr_a: LEMMA NOT tail = split AND tail <
split IMPLIES decr_tail(tail,split) <= split
```

```
lem_decr_b: LEMMA NOT tail = split AND t >
decr_tail(tail,split) AND t < split IMPLIES
decr_tail(t, split) <= split
```

```
lem_decr_c: LEMMA NOT tail = split AND tail
< split AND t > decr_tail(tail, split) AND
t < split IMPLIES decr_tail(tail, split) <=
decr_tail(t, split)
```

Figure 10: PVS proof of Lemmas 5, 6, and 7. Method `decr_tail` returns the modified value of `tail`. `lem_decr_a` is used in Lemma 5, `lem_decr_b` is used in Lemma 6, and `lem_decr_c` is used in Lemma 7. Variables `tail` and `split` correspond to the variables read at line 27 of `Lace`, variable `t` corresponds to the newly read variable `tail` at line 32 of `Lace`.

LEMMA 8. *shrink_shared returns false and $\text{tail} \leq \text{head} \Rightarrow \text{tail} \leq \text{split} < \text{head}$.*

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 $t \neq s$, i.e. `tail` \neq `head`, since `split` = `head` and there occur no modifications to either variables in the meantime. Lemmas 5 and 6 state `tail` can only decrease at these points. Combined with the assumption that `split` \neq `head` it follows that `tail` \leq `split` $<$ `head`. \square

LEMMA 9. *grow_shared \Rightarrow $\neg \text{allstolen}$*

PROOF. Figure 7 specifies that `grow_shared`, represented by linearisation point 23, can only be reached if $\neg \text{o_allstolen}$. 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 an invocation of `grow_shared` \Rightarrow $\neg \text{allstolen}$. \square

LEMMA 10. *$\neg \text{allstolen} \Rightarrow \text{tail} \leq \text{split} \leq \text{head}$*

PROOF. This paper assumes that the deque is initialised with `tail` = `split` = `head` = 0 and $\neg \text{allstolen}$, hence, the lemma holds 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 4.

Table 1d demonstrates variable `tail` is modified at linearisation point 5 and 15. Lemma 4 states that for linearisation point 5 to execute $t < s$ and $t == \text{tail}$ and $s = \text{split}$. When modeling this in PVS as in Figure 11, we see that `tail` \leq `split` holds. Figure 6 expounds linearisation point 15 cannot be reached if $\neg \text{o_allstolen}$, because Lemma 1 states `o_allstolen` = `allstolen` at load operations, this linearisation point cannot be reached when $\neg \text{allstolen}$.

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 $\neg \text{allstolen}$. Linearisation point 23 can be reached, Lemma 11 states that after this linearisation point `split` \leq `head` still holds, using the assumption of this Lemma, that $\neg \text{allstolen} \Rightarrow \text{split} \leq \text{head}$. Linearisation point 30 may violate the lemma, since `tail` can grow to the value of `split.old` and Lemma 5 expounds `split.new` \leq `split.old` after the modification of linearisation point 30. However, as described in Figure 11 the linearisation point 36 will restore the value `split` so that `tail` \leq `split` holds.

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. Contrary, linearisation point 47 can still be reached. In this case, `shrink_shared` must have returned false, indicating that `split` $<$ `head` (Lemma 8). With this assumption we prove in PVS that `split` \leq `head-1` still holds. This is done in Figure 11, at `lem_head`, and proven using PVS' grind option.

This proves that for all linearisation points modifying `tail`, `split`, and `head` the Lemma still holds. \square

LEMMA 11. *Linearisation point 23 \Rightarrow `split.new` \leq `head`*

PROOF. Lemma 9 affirms $\neg \text{allstolen}$ holds when `grow_shared` is invoked. Table 1f shows `allstolen` is altered between the invocation of `grow_shared` and linearisation point 23. In combination with the assumption of Lemma 10 that

```

incr_tail(tail): posnat = tail+1
lem_tail: LEMMA tail < split IMPLIES
incr_tail(tail) <= split

decr_tail(tail,split): posnat = (tail+split)/2
lem_split: LEMMA NOT tail = split AND t >
decr_tail(tail, split) AND t < split IMPLIES
decr_tail(t, split) > t

decr_head(head): posnat = head-1
lem_head: LEMMA split < head IMPLIES split <=
decr_head(head)

```

Figure 11: PVS proof of Lemma 10

$\text{tail} \leq \text{head}$ we can model the linearisation point in PVS and proof that this Lemma holds. Figure 12 contains the PVS code used to proof this Lemma. Note how the function `grow_shared` adds 2 to `split+head` instead of 1 and how it subtracts 1 after the division. This is to cope with rounding the resulting integer down as Lace does. Using PVS' grind option lemma `lem_incr_split` is proven and thus Lemma 11 holds. \square

```

grow_shared(split, head): posnat =
((split+head+2)/2)-1
lem_incr_split: LEMMA split <= head IMPLIES
grow_shared(split, head) <= head

```

Figure 12: PVS proof of Lemma 11.

LEMMA 12. $\text{pop}() \iff \text{head.new} = \text{head.old} - 1$

PROOF. Table 1a shows `head = head-1` can only be invoked from within the method `pop`, hence `head.new = head.old-1 \Rightarrow 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() \Rightarrow head.new = head.old -1` holds. \square

LEMMA 13. *shrink_shared* returns *True* \Rightarrow *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, since no more linearisation points occur between that point and the return `true` statement. Therefore, the code depicted in Figure 13 is used to proof the `allstolen` variables are set when `shrink_shared` returns true. By adding lemmas `allstolen_set` and `o_allstolen_set` to the proof, and using PVS' grind option, the tool verifies both `allstolen` flags are set to true if `shrink_shared` returns 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. \square

LEMMA 14. $\text{head.new} = \text{head.old} - 1$ and $\text{pop}() = \text{STOLEN} \Rightarrow \text{allstolen}$

```

allstolen_set: AXIOM allstolen
o_allstolen_set: AXIOM o_allstolen
shr_shared_ret: LEMMA allstolen AND o_allstolen

```

Figure 13: PVS proof of $\text{shrink_shared} \Rightarrow \text{allstolen} \ \&\& \ \text{o_allstolen}$

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`, `o_allstolen`). Therefore `pop = STOLEN \Rightarrow o_allstolen`. This can be coded into PVS as illustrated in Figure 14. Here we assume `head` is decremented and `STOLEN` is returned if the if-statement of Lace rule 44 is true. Using PVS' grind option we find this lemma holds. From Lemma 13 we find `o_allstolen` or (`o_split == head` and `allstolen`). In combination with Lemma 1 stating that the `allstolen` variables are equal when at their load operations we conclude that `pop = STOLEN \Rightarrow allstolen`. \square

```

shrink_shared(o_allstolen, allstolen): bool =
o_allstolen AND allstolen

```

```

pop(head, o_split, o_allstolen, allstolen):
bool = NOT (head = 0) AND (o_allstolen OR
shrink_shared(o_allstolen, allstolen))

```

```

lem_allstolen: LEMMA pop(head, o_split,
o_allstolen, allstolen) IMPLIES o_allstolen

```

Figure 14: PVS proof `pop() = STOLEN` implies `allstolen`

LEMMA 15. *Linearisation point 23* \Rightarrow $\text{split.new} \geq \text{split.old}$

PROOF. Table 1b states `grow_shared` invokes linearisation point 23. In addition, Lemma 9 implies $\neg \text{allstolen}$ when `grow_shared` is invoked. Table 7 indicates `allstolen` is not altered within this method, therefore we can assume $\neg \text{allstolen}$ at linearisation point 23. Lemma 10 suggests that $\neg \text{allstolen} \Rightarrow \text{tail} \leq \text{split} \text{ leq } \text{head}$, therefore we can use this assumption in proving this Lemma. Figure 15 models the desired Lemma in PVS. Here PVS proves `lem_grow` using its grind option, proving that linearisation point 23 \Rightarrow `split.new \geq split.old`. \square

```

grow_shared(split,head):posnat = (split+head+1)/2
lem_grow: LEMMA head >= split IMPLIES
grow_shared(split, head) >= split

```

Figure 15: PVS proof of Lemma 15

LEMMA 16. $\text{allstolen} \Rightarrow \neg \text{tail.new} = \text{tail.old} + 1$

PROOF. Table 1d affirms `tail` can only increase at linearisation point 5 of Lace and Lemma 4 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 4. Figure 7 expounds `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 4, `tail` cannot increase, endorsing this Lemma. These `tail` and `split` variables are globally visible, for linearisation point 23 increases `split` (Lemma 15) and a decrease in `split` is made globally visible according to Lemma 7. 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 occurred. In the former case, `cas` ensures the operation of increasing `tail` fails, since `split` \neq `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 6. Or `tail` \leq new value of `split`. Figure 7 affirms that in this case `allstolen` will not be set and this Lemma is not applicable. \square

LEMMA 17. $tail.new \geq tail.old$ or
 $(tail.new < tail.old \text{ and } tail.old - tail.new \leq \#pop() == STOLEN)$

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). Figure 16 models these linearisation points by describing `tail` as `decr(incr(head))` and `head.old` as `head`. Using PVS' grind option, we conclude that indeed `tail = head.old`.

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 Lemma to hold, but it is convenient for the lemma to be proven. Lemma 14 states that `allstolen` is true in the described scenario, i.e. `head` is decreased and `pop` returns `STOLEN`. Along with Lemma 16, we find it impossible for `tail` to increase once `pop` returned `stolen`.

Because Lemma 12 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 17 correct. \square

6. REFERENCES

- [1] The coq proof assistant.
<https://coq.inria.fr/what-is-coq>. Accessed: 2015-03-29.

```
incr(head): posnat = head+1
decr(head): posnat = head-1
lem_set_tail: LEMMA decr(incr(head)) = head
```

Figure 16: PVS proof Lemma 17

- [2] A. Amighi, S. C. C. Blom, S. Darabi, M. Huisman, W. I. Mostowski, and M. Zaharieva-Stojanovski. Verification of concurrent systems with `vercors`. In M. Bernardo, F. Damiani, R. Hahnle, E. Johnsen, and I. Schaefer, editors, *Formal Methods for Executable Software Models - 14th International School on Formal Methods for the Design of Computer, Communication, and Software Systems*, volume 8483 of *Lecture Notes in Computer Science*, pages 172–216. Springer Verlag, Berlin, June 2014.
- [3] B. Badban, W. Fokkink, J.F. Groote, J. Pang, and J. van de Pol. Verification of a sliding window protocol in `μcr` and `pvs`. *Formal Aspects of Computing*, 17(3):342–388, 2005.
- [4] R.D Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JAMC)*, 46(5):720–748, 1994.
- [5] R. Colvin, L. Groves, V. Luchangco, and M. Moir. Formal verification of a lazy concurrent list-based set algorithm. In *Computer Aided Verification*, pages 475–488. Springer, 2006.
- [6] M.P. Herlihy and J.M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.
- [7] S. Owre, J. M. Rushby, and N. Shankar. Pvs: A prototype verification system. In *Automated Deduction - CADE-11*, pages 748–752. Springer, 1992.
- [8] Lawrence C Paulson. *Isabelle: A generic theorem prover*, volume 828. Springer Science & Business Media, 1994.
- [9] John C Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74. IEEE, 2002.
- [10] N. Shankar. Verification of real-time systems using `pvs`. In *Computer Aided Verification*, pages 280–291. Springer, 1993.
- [11] N. Shankar, S. Owre, J.M. Rushby, and D.W.J. Stringer-Calvert. *PVS Prover Guide*. SRI International, Computer Science Laboratory 333 Ravenswood Avenue Menlo Park CA 94025, 2.4 edition, November 2001. <http://pvs.csl.sri.com/>.
- [12] Raymond M Smullyan. *First-order logic*. Courier Corporation, 1995.
- [13] T. van Dijk and J.C. van de Pol. `Lace`: Non-blocking split deque for work-stealing. *Lecture Notes in Computer Science*, 8806:206–217, 2014.

APPENDIX

A. LACE ALGORITHM

This paper uses the `Lace` algorithm as given in Figure 17. 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"[13].


```

1 def steal():
2     if allstolen: return NOWORK
3     (t,s) = (tail,split) #t, 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 not movesplit: movesplit = 1
9     return NONE #no work

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

21 def grow_shared():
22     new_s = (o_split+head+1)/2
23     split = new_s
24     o_split = new_s
25     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     head = head-1
49     if movesplit: grow_shared()
50     return WORK, head

```

Figure 17: Lace algorithm as described in "Lace: Non-blocking Split Deque for Work-Stealing"[13]