Bounded Context-Switching and Reentrant Locking

Rémi Bonnet[1] and Rohit Chadha[2]

[1] LSV, ENS Cachan & CNRS
[2] University of Missouri

Abstract. Reentrant locking is a *recursive locking* mechanism which allows a thread in a multi-threaded program to acquire the reentrant lock multiple times. The thread must release this lock an equal number of times before another thread can acquire this lock. We consider the control state reachability problem for recursive multi-threaded programs synchronizing via a finite number of reentrant locks. Such programs can be abstracted as multi-pushdown systems with a finite number of counters. The pushdown stacks model the call stacks of the threads and the counters model the reentrant locks. The control state reachability problem is already undecidable for non-reentrant locks. As a consequence, for non-reentrant locks, under-approximation techniques which restrict the search space have gained traction. One popular technique is to limit the number of context switches. Our main result is that the problem of checking whether a control state is reachable within a bounded number of context switches is decidable for recursive multi-threaded programs synchronizing via a finite number of reentrant locks if we restrict the lock-usage to contextual locking: a release of an instance of reentrant lock can only occur if the instance was acquired before in the same procedure and each instance of a reentrant lock acquired in a procedure call must be released before the procedure returns. The decidability is obtained by a reduction to the reachability problem of Vector Addition Systems with States (VASS).

1 Introduction

A mutex lock is a synchronization primitive used in multi-threaded programs to enable communication amongst threads and guide their computations. A lock is either *free* or is *held (owned)* by a thread. If the lock is free then any thread can *acquire it* and in that case the lock is said to be held (owned) by that thread. The lock becomes free when the owning thread *releases* it. If the lock is held by some thread then any attempt to acquire it by any thread (including the owning thread) fails and the requesting thread blocks. However, some programming languages such as Java support non-blocking *reentrant locks*. In a reentrant locking mechanism, if a thread attempts to acquire a reentrant lock it already holds then the thread succeeds. The lock becomes free only when the owning thread releases the lock as many times as it has acquired the lock.

Verification of multi-threaded programs is an important challenge as they often suffer from subtle programming errors. One approach to tackle this challenge is static analysis, and this paper investigates this approach for multi-threaded recursive programs using reentrant locks. In static analysis of sequential recursive programs, a program is often abstracted into a pushdown system that captures the control flow of the program and where the stack models recursion [19]. Several static analysis questions are then formulated as reachability questions on the abstracted pushdown system. In a similar fashion, multi-threaded programs can be abstracted as multi-pushdown systems that synchronize using the synchronization primitives supported by the programming language. Important safety verification questions, such as data-race detection and non-interference, can then be formulated as control state reachability problem: given a global state $\boldsymbol{q}$ of a concurrent program, is $\boldsymbol{q}$ reachable?

The control state reachability problem for multi-pushdown systems is undecidable. As a consequence, under-approximation techniques which restrict the search space have become popular. One such restriction is to *bound the number of context switches* [18]: a context is a contiguous sequence of actions in a computation belonging to the same thread. The *bounded context-switching reachability problem* asks if given a global state $\boldsymbol{q}$, is $\boldsymbol{q}$ reachable within a bounded number of context switches. This was shown to be decidable for multi-threaded programs [18]. Such analyses are used to detect errors in programs.

Our Contributions. In this paper, we study the bounded context-switching reachability problem for multi-threaded recursive programs using a finite set of reentrant locks. Such programs can be abstracted by using standard abstraction techniques as multi-pushdown systems with a finite number of counters. Each counter corresponds to a lock and is used to model the number of times the corresponding lock has been acquired by its owning thread. Acquisition of the corresponding lock increments the counter and a release of the corresponding lock decrements the counter. There is, however, no explicit zero-test on the counters: when a thread P successfully acquires a lock l, it happens either because nobody held l before or P itself held l before. A successful acquisition does not explicitly distinguish these cases. An "explicit" zero-test can, however, be simulated by communication amongst threads.

Furthermore, we restrict our attention to *contextual* reentrant locking: we assume that a release of an instance of a reentrant lock can only occur if this instance was acquired before in the same procedure and that each instance of a reentrant lock acquired in a procedure call is released before the procedure returns. Not only is this restriction natural, several higher-level programming constructs automatically ensure contextual locking. For example, the `synchronized(o) { ...}` statement in Java enforces contextual locking.[1]

Our main result is that the bounded context-switching reachability problem of multi-threaded recursive programs using contextual reentrant locks is decidable. The proof of this fact is carried out in two steps.

[1] Please note that not all uses of reentrant locks in Java are contextual.

First, we associate to each computation a *switching vector*. Switching vectors were introduced in [21,13] for multi-threaded programs. A switching vector is a "snapshot" of the computation at the positions where context switches happen. A switching vector is a sequence; if a computation has r context switches then its switching vector has $r+1$ elements. The i-th element of the switching vector records the global state at the beginning of the i-th context and the active thread in the i-th context. For multi-threaded programs with reentrant locks, the i-th element also records the *lock ownership status*, i.e., which locks are owned by each thread at the start of the i-th context. Observe that the number of switching vectors $\leq r+1$ is finite. Thus, in order to decide whether a global state $\boldsymbol{q}$ is reachable within a bounded number of context switches, it suffices to check whether given a switching vector sig, is there a computation whose switching vector is sig and which leads to $\boldsymbol{q}$. This check is done iteratively: for each prefix sig' of sig, we check if there is a computation whose switching vector is sig'.

The iterative step above is reduced to checking whether a control state of a pushdown counter system is reachable by computations in which at most a bounded number of zero-tests are performed. The status of the latter problem, i.e., whether it is decidable or not is open. However, in our case, we exploit the fact that the constructed pushdown counter system is also *contextual*: the values of a counter in a procedure call is always greater than the value of the counter before the procedure call and the value of the counter immediately before a procedure return is the same as the value of the counter before the procedure call. We show that the control state reachability problem on a *contextual* pushdown counter system with bounded number of zero-tests is decidable. This is achieved by first showing that we only need to consider stacks of bounded height and thus the problem reduces to the problem of checking control-state reachability on counter systems with bounded number of zero-tests. The latter problem is easily seen to be equivalent to the (configuration) reachability problem of vector addition systems (VASS)[2]. The latter is known to be decidable [12,17,15].

We then show that the problem of bounded context-switching reachability is at least as hard as the configuration reachability problem of VASS even when the context switch bound is taken to be 1. Since the configuration reachability problem of VASS is EXPSPACE-hard [3], we conclude that the bounded context-switching reachability problem for VASS is also EXPSPACE-hard.

The rest of the paper is organized as follows. We give our formal model in Section 2. The result of deciding reachability in contextual pushdown counter systems with a bounded number of zero-tests is given in Section 3 and our main result in Section 4. We conclude and discuss future work in Section 5.

Related Work. For multi-threaded programs (without reentrant locks), bounded context-switching reachability problem was first posed and shown to be decidable in [18]. Several different proofs of this fact have been discovered since then (see, for example, [16,13,21]). The technique of switching vectors that we have

[2] For our purposes, VASS are counter systems in which there are no zero-test transitions.

adapted to establish our result for the case of multi-threaded programs with reentrant locks was first introduced in [13,21]. (Please note that switching vectors sometimes go by the name of interfaces).

For non-reentrant locks, it was shown in [10] that if we abstract away the data and assume that threads follow *nested locking* then the control state reachability (even with unbounded context-switching) is decidable. A thread is said to follow nested locking [10] if locks are released in the reverse order in which they are acquired. For contextual (non-reentrant) locking, we showed a similar result for 2-threaded programs in [4].

For reentrant locks, [11,14] observe that if threads follow both contextual and nested locking, then the stack of a thread can be used to keep track of both recursive calls as well as recursive lock acquisitions. Thus, the bounded context-switching reachability problem in this case reduces to the case of bounded context-switching reachability problem in multi-pushdown systems. The restriction of both contextual and nested locking is naturally followed by many programs. In Java, for example, if *only* `synchronized(o) { ...}` blocks are used for synchronization then locking is both contextual and nested. However, the assumption of nested locking in presence of other synchronizing primitives, for example, `wait/notify{ ...}` construct can break nested locking while preserving contextual locking.

A modular approach for verifying concurrent non-recursive Java programs with reentrant locks is proposed in [2]. In this approach, first a "lock interface" s guessed: a "lock interface" characterizes the sequence of lock operations that can happen in an execution of the program. Then, they check if each thread respects the lock interface. Since the number of possible lock interfaces is infinite, termination is not guaranteed. Thus, they check programs against specific lock interfaces and thus this approach is another way of restricting the search space.

The control state reachability problem for pushdown counter systems with no zero-tests has long been an open problem. The only non-trivial cases that we are aware of, for which decidability has been established, is when counters are decremented only the stack contents are empty [20,9,5] (in which case it is EXPSPACE-complete [6]), or when the stack is restricted to *index-bounded* behaviors [1] (equivalent to VASS with hierarchical zero-tests, complexity unknown), or when the number counter reversals are bounded [7,8] (in which case it is NP-complete).

2 Model

The set of natural numbers shall be denoted by $\mathbb{N}$. The set of functions from a set A to B shall be denoted by B^A. Given a function $f \in B^A$, the function $f|_{a\mapsto b}$ shall be the unique function g defined as follows: $g(a) = b$ and for all $a' \neq a$, $g(a') = f(a')$. If $\bar{a} = (a_1, \ldots, a_n) \in A_1 \times \cdots \times A_n$ then $\pi_i(a) = a_i$ for each $1 \leq i \leq n$.

Pushdown Systems. Recursive programs are usually modeled as pushdown systems for static analysis. We are modeling threads in concurrent recursive programs that synchronize via reentrant locks. A single thread can be modeled as follows:

Definition 1. *Given a finite set* Lcks*, a pushdown system (PDS)* $\mathcal{P}$ *using (reentrant locks)* Lcks *is a tuple* (Q, Γ, qs, δ) *where*

- Q *is a finite set of control states.*
- Γ *is a finite stack alphabet.*
- qs *is the initial state.*
- $\delta = \delta_{\mathsf{int}} \cup \delta_{\mathsf{cll}} \cup \delta_{\mathsf{rtn}} \cup \delta_{\mathsf{acq}} \cup \delta_{\mathsf{rel}}$ *is a finite set of transitions where*
  - $\delta_{\mathsf{int}} \subseteq Q \times Q$.
  - $\delta_{\mathsf{cll}} \subseteq Q \times (Q \times \Gamma)$.
  - $\delta_{\mathsf{rtn}} \subseteq (Q \times \Gamma) \times Q$.
  - $\delta_{\mathsf{acq}} \subseteq Q \times (Q \times \mathsf{Lcks})$.
  - $\delta_{\mathsf{rel}} \subseteq (Q \times \mathsf{Lcks}) \times Q$.

  A transition in δ_{int} *is said to be a* state transition*, a transition in* δ_{cll} *is said to be a* push transition*, a transition in* δ_{rtn} *is said to be a* pop transition*, a transition in* δ_{acq} *is said to be an* acquire transition *and a transition in* δ_{rel} *is said to be a* release transition*.*

The semantics of a PDS $\mathcal{P}$ using Lcks is given as a transition system. The set of configurations of $\mathcal{P}$ using Lcks is $\mathsf{Conf}_{\mathcal{P}} = Q \times \Gamma^* \times \mathbb{N}^{\mathsf{Lcks}}$. Intuitively, the elements of a configuration (q, w, hld) have the following meaning: q is the "current" control state of $\mathcal{P}$, w the contents of the pushdown stack and $\mathsf{hld} : \mathsf{Lcks} \to \mathbb{N}$ is a function that tells the number of times each lock has been acquired by $\mathcal{P}$. The transition relation is not a relation between configurations of $\mathcal{P}$, since a thread *executes* in an *environment*, namely the set of free locks (i.e., locks not being held by any thread). Thus, the transition relation can be seen as a binary relation on $2^{\mathsf{Lcks}} \times \mathsf{Conf}_{\mathcal{P}}$, i.e., a transition takes a pair (fr, c) (set of free locks and the configuration) and gives the resulting pair (fr', c'). In order to emphasize that the set of free locks is an "environment", we shall write $fr : c$ instead of the usual notation of (fr, c). In addition to the usual push, pop and internal actions of a PDS; a thread can acquire or release a lock. The thread can only acquire a lock if it is either free or was held by itself before. The thread can only release a lock if it is held by itself. A lock held by a thread is freed only after the thread releases all instances held by it. Formally,

Definition 2. *A PDS* $\mathcal{P} = (Q, \Gamma, qs, \delta)$ *using* Lcks *gives a labeled transition relation* $\longrightarrow_{\mathcal{P}} \subseteq (2^{\mathsf{Lcks}} \times (Q \times \Gamma^* \times \mathbb{N}^{\mathsf{Lcks}})) \times \mathsf{Labels} \times (2^{\mathsf{Lcks}} \times (Q \times \Gamma^* \times \mathbb{N}^{\mathsf{Lcks}}))$ *where* $\mathsf{Labels} = \{\mathsf{int}, \mathsf{cll}, \mathsf{rtn}\} \cup \{\mathsf{acq}(l), \mathsf{rel}(l) \mid l \in \mathsf{Lcks}\}$ *and* $\longrightarrow_{\mathcal{P}}$ *is defined as follows.*

- $\mathsf{fr} : (q, w, \mathsf{hld}) \xrightarrow{\mathsf{int}}_{\mathcal{P}} \mathsf{fr} : (q', w, \mathsf{hld})$ *if* $(q, q') \in \delta_{\mathsf{int}}$.
- $\mathsf{fr} : (q, w, \mathsf{hld}) \xrightarrow{\mathsf{cll}}_{\mathcal{P}} \mathsf{fr} : (q', wa, \mathsf{hld})$ *if* $(q, (q', a)) \in \delta_{\mathsf{cll}}$.
- $\mathsf{fr} : (q, wa, \mathsf{hld}) \xrightarrow{\mathsf{rtn}}_{\mathcal{P}} \mathsf{fr} : (q', w, \mathsf{hld})$ *if* $((q, a), q') \in \delta_{\mathsf{rtn}}$.
- $\mathsf{fr} : (q, w, \mathsf{hld}) \xrightarrow{\mathsf{acq}(l)}_{\mathcal{P}} \mathsf{fr} \setminus \{l\} : (q', w, \mathsf{hld}|_{l \mapsto \mathsf{hld}(l)+1})$ *if* $(q, (q', l)) \in \delta_{\mathsf{acq}}$ *and either* $l \in \mathsf{fr}$ *or* $\mathsf{hld}(l) > 0$.

– $\mathsf{fr} : (q, w, \mathsf{hld}) \xrightarrow{\mathsf{rel}(l)}_{\mathcal{P}} \mathsf{fr} : (q', w, \mathsf{hld}|_{l \mapsto \mathsf{hld}(l)-1})$ *if* $((q, l), q') \in \delta_{\mathsf{rel}}$ *and* $\mathsf{hld}(l) > 1$.
– $\mathsf{fr} : (q, w, \mathsf{hld}) \xrightarrow{\mathsf{rel}(l)}_{\mathcal{P}} \mathsf{fr} \cup \{l\} : (q', w, \mathsf{hld}|_{l \mapsto 0})$ *if* $((q, l), q') \in \delta_{\mathsf{rel}}$ *and* $\mathsf{hld}(l) = 1$.

2.1 Multi-pushdown Systems

Concurrent programs are usually modeled as multi-pushdown systems. For our paper, we assume that threads in a concurrent program also synchronize through reentrant locks which leads us to the following definition.

Definition 3. *Given a finite set* Lcks*, a n-pushdown system (n-PDS) $\mathcal{CP}$ communicating via (reentrant locks)* Lcks *and shared state Q is a tuple $(\mathcal{P}_1, \ldots, \mathcal{P}_n)$ where*

– *Q is a finite set of states.*
– *Each $\mathcal{P}_i$ is a PDS using* Lcks.
– *The set of control states of $\mathcal{P}_i$ is $Q \times Q_i$. Q_i is said to be the set of* local states *of thread i.*
– *There is a $qs \in Q$ s.t. for each i the initial state of $\mathcal{P}_i$ is (qs, qs_i) for some $qs_i \in Q_i$. The state qs is said to be the* initial shared state *and qs_i is said to be the* initial local state *of $\mathcal{P}_i$.*

Given a n-PDS $\mathcal{CP}$, we will assume that the set of local states and the stack symbols of the threads are mutually disjoint.

Definition 4. *The semantics of a n-PDS $\mathcal{CP} = (\mathcal{P}_1, \ldots, \mathcal{P}_n)$ communicating via* Lcks *and shared state Q is given as a labeled transition system $T = (S, s_0, \longrightarrow)$ where*

– *S, said to be the set of configurations of $\mathcal{CP}$, is the set $Q \times (Q_1 \times \Gamma_1^* \times \mathbb{N}^{\mathsf{Lcks}}) \times \cdots \times (Q_n \times \Gamma_n^* \times \mathbb{N}^{\mathsf{Lcks}})$ where Q_i is the set of local states of $\mathcal{P}_i$ and Γ_i is the stack alphabet of $\mathcal{P}_i$.*
– *s_0, said to be the initial configuration, is $(qs, (qs_1, \epsilon, \overline{0}), \cdots, (qs_m, \epsilon, \overline{0}))$ where qs is the initial shared state, qs_i is the initial local state of $\mathcal{P}_i$ and $\overline{0} \in \mathbb{N}^{\mathsf{Lcks}}$ is the function which takes the value 0 for each $l \in$* Lcks.
– *The set of labels on the transitions is* $\mathsf{Labels} \times \{1, \ldots, n\}$ *where* $\mathsf{Labels} = \{\mathsf{int}, \mathsf{cll}, \mathsf{rtn}\} \cup \{\mathsf{acq}(l), \mathsf{rel}(l) \mid l \in \mathsf{Lcks}\}$. *The labeled transition relation $\xrightarrow{(\lambda,i)}$ is defined as follows*

$$(q, (q_1, w_1, \mathsf{hld}_1), \cdots, (q_n, w_n, \mathsf{hld}_n)) \xrightarrow{(\lambda,i)} (q', (q_1', w_1', \mathsf{hld}_1'), \cdots, (q_n', w_n', \mathsf{hld}_n'))$$

iff for all $j \neq i$, $q_j = q_j'$, $w_j = w_j'$ and $\mathsf{hld}_j = \mathsf{hld}_j'$ and

$$\mathsf{Lcks} \setminus \{l \mid \cup_{1 \le r \le n} \mathsf{hld}_r(l) > 0\} : ((q, q_i), w_i, \mathsf{hld}_i) \xrightarrow{\lambda}_{\mathcal{P}_i} \mathsf{Lcks} \setminus \{l \mid \cup_{1 \le r \le n} \mathsf{hld}_r'(l) > 0\} : ((q', q_i'), w_i', \mathsf{hld}_i').$$

Notation: A *global state* is $(q, q_1, \ldots, q_n)$ where $q \in Q$ and $q_i \in Q_i$. Given a configuration $s = (q, (q_1, w_1, \mathsf{hld}_1), \cdots, (q_n, w_n, \mathsf{hld}_n))$ of a n-PDS $\mathcal{CP}$, we say that $\mathsf{ShdSt}(s) = q$, $\mathsf{GlblSt}(s) = (q, q_1, \cdots, q_n)$, $\mathsf{LckHld}(s) = (\mathsf{hld}_1, \cdots, \mathsf{hld}_n)$, $\mathsf{LckOwnd}(s) = (held_1, \cdots, held_n)$ and $\mathsf{LckOwnd}_i(s) = held_i$ where $held_i = \{l \mid \mathsf{hld}_i(l) > 0\}$, $\mathsf{Conf}_i(s) = (q_i, w_i, \mathsf{hld}_i)$, $\mathsf{CntrlSt}_i(s) = q_i$, $\mathsf{Stck}_i(s) = w_i$, $\mathsf{StHt}_i(s) = |w_i|$, the length of w_i and $\mathsf{LckHld}_i(s) = \mathsf{hld}_i$.

Computations. A *computation* of the n-PDS $\mathcal{CP}$, is a sequence $\sigma = s_0 \xrightarrow{(\lambda_1,i_1)} s_1 \cdots \xrightarrow{(\lambda_m,i_m)} s_m$, such that s_0 is the initial configuration of $\mathcal{CP}$. The transition $s_j \xrightarrow{(\mathsf{cll},i)} s_{j+1}$ is said to be a *procedure call by thread i*. Similarly, we can define *procedure return*, *internal action*, *acquisition of lock l* and *release of lock l* by thread i. A procedure return $s_j \xrightarrow{(\mathsf{rtn},i)} s_{j+1}$ is said to *match* a procedure call $s_p \xrightarrow{(\mathsf{cll},i)} s_{p+1}$ iff $p < j$, $\mathsf{StHt}_i(s_p) = \mathsf{StHt}_i(s_{j+1})$ and for all $p+1 \leq t \leq j$, $\mathsf{StHt}_i(s_{p+1}) \leq \mathsf{StHt}_i(s_t)$. A release of a lock $s_j \xrightarrow{(\mathsf{rel}(l),i)} s_{j+1}$ is said to *match* an acquisition $s_p \xrightarrow{(\mathsf{acq}(l),i)} s_{p+1}$ iff $p < j$, $\mathsf{LckHld}_i(s_p)(l) = \mathsf{LckHld}_i(s_{j+1})(l)$ and for each $p+1 \leq t \leq j$, $\mathsf{LckHld}_i(s_{p+1})(l) \leq \mathsf{LckHld}_i(s_t)(l)$.

2.2 Contextual Locking

We recall the notion of *contextual locking* [4] and adapt the notion to the reentrant locking mechanism. Informally, contextual locking means that –

- each instance of a lock acquired by a thread in a procedure call must be released before the corresponding return is executed, and
- the instances of locks held by a thread just before a procedure call is executed are not released during the execution of the procedure.

Formally,

Definition 5. *A thread i in a n-PDS $\mathcal{CP} = (\mathcal{P}_1, \ldots, \mathcal{P}_n)$ is said to follow contextual locking if whenever $s_\ell \xrightarrow{(\mathsf{cll},i)} s_{\ell+1}$ and $s_j \xrightarrow{(\mathsf{rtn},i)} s_{j+1}$ are matching procedure call and return along a computation $s_0 \xrightarrow{(\lambda_1,i)} s_1 \cdots \xrightarrow{(\lambda_m,i)} s_m$, we have that*

$$\mathsf{LckHld}_i(s_\ell) = \mathsf{LckHld}_i(s_{j+1}) \text{ and for all } \ell \leq r \leq j.\ \mathsf{LckHld}_i(s_\ell) \leq \mathsf{LckHld}_i(s_r).$$

Example 1. Consider the 3-threaded program shown in Figure 1. Threads P0 and P1 follow contextual locking, but thread P2 does not follow contextual locking.

2.3 Bounded Context-Switching

A context [18] is a contiguous sequence of actions in a computation belonging to the same thread:

Definition 6. *Given a computation $\sigma = s_0 \xrightarrow{(\lambda_1,i_1)} s_1 \cdots \xrightarrow{(\lambda_m,i_m)} s_m$ of a n-PDS $\mathcal{CP}$, we say that a context switch happens at position $j \in \{1, \cdots, m\}$ if $i_j \neq i_{j+1}$. The number of context switches in σ is the number of positions at which a context switch happens.*

The bounded context-switching reachability problem [18] is defined formally as:

```
int a(){
   acq l1;
   acq l2;
   if (..) then{
         ...
         rel l2;
         rel l1;
       };
   else{
        ...
        rel l1
        rel l2
       };
     return i;
};

public void P0() {
  n=a();
}
```

```
int b(){
    acq l1;
    ...
    rel l1;
    return j;
 };

 public void P1() {
   l=a();
 }
```

```
int c(){
   rel l2;
   acq l1;
   ...
   return i;
};
 public void P2(){
 acq l2;
 n=c();
 rel l1;
}
```

Fig. 1. Threads P0 and P1 follow contextual locking. Thread P2 does not follow contextual locking.

Definition 7. *For $k \in \mathbb{N}$, the k-bounded context-switching reachability problem asks that given a n-PDS $\mathcal{CP} = (\mathcal{P}_1, \ldots, \mathcal{P}_n)$ communicating via* Lcks *and shared state Q, and a global state $\boldsymbol{q}$ of $\mathcal{CP}$, if there is a computation $\sigma = s_0 \xrightarrow{(\lambda_1, i_1)} s_1 \cdots \xrightarrow{(\lambda_m, i_m)} s_m$ of $\mathcal{CP}$ such that i)* $\mathsf{GlblSt}(s_m) = \boldsymbol{q}$ *and ii) there are at most k context switches in σ.*

3 Contextual Pushdown Counter Systems

In order to establish our main result, we shall need an auxiliary result about pushdown counter systems. A pushdown counter system is an automaton which in addition to a pushdown stack also has counters. Formally, a k-counter pushdown system (k-counter PDS), $\mathcal{M}$, is a tuple (Q, Γ, qs, δ) where Q is a finite set of *control* states, Γ is a finite *stack alphabet*, qs is the *initial* state and δ, the set of *transitions* of $\mathcal{M}$, is a tuple $(\delta_{\mathsf{int}}, \delta_{\mathsf{cll}}, \delta_{\mathsf{rtn}}, \{\delta_{inc_i}, \delta_{dec_i}, \delta_{z_i}\}_{1 \le i \le k})$ where $\delta_{\mathsf{int}} \subseteq Q \times Q$ is the set of *state* transitions, $\delta_{\mathsf{cll}} \subseteq Q \times (Q \times \Gamma)$ is the set of *push* transitions, $\delta_{\mathsf{rtn}} \subseteq (Q \times \Gamma) \times Q$ is the set of *pop* transitions, and for each $1 \le i \le k$, $\delta_{inc_i} \subseteq Q \times Q$ is the set of *increment* transitions of the counter i, $\delta_{dec_i} \subseteq Q \times Q$ is the set of *decrement* transitions of the counter i and $\delta_{z_i} \subseteq Q \times Q$ is the set of *zero-tests* of the counter i.

The semantics of the k-counter PDS $\mathcal{M}$ is given in terms of a labeled transition system $\to_{\mathcal{M}}$. The definition of the semantics is as expected; we set out some notations here. The set of configurations of the transition system is the set

$Q \times \Gamma^* \times \mathbb{N}^k$. In a configuration $(q, w, j_1, \cdots, j_k)$, q is the control state, $w \in \Gamma^*$ is the stack contents and $j_i \in \mathbb{N}$ is the value of the i-th counter. The set of transition labels are $\{\mathsf{int}, \mathsf{cll}, \mathsf{rtn}\} \cup \{inc_i, dec_i, z_i \mid 1 \leq i \leq k\}$. The initial configuration is $s_0 = (qs, \epsilon, 0, \cdots, 0)$. The definition of computations and the definition of matching push and pop transitions along a computation are as expected.

Given a k-counter PDS $\mathcal{M} = (Q, \Gamma, qs, \delta)$ and a computation $C = s_0 \xrightarrow{\lambda_1}_{\mathcal{M}} s_1 \cdots \xrightarrow{\lambda_m}_{\mathcal{M}} s_m$ of $\mathcal{M}$, the *number of zero-tests* along the computation is $|\{\lambda_j \mid \lambda_j = z_i \text{ for some } 1 \leq i \leq n\}|$. We are interested in the problem of checking whether a control state is reachable by computations in which the number of zero-tests are bounded. However, we will be only interested in contextual counter PDSs. Contextual counter PDSs are analogous to threads that follow contextual locking; in any computation, a) there are an equal number of increments and decrements in a procedure call and b) counter values during procedure call are at least as large as the counter values before the procedure call. Formally,

Definition 8. *The k-counter PDS $\mathcal{M}$ is said to be* contextual *if whenever $s_\ell \xrightarrow{\mathsf{cll}}_{\mathcal{M}} s_{\ell+1}$ and $s_j \xrightarrow{\mathsf{rtn}}_{\mathcal{M}} s_{j+1}$ are matching push and pop transitions along a computation $s_0 \xrightarrow{\lambda_1}_{\mathcal{M}} s_1 \cdots \xrightarrow{\lambda_m}_{\mathcal{M}} s_m$ then for each $1 \leq i \leq k$, a) $c_i(s_\ell) = c_i(s_{j+1})$ and b) for all each $\ell \leq r \leq j$, $c_i(s_\ell) \leq c_i(s_r)$, where $c_i(s)$ is the value of i-th counter in the configuration s.*

In order to establish our result about contextual counter PDSs, we need one auxiliary lemma. The stack in configuration $s = (q, w, j_1, \cdots, j_k)$ is said to be *strictly larger than the stack* in configuration $s' = (q', w', j'_1, \cdots, j'_k)$ if $w = w'u$ where u is a nonempty word over Γ.

Lemma 1. *Let $\mathcal{M} = (Q, \Gamma, qs, \delta)$ be a k-counter contextual PDS. Consider three computations of $\mathcal{M}$:*

$$\begin{aligned}
C_1 &: (q_1, w, j_1, \ldots, j_k) \xrightarrow{cll}_{\mathcal{M}} \ldots (q_1, ww', j'_1, \ldots, j'_k)\\
C_2 &: (q_1, ww', j'_1, \ldots, j'_k) \xrightarrow{cll}_{\mathcal{M}} \ldots \xrightarrow{rtn}_{\mathcal{M}} (q_2, ww', j'_1, \ldots, j'_k)\\
C_3 &: (q_1, ww', j'_1, \ldots, j'_k) \ldots \xrightarrow{rtn}_{\mathcal{M}} (q_2, w, j_1, \ldots, j_k)
\end{aligned}$$

such that the stack stays strictly larger than w in the intermediate states of C_1 and C_3 and stays strictly larger than ww' (with w' non-empty) in the intermediate states of C_2. Then, if $\{i \mid j_i = 0\} = \{i \mid j'_i = 0\}$ then there is a computation from $(q_1, w, j_1, \ldots, j_k)$ leading to $(q_2, w, j_1, \ldots, j_k)$ by using exactly the transitions used in C_2.

Proof. Consider the computation $C_2 = (q_1, ww', j'_1, \ldots, j'_k) \xrightarrow{cll}_{\mathcal{M}} s'_0 \xrightarrow{\lambda_1}_{\mathcal{M}} \ldots \xrightarrow{\lambda_p}_{\mathcal{M}} s'_p \xrightarrow{rtn}_{\mathcal{M}} (q_2, ww', l'_1, \ldots, l'_k)$. As the stack stays strictly larger than ww' during this computation, it means that the initial call matches the final return. Therefore as $\mathcal{M}$ is contextual, the counter values in any intermediate state of C_2 are at least as large as $(j'_1, \ldots, j'_k)$; and by contextuality, $(j'_1, \ldots, j'_k)$ is itself larger than $(j_1, \ldots, j_k)$.

We establish the following invariant by induction on t: if $(q_2, ww', j'_1, \dots, j'_k) \xrightarrow{cll}_{\mathcal{M}} s'_0 \cdots \xrightarrow{\lambda_t}_{\mathcal{M}} (q_3, ww'\sigma, l'_{t,1}, \dots, l'_{t,n})$ then $(q_1, w, j_1, \dots, j_k) \dots \xrightarrow{\lambda_t}_{\mathcal{M}} (q_3, w\sigma, l_{t,1}, \dots, l_{t,n})$ with $l'_{t,i} = l_{t,i} + (j'_i - j_i)$. The base case, $t = 0$, is immediate. In the inductive step, we proceed by cases on the label λ_t. The case of push, pop and internal state transitions is immediate because the w' part of the stack is never used in C_2. A transition that increments a counter also fulfills this invariant immediately. We now consider a transition that can decrement a counter i. Then, because $\mathcal{M}$ is contextual, we have $l'_{t+1,i} \geq j'_i$ which means that $l'_t \geq j'_i + 1$, and thus $l_{t,i} = (l'_{t,i} - j'_i) + j_i \geq 1$ and the decrement can be performed. For a zero-test, we have that $l'_{t,i} = 0$, which means by contextuality that $j'_i = j_i = 0$. Thus, by induction hypothesis, $l_{t,i} = 0$ and the zero-test can be performed. This concludes the demonstration of the invariant and the statement of the lemma follows directly. $\square$

We are ready to establish that the control state reachability problem for contextual pushdown counter systems with a bounded number of zero tests is decidable.

Theorem 1. *The following problem is decidable: Given a k-counter contextual PDS $\mathcal{M}$ with initial configuration s_0, a control state q of $\mathcal{M}$ and a number $r \in \mathbb{N}$, check if there is a computation $C = s_0 \xrightarrow{\lambda_1}_{\mathcal{M}} s_1 \cdots \xrightarrow{\lambda_m}_{\mathcal{M}} s_m$ s.t.*

- *there are at most r zero-tests along C, and*
- *$s_m = (q, w, j_1, \cdots j_k)$ for some $w \in \Gamma^*$, $j_1, \cdots, j_k \in \mathbb{N}$.*

Proof. We first turn the problem into one where the final state must have an empty stack as follows.

We first encode in the control states of the PDS counter system the information about whether the stack is empty as follows. When symbol a is to be pushed on an empty stack, we push a marked symbol a^* instead. Popping a marked symbol indicates that the resulting stack is empty.

Now, if a stack symbol is pushed but never popped in a computation leading to q; it means that this stack symbol is never subsequently accessed, and thus can be ignored. Therefore, when the stack is empty and a symbol is ready to be pushed, we allow a non-deterministic choice: either to push the symbol (guessing that it would have been popped later) or to not push it (guessing that it would have never been popped subsequently). In the latter case, we perform the same change of control state. As the discarded symbols were at the bottom of the stack, we don't expose symbols that could be used in the computation.

The resulting system is still contextual (because any transition sequence between matching push and pop in the new system was already present in the old one). Moreover, one can reach a state $(q, \epsilon, j_1, \dots, j_k)$ for some $(j_1, \dots, j_k)$ in the new system if and only if one could reach $(q, w, j_1, \dots, j_k)$ for some $(j_1, \dots, j_k)$ and w in the old one.

For the case with a final empty stack, we show that if such a computation exists, then there exists one such that the stack size in any intermediate state is bounded by $|Q|^2 2^k$. Indeed, if we assume this is not the case, fix a computation C whose length is minimal amongst computations ending in control state q

with empty stack. By assumption, there are at least $|Q|^2 2^k + 1$ nested pairs of matching push and pop in C. But, if to each pair of matching push and pop $(q_1, w, j_1, \ldots, j_k) \xrightarrow{cll}_{\mathcal{M}} \ldots \xrightarrow{rtn}_{\mathcal{M}} (q_2, w, j_1, \ldots, j_k)$ in C we associate the vector $(q_1, q_2, \{i \mid j_i = 0\})$, by the pigeonhole principle, there exist two nested pairs such that:

$$\begin{aligned}(q_1, w, j_1, \ldots, j_k) &\xrightarrow{cll}_{\mathcal{M}} \ldots (q_1, ww', j'_1, \ldots, j'_k)\\ &\xrightarrow{cll}_{\mathcal{M}} \ldots \xrightarrow{rtn}_{\mathcal{M}} (q_2, ww', j'_1, \ldots, j'_k)\\ &\ldots \xrightarrow{rtn}_{\mathcal{M}} (q_2, w, j_1, \ldots, j_k)\end{aligned}$$

such that a) (due to the fact that we are consider matching pushes and pops) the stack stays strictly larger than w in the intermediate states of the first and third part and stays strictly larger than ww' (with w' non-empty) during the second part, and b) $\{i \mid j_i = 0\} = \{i \mid j'_i = 0\}$. Thanks to Lemma 1, we get a shorter computation, which contradicts the assumption of minimality of C.

Now, because the stack is bounded, it means we can just encode it in the control state, which gives us a reachability problem in a counter system with restricted zero-tests. We reduce it to reachability in Vector Addition Systems [12,17,15]. As the number of possible zero-tests is known, we encode in the control state the number of zero-tests remaining, and work on t copies of each counter, where t is the number of zero-tests remaining. When a zero-test is performed, we only change the control state, remembering the index of the counter on which the zero-test is supposed to be performed, continue to work on $t-1$ copies of the counters, leaving the remaining counters frozen. At the end of the computation, we test whether all frozen counters which should have been zero when they were frozen are indeed zero. □

4 Bounded Context-Switching Reachability

We shall now establish the decidability of the bounded context-switching reachability problem. A key technique we will use is the technique of *switching vectors* developed for bounded context-switching reachability for multi-pushdown systems [21,13]. Intuitively, a switching vector is "snapshot" of a computation in a multi-pushdown system: it is the sequence of active threads and the global states at the beginning of a context in the computation. We extend this definition to n-PDS communicating via reentrant locks by also taking into account which locks are held by which thread at the positions where context-switches happen.

We start by fixing some definitions. Fix a n-PDS $\mathcal{CP} = (\mathcal{P}_1, \ldots, \mathcal{P}_n)$ communicating via Lcks and shared state Q. Let Q_i be the set of local states of $\mathcal{P}_i$. Recall that a *global state* is $(q, q_1, \ldots, q_n)$ where $q \in Q$ and $q_i \in Q_i$; given a configuration $s = (q, (q_1, w_1, \mathsf{hld}_1), \ldots, (q_n, w_n, \mathsf{hld}_n))$, $\mathsf{GlblSt}(s) = (q, q_1, \cdots, q_n)$, $\mathsf{LckOwnd}(s) = (held_1, \cdots, held_n)$ and $\mathsf{LckOwnd}_i(s) = held_i$ where $held_i = \{l \,|\, \mathsf{hld}_i(l) > 0\}$. We say that $\mathsf{LckOwnd}_i(s)$ is the set of locks *owned by* $\mathcal{P}_i$ and the tuple $\mathsf{LckOwnd}(s)$ is the *lock ownership status.* We are ready to define switching vectors formally.

Definition 9. *Let $\mathcal{CP} = (\mathcal{P}_1, \ldots, \mathcal{P}_n)$ be a n-PDS communicating via* Lcks *and shared state Q. For each $1 \leq i \leq n$, let Q_i be the set of local states of $\mathcal{P}_i$. A sequence $(gs_0, ls_0, p_0), \cdots, (gs_r, ls_r, p_r)$ is said to be a $\mathcal{CP}$-*switching vector *if the following holds for each $0 \leq t \leq r$:*

- *p_t is an element of the set $\{1, \cdots, n\}$ and for $0 \leq t < r$, $p_t \neq p_{t+1}$.*
- *$gs_t \in Q \times Q_1 \times \cdots \times Q_n$. gs_0 is the global state of the initial configuration, and for all $t > 0$, If $gs_{t-1} = (q, q_1, \cdots, q_n)$ and $gs_t = (q', q'_1, \ldots, q'_n)$ then $q_x = q'_x$ for each $x \neq pr_{t-1}$.*
- *$ls_t \in (2^{\mathsf{Lcks}})^n$, $ls_0 = (\emptyset, \ldots, \emptyset)$, and for all $t > 0$, if $ls_t = (held'_1, \ldots, held'_n)$ then $held'_y \cap held'_z = \emptyset$ for each $y \neq z$; and if $ls_{t-1} = (held_1, \ldots, held_n)$ then $held_x = held'_x$ for each $x \neq pr_{t-1}$.*

Note that the last two conditions are consistency checks: an active thread cannot affect the local states of other threads and the locks owned by them. The following definition captures the intuitive meaning of a switching vector being the "snapshot" of a computation.

Definition 10. *Let $\mathcal{CP} = (\mathcal{P}_1, \ldots, \mathcal{P}_n)$ be a n-PDS and let $sig = (gs_0, ls_0, p_0), \cdots, (gs_r, ls_r, p_r)$ be a $\mathcal{CP}$-switching vector. We say that a computation $C = s_0 \xrightarrow{(\lambda_1, i_1)} s_1 \cdots \xrightarrow{(\lambda_m, i_m)} s_m$ of $\mathcal{CP}$ is* compatible with *sig if*

- *C has r context switches.*
- *$gs_0 = \mathsf{GlblSt}(s_0), ls_0 = (\emptyset, \cdots, \emptyset)$ and $p_0 = i_1$.*
- *If the context switches occur at positions $j_1, \cdots, j_r$ then for each $1 \leq t \leq r$,*
  - *$gs_t = \mathsf{GlblSt}(s_t)$.*
  - *$p_t = i_{j_t+1}$.*
  - *If $ls_t = (held_1, \ldots, held_n)$ then $\mathsf{LckOwnd}_i(s_{j_t}) \subseteq held_i$ for each $1 \leq i \leq n$.*
  - *Let j_{r+1} be m. If $ls_t = (held_1, \ldots, held_n)$ then in the sequence $s_{j_t} \xrightarrow{(\lambda_{j_{t+1}}, p_t)} \cdots \xrightarrow{(\lambda_{j_{t+1}}, p_t)} s_{j_{t+1}}$, the thread p_t does not do any lock acquisitions and releases of locks in the set $\cup_{i \neq p_t} held_i$.*

It is easy to see that that if $\boldsymbol{q}$ is reachable by a computation C at most r bounded context-switches then C must be compatible with a switching vector sig of length $\leq r+1$ (the compatible switching vector is the sequence of the global state, the identifier of the active thread and lock ownership status at the beginning of each context). Hence, we can decide the bounded context-switching reachability problem if we can give an algorithm that given an a n-PDS $\mathcal{CP}$ communicating via Lcks, a $\mathcal{CP}$-switching vector sig and a global state $\boldsymbol{q}$, checks if there is a computation C compatible with sig leading to $\boldsymbol{q}$. We establish this result next.

Lemma 2. *The following problem is decidable:*

Given a n-PDS $\mathcal{CP} = (\mathcal{P}_1, \ldots, \mathcal{P}_n)$ communicating via Lcks *and shared state Q, s.t. each thread is contextual, a $\mathcal{CP}$-switching vector sig and a global state $\boldsymbol{q}$ of $\mathcal{CP}$, is there a computation that is a) compatible with sig and b) ends in global state $\boldsymbol{q}$?*

Proof. We give an algorithm that decides the above problem. Note if $r = 0$, then we can decide the problem by using Theorem 1. So, we only consider the case $r > 0$. Let num_{locks} be the cardinality of Lcks. Fix an enumeration $l_1, l_2, \cdots, l_{num_{locks}}$ of the elements of Lcks. Let $\mathcal{P}_i = (Q \times Q_i, \Gamma_i, (qs, qs_i), \delta_i)$ where qs_i is the initial local state of $\mathcal{P}_i$. Let $sig = (gs_0, ls_0, p_0), \cdots, (gs_r, ls_r, p_r)$. For each $1 \le t \le r$, let sig_t be $(gs_0, ls_0, p_0), \cdots, (gs_t, ls_t, p_t)$. First note that if $\boldsymbol{q} = (q, q_1, \ldots, q_n)$ and $gs_r = (q', q'_1, \ldots, q'_n)$ then for each $i \neq p_r$, q_i must be the same q'_i (since p_r is the last active thread). Therefore the algorithm immediately outputs "NO" if there is some $i \neq p_r$ s.t. $q_i \neq q'_i$.

Otherwise, the algorithm proceeds iteratively and will have at most $r + 1$ iterations. At the end of each iteration $t \le r$, the algorithm will either output "NO" or move to the next iteration. If the algorithm outputs "NO" at the end of iteration $t < r$, then it would mean that there is no computation of $\mathcal{CP}$ compatible with sig_t. If the algorithm moves to the next iteration then it would mean that there is a computation of $\mathcal{CP}$ compatible with sig_t ending in a configuration s such that $\mathsf{GlblSt}(s) = gs_{t+1}$ and $\mathsf{LckOwnd}_i(s) \subseteq \pi_i(ls_{t+1})$ for each $1 \le i \le n$.

In each iteration t, the algorithm constructs n pushdown counter systems $\mathcal{M}_1^t, \ldots, \mathcal{M}_n^t$. Intuitively, the pushdown counter system $\mathcal{M}_i^t$ will "simulate" the actions of the ith thread up-to the t-th context switch. Each $\mathcal{M}_i^t$ has num_{locks} counters: the counter j keeps track of number of times lock l_j has been acquired by thread i. The algorithm proceeds as follows. For the sake of brevity, we only illustrate the first iterative step. The other iterative steps are similar.

– (Iterative step 1.) For each $1 \le i \le n$, we pick new states $q_i^\dagger, test_{i,1}, \ldots, test_{i,n}$ and let $Q_{new,i} = \{q_i^\dagger, test_{i,1}, \ldots, test_{i,num_{locks}}\}$.
  In the first iterative step, the active thread is supposed to be p_0. For $i \neq p_0$, let $\mathcal{M}_i^1 = (Q_i^1, \Gamma_i, qs_i^1, \delta_i^1)$ be the num_{locks}-counter PDS where $Q_i^1 = Q_{new,i} \times \{1\}$, $qs_i^1 = (q_i^\dagger, 1)$ and δ_i^1 is $\emptyset$.
  The num_{locks}-counter PDS $\mathcal{M}_{p_0}^1 = (Q_{p_0}^1, \Gamma_i, qs_{p_0}^1, \delta_{p_0}^1)$ is constructed as follows. Intuitively, $\mathcal{M}_{p_0}^1$ simulates the thread p_0.
  - $Q_{p_0}^1 = ((Q \times Q_{p_0}) \cup Q_{new,p_0}) \times \{1\}$. The initial state $qs_{p_0}^1 = ((qs, qs_{p_0}), 1)$.
  - $\delta_{p_0}^1$ is constructed as follows. If (qp, qn) is a state transition of $\mathcal{P}_{p_0}$ then $((qp, 1), (qn, 1))$ is a state transition of $\mathcal{M}_{p_0}^1$. If $(qp, (qn, a))$ $(((qp, a), qn)$ respectively) is a stack push (stack pop respectively) transition of $\mathcal{P}_{p_0}$ then $((qp, 1), ((qn, 1), a))$ $((((qp, 1), a), (qn, 1))$ respectively) is a stack push (stack pop respectively) transition of $\mathcal{M}_{p_0}^1$. If $(qp, (qn, l_j))$ $((qp, l_j)$, $qn)$ respectively) is a lock acquisition (lock release respectively) transition of $\mathcal{P}_{p_0}$ then $((qp, 1), (qn, 1))$ is an increment (decrement respectively) transition of the jth counter.

    In addition there are some zero-test transitions and one additional state transition constructed as follows. These extra transitions are to ensure that just before the first context switch happens, the set of the locks owned by $\mathcal{P}_{p_0}$ is a subset of $\pi_{p_0}(ls_1)$. This is achieved as follows. Let $gs_1 = (q, q_1, \ldots, q_n)$. If $\pi_{p_0}(ls_1) = \mathsf{Lcks}$ then we add a state transition that takes $(q, q_{p_0}, 1)$ to $(q_{p_0}^\dagger, 1)$. Otherwise, let $\ell_1 < \cdots < \ell_m$ be the indices of the elements in $\mathsf{Lcks} \setminus \pi_{\ell_0}(ls_1)$. We add a zero-test of the counter ℓ_1 which takes the state $((q, q_{p_0}), 1)$ to the state $(test_{p_0,\ell_1}, 1)$. For each $1 \leq x < m$, we add a zero-test of the counter ℓ_{x+1} which takes the state $(test_{p_0,\ell_x}, 1)$ to $(test_{p_0,\ell_{x+1}}, 1)$. From the state $(test_{p_0,\ell_m}, 1)$ we add a state transition to $(q^{\dagger}_{p_0}, 1)$.

It is easy to see that the PDS $\mathcal{M}^1_{p_0}$ is a contextual PDS (since every thread of $\mathcal{CP}$ is contextual) and the state $(q^{\dagger}_{p_0}, 1)$ is reachable iff it is reachable with $\leq num_{locks}$ zero-test. Thus, after constructing $\mathcal{M}^1_{p_0}$, we check whether $(q^{\dagger}_{p_0}, 1)$ is reachable or not (thanks to Theorem 1). If it is not, the algorithm outputs "NO." If it is reachable then we can conclude that there is a computation compatible with sig_1. The next iteration begins.

The details of the other iterative steps are similar. The main difference is that in the iterative step t, the thread p_{t-1} cannot manipulate counters that correspond to the locks in the set $\cup_{i \neq t} held_i$. Furthermore, in the last iterative step, we check for reachability of $\boldsymbol{q}$. □

Hence, we can establish the main result of the paper.

Theorem 2. *Given $k \in \mathbb{N}$, the k-bounded context-switching reachability problem is decidable for n-PDS in which each thread exhibits contextual locking. For any fixed $k > 0$, the problem is at least as hard as the VASS configuration reachability problem.*

Proof. The decidability follows from Lemma 2. The VASS configuration reachability problem is as follows:

Given a n-counter system $\mathcal{M} = (Q, qs, \{\delta_{inc_i}, \delta_{dec_i}\}_{1 \leq i \leq n})$ with no zero-test transitions and a control state $q \in Q$, check if there is a computation starting with $(qs, \bar{0})$ that leads to $(q, \bar{0})$.

Now, given a n-counter system $\mathcal{M}$, we construct a 2-PDS $\mathcal{CP} = (\mathcal{P}_1, \mathcal{P}_2)$ that synchronizes only using reentrant locks as follows:

1. The set of locks, Lcks has $n + 1$ elements, $l_0, l_1, \ldots, l_n$.
2. $\mathcal{P}_1$ is non-recursive and simulates $\mathcal{M}$. The initial state of $\mathcal{P}_1$ is qs. For each $i > 0$, the value of the counter c_i is maintained by the number of times l_i is acquired by $\mathcal{P}_i$. The sum of the counters $c_1 + \cdots + c_n$ is maintained by the number of times l_0 is acquired. The simulation is achieved as follows. Whenever $\mathcal{M}$ makes an internal transition, so does $\mathcal{P}_1$. Whenever $\mathcal{M}$ increments (decrements respectively) counter i, $\mathcal{P}_1$ acquires (releases respectively) locks l_i and l_0.
3. $\mathcal{P}_2$ is also non-recursive and has two states $\{qs_2, qf_2\}$. qs_2 is the initial state of $\mathcal{P}_2$. There is only one transition of $\mathcal{P}_2$: $\mathcal{P}_2$ can acquire lock l_0 and transit to qf_2 from qs_2.

It is easy to see that there is a computation of $\mathcal{M}$ starting with $(qs, \bar{0})$ that leads to $(q, \bar{0})$ iff the state (q, qf_2) is reachable in $\mathcal{CP}$ by a computation with at most 1 context switch. □

Remark 1. Since the VASS configuration reachability problem is EXPSPACE-hard [3], Theorem 2 implies that the bounded context-switching reachability problem for n-PDS communicating via contextual reentrant locks is EXPSPACE-hard.

5 Conclusions

We have investigated the bounded context-switching problem for multi-threaded recursive programs synchronizing with contextual reentrant locks, showing it to be decidable. The decidability result is established by proving a novel result on pushdown counter systems: if the pushdown counter system is contextual then the problem of deciding whether a control state is reachable with a bounded number of zero tests is decidable. The result on pushdown counter systems is obtained by a reduction to the configuration reachability problem of VASS (Vector Addition System with States) and may be of independent interest. We also establish that the bounded context-switching reachability problem problem is at least as hard as the configuration reachability problem for VASS.

There are a few open problems. The status of the bounded context-switching reachability problem for the case when the locks are not contextual is open. This appears to be a very difficult problem. Our techniques imply that this problem is equivalent to the problem of checking configuration reachability in pushdown counter systems with a bounded number of zero-tests. The latter has been a longstanding open problem.

Another line of investigation is to explore other under-approximation techniques such as bounded phases [16]. It would also be useful to account for other synchronization primitives such as thread creation and barriers in addition to reentrant locks.

Practical aspects of our decision algorithm is left to future investigation. Since earlier static analysis techniques for analyzing programs with reentrant locks [11] mainly consider locks to be both nested and contextual, our techniques should be useful in analyzing a larger class of problems.

References

1. Atig, M.F., Ganty, P.: Approximating Petri net reachability along context-free traces. In: Foundations of Software Technology and Theoretical Computer Science. LIPIcs, vol. 13, pp. 152–163. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2011)
2. Bultan, T., Yu, F., Betin-Can, A.: Modular verification of synchronization with reentrant locks. In: MEMOCODE, pp. 59–68 (2010)
3. Cardoza, E., Lipton, R., Meyer, A.R.: Exponential space complete problems for Petri nets and commutative semigroups. In: Proceedings of the ACM Symposium on Theory of Computing, pp. 50–54 (1976)
4. Chadha, R., Madhusudan, P., Viswanathan, M.: Reachability under Contextual Locking. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 437–450. Springer, Heidelberg (2012)

5. Chadha, R., Viswanathan, M.: Decidability Results for Well-Structured Transition Systems with Auxiliary Storage. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 136–150. Springer, Heidelberg (2007)
6. Ganty, P., Majumdar, R.: Algorithmic verification of asynchronous programs. ACM Transactions on Programming Languages and Systems 34(1), 6 (2012)
7. Hague, M., Lin, A.W.: Model Checking Recursive Programs with Numeric Data Types. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 743–759. Springer, Heidelberg (2011)
8. Hague, M., Lin, A.W.: Synchronisation- and Reversal-Bounded Analysis of Multithreaded Programs with Counters. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 260–276. Springer, Heidelberg (2012)
9. Jhala, R., Majumdar, R.: Interprocedural analysis of asynchronous programs. In: Proceedings of the ACM Symposium on the Principles of Programming Languages, pp. 339–350 (2007)
10. Kahlon, V., Ivančić, F., Gupta, A.: Reasoning About Threads Communicating via Locks. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 505–518. Springer, Heidelberg (2005)
11. Kidd, N., Lal, A., Reps, T.W.: Language strength reduction. In: Static Analysis, pp. 283–298 (2008)
12. Rao Kosaraju, S.: Decidability of reachability in vector addition systems (preliminary version). In: Proceedings of the ACM Symposium on Theory of Computing, pp. 267–281 (1982)
13. Lal, A., Reps, T.W.: Reducing concurrent analysis under a context bound to sequential analysis. Formal Methods in System Design 35(1), 73–97 (2009)
14. Lammich, P., Müller-Olm, M., Wenner, A.: Predecessor Sets of Dynamic Pushdown Networks with Tree-Regular Constraints. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 525–539. Springer, Heidelberg (2009)
15. Leroux, J.: Vector addition system reachability problem: a short self-contained proof. In: Proceedings of the ACM Symposium on the Principles of Programming Languages, pp. 307–316. ACM (2011)
16. Madhusudan, P., Parlato, G.: The tree width of auxiliary storage. In: Proceedings of the ACM Symposium on the Principles of Programming Languages, pp. 283–294 (2011)
17. Mayr, E.W.: An algorithm for the general Petri net reachability problem. In: Proceedings of the ACM Symposium on Theory of Computing, pp. 238–246 (1981)
18. Qadeer, S., Rehof, J.: Context-Bounded Model Checking of Concurrent Software. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 93–107. Springer, Heidelberg (2005)
19. Reps, T.W., Horwitz, S., Sagiv, S.: Precise interprocedural dataflow analysis via graph reachability. In: Proceedings of the ACM Symposium on the Principles of Programming Languages, pp. 49–61 (1995)
20. Sen, K., Viswanathan, M.: Model Checking Multithreaded Programs with Asynchronous Atomic Methods. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 300–314. Springer, Heidelberg (2006)
21. La Torre, S., Madhusudan, P., Parlato, G.: The Language Theory of Bounded Context-Switching. In: López-Ortiz, A. (ed.) LATIN 2010. LNCS, vol. 6034, pp. 96–107. Springer, Heidelberg (2010)