Context-Bounded Verification of Thread Pools

PASCAL BAUMANN, Max Planck Institute for Software Systems (MPI-SWS), Germany
RUPAK MAJUMDAR, Max Planck Institute for Software Systems (MPI-SWS), Germany
RAMANATHAN S. THINNIYAM, Max Planck Institute for Software Systems (MPI-SWS), Germany
GEORG ZETZSCHE, Max Planck Institute for Software Systems (MPI-SWS), Germany

Thread pooling is a common programming idiom in which a fixed set of worker threads are maintained to execute tasks concurrently. The workers repeatedly pick tasks and execute them to completion. Each task is sequential, with possibly recursive code, and tasks communicate over shared memory. Executing a task can lead to more new tasks being spawned. We consider the safety verification problem for thread-pooled programs. We parameterize the problem with two parameters: the size of the thread pool as well as the number of context switches for each task. The size of the thread pool determines the number of workers running concurrently. The number of context switches determines how many times a worker can be swapped out while executing a single task—like many verification problems for multithreaded recursive programs, the context bounding is important for decidability.

We show that the safety verification problem for thread-pooled, context-bounded, Boolean programs is EXPSPACE-complete, even if the size of the thread pool and the context bound are given in binary. Our main result, the EXPSPACE upper bound, is derived using a sequence of new succinct encoding techniques of independent language-theoretic interest. In particular, we show a polynomial-time construction of downward closures of languages accepted by succinct pushdown automata as doubly succinct nondeterministic finite automata. While there are explicit doubly exponential lower bounds on the size of nondeterministic finite automata accepting the downward closure, our result shows these automata can be compressed. We show that thread pooling significantly reduces computational power: in contrast, if only the context bound is provided in binary, but there is no thread pooling, the safety verification problem becomes 3EXPSPACE-complete. Given the high complexity lower bounds of related problems involving binary parameters, the relatively low complexity of safety verification with thread-pooling comes as a surprise.

CCS Concepts: • **Theory of computation → Concurrency**; • **Software and its engineering → Software verification**.

Additional Key Words and Phrases: verification, safety, multithreaded programs, thread pool, context bounded, computational complexity

ACM Reference Format:
Pascal Baumann, Rupak Majumdar, Ramanathan S. Thinniyam, and Georg Zetzsche. 2022. Context-Bounded Verification of Thread Pools. *Proc. ACM Program. Lang.* 6, POPL, Article 17 (January 2022), 28 pages. https://doi.org/10.1145/3498678

Authors' addresses: Pascal Baumann, Max Planck Institute for Software Systems (MPI-SWS), Paul-Ehrlich-Straße, Building G26, Kaiserslautern, 67663, Germany, pbaumann@mpi-sws.org; Rupak Majumdar, Max Planck Institute for Software Systems (MPI-SWS), Paul-Ehrlich-Straße, Building G26, Kaiserslautern, 67663, Germany, rupak@mpi-sws.org; Ramanathan S. Thinniyam, Max Planck Institute for Software Systems (MPI-SWS), Paul-Ehrlich-Straße, Building G26, Kaiserslautern, 67663, Germany, thinniyam@mpi-sws.org; Georg Zetzsche, Max Planck Institute for Software Systems (MPI-SWS), Paul-Ehrlich-Straße, Building G26, Kaiserslautern, 67663, Germany, georg@mpi-sws.org.

2475-1421/2022/1-ART17
https://doi.org/10.1145/3498678

1 INTRODUCTION

Thread pooling is a common programming idiom in which a fixed set of worker threads are maintained to execute tasks concurrently. The worker threads repeatedly pick tasks from a task buffer and execute them to completion. Each task is sequential code, possibly recursive, and tasks communicate over shared memory. Executing a task can lead to more new tasks being spawned; these tasks get added to the task buffer for execution. When a worker thread finishes executing a task, it goes to the task buffer and picks another task non-deterministically. Thread pools reduce latency by executing tasks concurrently without paying the cost of creating and destroying new threads for each, possibly short-lived, task. Additionally, they improve system stability because resource requirements are more stable compared to unbounded creation and destruction of threads. Thus, most languages and runtimes for concurrency explicitly support thread pooling.

We consider the safety verification problem for thread-pooled programs modeled as reachability of some global state. Even if the global state is finite, the system is unbounded in multiple dimensions: each executing task can have an unbounded stack, and the task buffer of pending tasks can be unbounded. The safety verification problem is undecidable as soon as there are two workers in the thread pool and threads are recursive. Thus, we take the usual approach of bounding the number of context switches for each task [Qadeer and Rehof 2005]: the context bound is an upper bound on the number of times a worker thread can be interrupted by other threads while executing a specific task. The context-bounded safety verification problem for thread pools takes as input a program description and two parameters written in binary: the size of the thread pool N (i.e., the number of worker threads executing concurrently) and the context switch bound K. Our main result shows that the safety verification problem for this model can be decided in EXPSPACE, even when we assume the parameters are given in binary. In fact, our EXPSPACE upper bound holds for the model of *Boolean programs* [Ball and Rajamani 2000, 2001; Godefroid and Yannakakis 2013], where the global state of the program is specified succinctly in the form of Boolean variables and each thread is allowed its own set of local Boolean variables.

When there are no local variables, the global state is specified explicitly rather than succinctly, and the size of the thread pool is one, the model degenerates to the well-studied model of asynchronous programs [Ganty and Majumdar 2012; Jhala and Majumdar 2007; Sen and Viswanathan 2006]. Safety verification for asynchronous programs is already EXPSPACE-complete [Ganty and Majumdar 2012]. Thus, we get EXPSPACE-completeness for our problem—hardness holding already for a single executing thread in the thread pool. (We note that we consider a model in which Boolean parameters are only allowed to be passed to recursive calls within a thread, but not to newly spawned tasks. Allowing formal parameters to spawns as well would result in a 2EXPSPACE-complete problem: membership follows from our methods, whereas hardness can be shown by applying the techniques of Baumann et al. [2020] to our slightly different problem and setting.)

The EXPSPACE upper bound requires us to develop a number of new techniques, of independent interest, to deal with *succinct* representations of computations. Our goal is to reduce the decision problem to the coverability problem of a polynomial-sized vector addition system with states (VASS), which can be solved in EXPSPACE [Rackoff 1978]. Our starting point is the 2EXPSPACE algorithm for context-bounded reachability of Atig et al. [2009]. Their proof has the following steps. First, they observe that safety verification is preserved under "downward closures," where some spawned tasks are lost. Second, they show that the language of spawns and context switches of a single task execution can be represented as a pushdown automaton (PDA), and from this PDA, they can construct a non-deterministic finite automaton (NFA) for the downward closure of the language of the PDA that preserves the context switches and only loses spawns. Using the NFA for

each thread, they construct a VASS of polynomial size that counts the number of threads in each location of the NFA; safety verification in the original program reduces to coverability in this VASS.

A careful accounting of this algorithm gives a 3EXPSPACE algorithm for our problem. This is because the NFA representation for the downclosure of a PDA can be exponentially large [Courcelle 1991]; in fact, there are lower bounds showing that an exponential blow-up is necessary [Bachmeier et al. 2015]. Second, since the number of context switches is given in binary, the context switch preserving downclosure is in fact *doubly exponential* in the size of the PDA and K; again, there are examples showing that the blow-up is necessary. While the size N of the thread pool is given in binary, the VASS representation does not incur a further blow-up since we can precisely track the states of N threads with a mutiplicative cost. Additionally, specifying the global state succinctly via global Boolean variables increases the size of the downclosure in a way that is multiplicative to the size increase caused by a binary encoded K, therefore incurring no further blow-up either. Thus, a naive reduction to a VASS is doubly exponential, giving an overall triply exponential algorithm.

We overcome the complexity obstacles by developing algorithms for succinct machines. We introduce *succinct* representations of PDAs and NFAs. The states of a succinct PDA are encoded as strings over an alphabet, and the transitions are encoded as transducers. (Equivalently, one could represent them using Boolean circuits; our choice of transducers simplifies some language theoretic constructions.) A succinct PDA encodes an exponentially larger PDA. We also define *doubly succinct* NFAs, whose states are encoded as exponentially long strings, which represent NFAs that are doubly exponentially larger.

Our first technical result is that downclosures of succinctly represented PDAs are "well-compressible": they can be represented by *doubly succinct* NFA of polynomial size. The result strengthens a result of Majumdar et al. [2021] that shows a single exponential compression of the downclosure of normal PDAs. Our result is of independent interest: it implies, for example, that downclosures of Boolean programs are doubly succinctly representable. It is surprising, because the emptiness problem for succinct PDAs is EXPTIME-complete and one would expect an exponential blowup. Indeed, our construction carries out computations of alternating polynomial space Turing machines represented succinctly within the automaton. We further strengthen the result to show that even when we preserve the K context switches, the downclosure is still representable by a doubly succinct NFA.

Our next obstacle is to obtain the language of a thread pool containing N threads, each of which is given by a doubly succinct NFA. Here, an explicit representation of N separate states of the doubly succinct NFAs will lead to an exponential blowup. We introduce a succinct representation for VASS, where the control states are represented doubly succinctly—such a succinct VASS represents a VASS with doubly exponentially many control states. We show that the language of a thread pool can be represented as a VASS with a doubly succinct control.

Finally, we show that the language of a doubly succinct NFA can be seen as the coverability language of a (normal) VASS. The idea of the proof goes back to Lipton's encoding of doubly exponential counter machines succinctly using a VASS [Esparza 1998; Lipton 1976]; however, instead of proving a lower bound, as Lipton did, we use the succinct encoding to show a better upper bound. The overall construction implies that the coverability language of the doubly succinct VASS is the same as the coverability language of a normal VASS that can be constructed from the succinct representation. Thus, coverability of doubly succinct VASS can be decided in EXPSPACE.

Overall, a composition of these steps gives us the desired EXPSPACE upper bound for the thread-pooled reachability problem.

Our singly exponential space upper bounds are unexpected, since moving from unary to binary parameters usually involves an exponential jump in the complexity of verification. For example, in the special case where threads do not spawn new tasks and states are specified explicitly without

Boolean variables, Qadeer and Rehof [2005] showed that context-bounded reachability is NP-hard when the number of context switches is fixed or given in unary. We show that, even when the thread pool has just two threads ($N = 2$), the problem is NEXP-complete if the number of context switches is given in binary. (The upper bound follows from Qadeer and Rehof [2005], even when Boolean variables are present, and the lower bound can be shown by reducing from the following NEXP-complete problem: given context free grammars G_1 and G_2 and a number k in binary, are there words $w_1 \in \mathsf{L}(G_1)$ and $w_2 \in \mathsf{L}(G_2)$ that agree on the first k letters?)

Similarly, Atig et al. [2009] study the context-bounded safety verification problem for multi-threaded shared memory programs with neither thread pooling nor Boolean variables, that is, in a model where the global state is specified explicitly and each task is executed on a separately created thread. They show that the problem is in 2EXPSPACE for a fixed context bound, or if the context bound is given in unary. Baumann et al. [2020] show a matching lower bound for this case. We improve upon these results to show that safety verification (without thread pooling) is 3EXPSPACE-complete when the context bound is given in binary. The hard part is the lower bound. We extend the encoding of Baumann et al. [2020] to perform an "exponentially larger" computation when K is in binary. Interestingly, succinct computations are now used in this proof to show *lower bounds* and hardness!

Given the above lower bounds, it is indeed surprising that fixing the thread pool parameter in binary leads to a doubly exponential reduction in complexity of safety verification (or, seen through a lens of complexity theory, in the reduction in expressiveness of languages computable by these machines). Indeed, the complexity is no higher than the special case of one thread that executes tasks to completion!

Related Work. Programming with thread pools is a ubiquitous concurrency pattern, and almost every language or library supporting concurrent programming supports thread pooling. Usually, the thread pool needs to be configured by programmers; there are real-world instances that show that such configurations may be tricky to get right.[1] Since thread pools appear in many large-scale systems, dynamic analysis tools often provide support for thread pools [Li et al. 2018]. However, the complexity implications of thread pools on static verification had not been considered.

There are, by now, many decidability results for context bounded verification in multi-threaded settings [Atig et al. 2009; Baumann et al. 2021; La Torre et al. 2009, 2010; Lal and Reps 2009; Meyer et al. 2018; Musuvathi and Qadeer 2007; Qadeer and Rehof 2005]. The work of Atig et al. [2009] is closest to ours in the programming model: they consider safety verification for a multithreaded shared memory model with *dynamic thread spawns*, the same as us. Our model additionally has global and local variables and we consider the effect of thread pools. Prior results on context bounded reachability focused on a fixed number of threads [Chini et al. 2017; Lal and Reps 2009; Qadeer and Rehof 2005]; interestingly, in most of these papers, the context bound parameter was assumed to be given in unary, and as we stated before, the complexity results are "one exponential higher" when the parameter is assumed to be binary. Chini et al. [2017] carry out a multi-parameter analysis of bounded context switching for a fixed number of threads. Their main results connect the complexity of this problem to hypotheses in fine-grained complexity. However, they do not consider binary encodings of the context switch bound.

[1] See, e.g., discussions at https://engineering.zalando.com/posts/2019/04/how-to-set-an-ideal-thread-pool-size.html, https://developer.android.com/guide/background/threading, https://developer.android.com/topic/performance/threads, and especially https://www.techyourchance.com/threadposter-explicit-unit-testable-multi-threading-library-for-android/ for discussions of the complexity of thread pool configuration and resulting crashes in the Android settings application.

```
1  global lock l;
2  main() { spawn handler(); spawn main(); }
3  handler() {
4    if * { a(0); }
5    else { a(1); }
6    unlock(l);
7  }
8  a(bool i) {
9    if * { a(0); }
10   else if * { a(1); }
11   else { lock(l); }
12   write(i); // critical section
13   return i;
14 }
```

Fig. 1. An example concurrent program

For a class of graph algorithms, one can prove a metatheorem that shows that succinct encodings give an exponential blowup: NP becomes NEXP, and so on [Papadimitriou and Yannakakis 1986]. Our EXPSPACE result shows that the space of safety verification problems is more nuanced.

The core of our results provide new and efficient constructions on *succinct* machines. Specifically, we show that the downward closure of the language of a succinct pushdown automaton has a small representation as a doubly succinct finite automaton. It is well known that the downward closure of a context free language is effectively computable [Courcelle 1991; van Leeuwen 1978]. Bachmeier et al. [2015], following results by Gruber et al. [2009], show exponential lower bounds on the state complexity of an NFA representing the downward closure. Later, Majumdar et al. [2021] showed that the NFA is "compressible", meaning that there is a polynomial succinctly represented NFA for the downward closure. Our results improve the succinct representability for succinctly defined PDAs: there is a polynomial (doubly succinct) NFA representation, even though there is a tight doubly exponential lower bound for an NFA representation. Further, we show that the same result holds for a stronger variant of the downward closure, where a subset of the alphabet is preserved.

While the use of thread pools is usually motivated by performance concerns, the fact that it might lead to a verification problem of lower complexity despite binary encodings, and indeed the *same* complexity as the special case of one thread and execute-to-completion, came as a surprise.

2 A MODEL OF THREAD POOLING

2.1 Dynamic Networks of Concurrent Boolean Programs with Recursion (DCBP)

Figure 1 shows a simple example of the different features of the programs that we want to verify. In the example, the main program *spawns* new tasks when it is executed: an instance of a handler task and a further instance of itself. A task executes sequential code, with *(recursive) function calls*, *parameters*, and *local variables*. They can also read and write *shared global variables* (e.g., the lock l). In the example, a handler iteratively stores the value of some outside condition (modeled as non-determinism) in the parameter to a recursive function call. When the recursion ends (non-deterministically), the handler acquires the global lock l to write out the stored values (in-reverse, due to how recursion works) and then returns the lock.

Note that the program can spawn an unbounded number of handlers, that can all execute in parallel. We shall consider a *thread-pooled* execution, where a fixed number of worker threads repeatedly pick and execute the tasks. The threads run in an interleaved fashion and can be swapped in and out. However, each task is executed to completion before a new spawned task is picked.

A possible safety property for the program is to ensure mutual exclusion for the write operation on line 12. The property does hold for this program, but proving this is difficult as the state space is unbounded in several directions: the unbounded number of spawned tasks and the unbounded stack of an executing task.

To model programs like the one above, we consider an abstract, language-theoretic model of thread-pooled shared memory programs, following [Atig et al. 2009] and subsequent work.

The model involves tasks, which can be spawned dynamically during execution, and a stack per thread to capture recursion. We furthermore augment this model with global and thread-local Boolean variables, which is a new addition when compared to prior work. The lock from the example program can be easily modeled as a global Boolean variable, and the stack and thread-local variables allow us to simulate features like Boolean function parameters. We will go into more detail about how this model captures program behavior at the end of this subsection.

Towards Syntax: Defining transducers. To introduce the syntax of our model, we first need to define the notion of transducers. For $k \in \mathbb{N}$, a *(length preserving) k-**ary transducer*** $T = (Q, \Delta, q_0, Q_f, E)$ consists of a finite set of *states* Q, an alphabet Δ, an *initial state* $q_0 \in Q$, a set of *final states* $Q_f \subseteq Q$, and a *transition relation* $E \subseteq Q \times (\Delta^k \cup \{\varepsilon\}^k) \times Q$. For a *transition* $(q, a_1, \ldots, a_k, q') \in E$, we write $q \xrightarrow{(a_1,\ldots,a_k)} q'$. The *size* of T is defined as $|T| = k \cdot |E|$.

The *language* of T is the k-ary relation $L(T) \subseteq (\Delta^*)^k$ containing precisely those k-tuples $(w_1, \ldots, w_k)$, for which there is a transition sequence $q_0 \xrightarrow{(a_{1,1},\ldots,a_{k,1})} q_1 \xrightarrow{(a_{1,2},\ldots,a_{k,2})} \ldots \xrightarrow{(a_{1,m},\ldots,a_{k,m})} q_m$ with $q_m \in Q_f$ and $w_i = a_{i,1}a_{i,2}\cdots a_{i,m}$ for all $i \in \{1, \ldots, k\}$. Such a transition sequence is called an *accepting run* of T.

We note that in the more general (i.e., non-length-preserving) definition of a transducer (see, e.g., [Ginsburg 1966]), the transition relation E is a subset of $Q \times (\Delta_\varepsilon)^k \times Q$, where $\Delta_\varepsilon = \Delta \cup \{\varepsilon\}$. All transducers we consider in this paper are length-preserving.

Syntax. A *Dynamic Network of Concurrent Boolean Programs with Recursion* (DCBP) $\mathcal{D} = (V_{\mathrm{gl}}, V_{\mathrm{loc}}, \Gamma, \mathcal{T}_c, \mathcal{T}_s, \mathcal{T}_r, \mathcal{T}_t, \vec{a}_0, \gamma_0)$ consists of a finite set of *global Boolean variables* $V_{\mathrm{gl}} = \{v_1, v_2, \ldots, v_m\}$, a finite set of *local Boolean variables* $V_{\mathrm{loc}} = \{v'_1, v'_2, \ldots, v'_n\}$, a finite alphabet of *stack symbols* Γ, an *initial assignment* $\vec{a}_0 \in \{0,1\}^{V_{\mathrm{gl}}}$ of the global (Boolean) variables, an *initial stack symbol* $\gamma_0 \in \Gamma$, and four sets of transducers $\mathcal{T}_c, \mathcal{T}_s, \mathcal{T}_r, \mathcal{T}_t$ which succinctly describe the allowed transitions in the program, described below.

Let $\bar{\Gamma} = \{\bar{\gamma} \mid \gamma \in \Gamma\}$. The four sets of transducers have the following form:

(1) For each tuple $a = (\gamma, v)$ where $\gamma \in \Gamma \cup \{\varepsilon\}$, $v \in \Gamma \cup \bar{\Gamma} \cup \{\varepsilon\}$, there exists a 2-ary transducer $T_a \in \mathcal{T}_c$ which works over the alphabet $\Delta = \{0,1\}$ and accepts strings from $(\Delta \times \Delta)^{m+n}$.
(2) For each $\gamma \in \Gamma \cup \{\varepsilon\}$, there exists a 2-ary transducer $T_\gamma \in \mathcal{T}_s$ which works over the alphabet $\Delta = \{0,1\}$ and accepts strings from $(\Delta \times \Delta)^{m+n}$.
(3) For each $\gamma \in \Gamma$, there exists a 2-ary transducer $T_\gamma \in \mathcal{T}_r$ which works over the alphabet $\Delta = \{0,1\}$ and accepts strings from $(\Delta \times \Delta)^m$.
(4) $\mathcal{T}_t$ contains one 2-ary transducer T_t which works over the alphabet $\Delta = \{0,1\}$ and accepts strings from $(\Delta \times \Delta)^m$.

The *size* $|\mathcal{D}|$ of $\mathcal{D}$ is defined as $m + n + |\Gamma| + \sum_{T \in (\mathcal{T}_c \cup \mathcal{T}_s \cup \mathcal{T}_r \cup \mathcal{T}_t)} |T|$: the number of global variables, the number of local variables, the stack alphabet, and the sizes of the transducers which define the valid transitions.

Intuition. A DCBP represents a multi-threaded program with a thread pool of fixed size and a shared global memory represented by the set V_{gl} of global Boolean variables. Computation is carried out by tasks. The tasks can read and write global variables; in addition, every task has its own copy of the local variables V_{loc}. Tasks make potentially recursive function calls, and we use the stack alphabet Γ to maintain their stacks. Each running task can manipulate the global variables, its local variables, and its stack. It can also spawn new tasks into a task buffer.

Since the set of possible assignments to the global and local variables is exponentially large, the transitions between two different configurations of a thread, which potentially involve a change in the assignments of these variables, are given succinctly using transducers.

Execution of tasks is coordinated by a thread pool, which is an a priori fixed number of concurrent threads that are used to execute the tasks. A nondeterministic scheduler schedules the threads in the thread pool. When some thread in the thread pool is idle, it can pick one of the pending tasks from the task buffer and start executing it. The task is executed concurrently with other threads in the thread pool, until completion. On completion, the executing thread will nondeterministically pick another task from the task buffer.

Intuitively, the transducers represent "transition relations" relating the assignments to global and local variables of a program when it takes a step. Each set of transducers $\mathcal{T}_c, \mathcal{T}_s, \mathcal{T}_r, \mathcal{T}_t$ corresponds to different types of transitions applicable to the DCBP. The transitions are divided into two major kinds: *creation* transitions applicable to the running of a single thread given by $\mathcal{T}_c$ and *swap*, *resumption*, and *termination* transitions corresponding to the actions of the scheduler, given by $\mathcal{T}_s$, $\mathcal{T}_r$ and $\mathcal{T}_t$ respectively. A transducer in $\mathcal{T}_c$ describes the updates to the global and local variables during a single step of a thread. A transducer in $\mathcal{T}_s$ describes the updates when a thread is swapped out. Finally, transducers in $\mathcal{T}_r$ and $\mathcal{T}_t$ describe how the global variables are updated when a thread is resumed and on thread termination, respectively. We explain these in detail below.

Towards Semantics: Preliminary Definitions. In order to provide the semantics of DCBP, we need a few definitions and notation. A *multiset* $\mathbf{m} : S \to \mathbb{N}$ over a set S maps each element of S to a natural number. Let $\mathbb{M}[S]$ be the set of all multisets over S. We treat sets as a special case of multisets where each element is mapped onto 0 or 1. We write $\mathbf{m} = [\![a_1, a_1, a_3]\!]$ for the multiset $\mathbf{m} \in \mathbb{M}[S]$ such that $\mathbf{m}(a_1) = 2$, $\mathbf{m}(a_3) = 1$, and $\mathbf{m}(a) = 0$ for each $a \in S \backslash \{a_1, a_3\}$. The empty multiset is denoted $\emptyset$. The *size* of $\mathbf{m} \in \mathbb{M}[S]$, denoted $|\mathbf{m}|$, is given by $\sum_{a \in S} \mathbf{m}(a)$. This definition applies to sets as well.

Given two multisets $\mathbf{m}, \mathbf{m}' \in \mathbb{M}[S]$ we define $\mathbf{m} + \mathbf{m}' \in \mathbb{M}[S]$ to be a multiset such that for all $a \in S$, we have $(\mathbf{m} + \mathbf{m}')(a) = \mathbf{m}(a) + \mathbf{m}'(a)$. For $c \in \mathbb{N}$, we define $c\mathbf{m}$ as the multiset that maps each $a \in S$ to $c \cdot \mathbf{m}(a)$. We also define the natural order $\preceq$ on $\mathbb{M}[S]$ as follows: $\mathbf{m} \preceq \mathbf{m}'$ iff there exists $\mathbf{m}^E \in \mathbb{M}[S]$ such that $\mathbf{m} + \mathbf{m}^E = \mathbf{m}'$. We also define $\mathbf{m} - \mathbf{m}'$ for $\mathbf{m}' \preceq \mathbf{m}$ analogously: for all $a \in S$, we have $(\mathbf{m} - \mathbf{m}')(a) = \mathbf{m}(a) - \mathbf{m}'(a)$.

Semantics. The set of configurations of $\mathcal{D}$ is

$$\underbrace{\{0,1\}^{V_{\text{gl}}}}_{\text{global state}} \times \underbrace{\left(\left(\{0,1\}^{V_{\text{loc}}} \times \Gamma^* \times \mathbb{N}\right) \cup \{\#\}\right)}_{\text{local config or schedule pt}} \times \underbrace{\mathbb{M}[\{0,1\}^{V_{\text{loc}}} \times \Gamma^* \times \mathbb{N}]}_{\text{thread pool}} \times \underbrace{\mathbb{M}[\Gamma]}_{\text{task buffer}}$$

A *configuration* of a DCBP consists of (a) an assignment $\vec{a}$ to the global variables, (b) a *local configuration* of the active thread (or a special symbol "#" signifying a schedule point), (c) a *thread pool*, which is a multiset of local configurations of inactive, partially executed threads, and (d) a *task buffer* of pending (that is, unstarted) tasks. The local configuration of a thread is a tuple $(\vec{b}, w, i)$ where $\vec{b}$ is the assignment of the local variables, w is the stack content, and i is a context switch number. The context switch number tracks how many times an executing task has already been context switched by the underlying scheduler.

In order for us to define state transitions succinctly using transducers, we need to impose an order on the set of global and local variables in order to form a well-defined string from $\{0,1\}^*$ given the variable values. Hence for a set of variables $V = \{v_1, \ldots, v_m\}$ and an assignment $\vec{a} \in \{0,1\}^V$ we identify $\vec{a}$ with the string $\vec{a}(v_1) \cdots \vec{a}(v_m)$. We shall write $\langle \vec{a}, (\vec{b}, w, i), \mathbf{m}, \mathbf{t} \rangle$ or $\langle \vec{a}, \#, \mathbf{m}, \mathbf{t} \rangle$ for configurations. We shall assume that the thread pool $\mathbf{m}$ is bounded by a number $N > 0$: this represents an a priori fixed number of threads used for executing tasks. We shall also consider the special case of $N = \infty$, that corresponds to no thread pooling and in which every pending task can start executing immediately.

The initial configuration of $\mathcal{D}$ is $\langle \vec{a}_0, \#, [[(\vec{\mathbf{0}}, \gamma_0, 0)]], \emptyset \rangle$. For a configuration c of $\mathcal{D}$, we will sometimes write $c.\vec{a}$ for the state of c and $c.\mathbf{m}$ for the multiset of threads of c (both active and inactive). The *size* of a configuration $c = \langle \vec{a}, (\vec{b}, w, i), \mathbf{m}, \mathbf{t} \rangle$ is defined as

$$|c| = m + n + |w| + \sum_{(\vec{b}', w', j) \in \mathbf{m}} (n + |w'|) + \sum_{\gamma \in \Gamma} \mathbf{t}(\gamma).$$

First let us explain the transitions corresponding to the action of a single thread. The steps of a single executing task define the following *thread step* relation $\rightarrow$ on configurations of $\mathcal{D}$: we have $\langle \vec{a}, (\vec{b}, w, i), \mathbf{m}, \mathbf{t} \rangle \rightarrow \langle \vec{a}', (\vec{b}', w', i), \mathbf{m}', \mathbf{t}' \rangle$ for all $w \in \Gamma^*$ iff

(1) $(\vec{a}\vec{b}, \vec{a}'\vec{b}') \in \mathsf{L}(T_{\varepsilon,v})$ where $T_{\varepsilon,v} \in \mathcal{T}_c$, $\mathbf{m}' = \mathbf{m}$, $\mathbf{t}' = \mathbf{t}$ and one of the following conditions hold:

i $v \in \Gamma \cup \{\varepsilon\}$ and $w' = vw$, or

ii $v = \bar{\gamma} \in \bar{\Gamma}$ and $\gamma w' = w$.

or (2) $(\vec{a}\vec{b}, \vec{a}'\vec{b}') \in \mathsf{L}(T_{\gamma',v})$ where $T_{\gamma',v} \in \mathcal{T}_c$ and $\mathbf{m}' = \mathbf{m}$ and $\mathbf{t}' = \mathbf{t} + [[\gamma']]$. Additionally, one of the stack conditions i or ii from (1) above hold.

We extend the *thread step* relation $\rightarrow^+$ to be the irreflexive-transitive closure of $\rightarrow$; thus $c \rightarrow^+ c'$ if there is a sequence $c \rightarrow c_1 \rightarrow \ldots c_k \rightarrow c'$ for some $k \geq 0$.

Secondly, we have the actions of the non-deterministic scheduler which switches between concurrent threads. The active thread is the one currently being executed and the multiset $\mathbf{m}$ keeps partially executed tasks in the thread pool; the size of $\mathbf{m}$ is bounded by the size N of the thread pool. All other pending tasks that have not been picked for execution yet remain in the task buffer $\mathbf{t}$. The scheduler may interrupt a thread based on the interruption transitions, non-deterministically resume a thread based on the resumption transitions, and terminate a thread based on the termination transitions. Picking a new task to execute is handled independently of the transducers and does not change any variable assignments.

The actions of the scheduler define the *scheduler step* relation $\mapsto$ on configurations of $\mathcal{D}$:

$$\textsc{Swap}\ \frac{(\vec{a}\vec{b}, \vec{a}'\vec{b}') \in \mathsf{L}(T_\gamma) \text{ where } T_\gamma \in \mathcal{T}_s}{\langle \vec{a}, (\vec{b}, w, i), \mathbf{m}, \mathbf{t} \rangle \mapsto \langle \vec{a}', \#, \mathbf{m} + [[\vec{b}', \gamma w, i+1]], \mathbf{t} \rangle} \qquad \textsc{Resume}\ \frac{(\vec{a}, \vec{a}') \in \mathsf{L}(T_\gamma) \text{ where } T_\gamma \in \mathcal{T}_r}{\langle \vec{a}, \#, \mathbf{m} + [[\vec{b}, \gamma w, i]], \mathbf{t} \rangle \mapsto \langle \vec{a}', (\vec{b}, \gamma w, i), \mathbf{m}, \mathbf{t} \rangle}$$

$$\textsc{Terminate}\ \frac{(\vec{a}, \vec{a}') \in \mathsf{L}(T_t) \text{ where } T_t \in \mathcal{T}_t}{\langle \vec{a}, (\vec{b}, \varepsilon, i), \mathbf{m}, \mathbf{t} \rangle \mapsto \langle \vec{a}', \#, \mathbf{m}, \mathbf{t} \rangle} \qquad \textsc{Pick}\ \frac{|\mathbf{m}| < N}{\langle \vec{a}, \#, \mathbf{m}, \mathbf{t} + [[\gamma]] \rangle \mapsto \langle \vec{a}, \#, \mathbf{m} + [[\vec{\mathbf{0}}, \gamma, 0]], \mathbf{t} \rangle}$$

If a thread can be interrupted, then Swap swaps it out and increases the context switch number of the executing task. The global and local variables can both be changed and the corresponding transducer accepts words of length $m + n$. In the Resume and Terminate rules, only the global variables are modified and so the transducers accept words of length m. The rule Resume picks a thread that is ready to run based on the current assignment of global variables and its top of stack symbol and makes it active. The rule Terminate removes a task on termination (empty stack), freeing up the thread that was executing it. The rule Pick picks a pending task for execution, if there is space in the thread pool. Note that if $N = \infty$, the rule is always enabled.

A *run* of a DCBP is a finite or infinite sequence of alternating thread execution and scheduler step relations

$$c_0 \rightarrow^+ c_0' \mapsto^+ c_1 \rightarrow^+ c_1' \mapsto^+ \ldots$$

such that c_0 is the initial configuration. The run is *N-thread-pooled* if the thread pool is bounded by N. The run is *K-context switch bounded* if, moreover, for each $j \geq 0$, the configuration $c_j = (\vec{a}, (\vec{b}, w, i), \mathbf{m}, \mathbf{t})$ satisfies $i \leq K$. In a K-context switch bounded run, each thread is context switched

at most K times and the scheduler never schedules a thread that has already been context switched $K + 1$ times. When the distinction between thread and scheduler steps is not important, we write a run as a sequence $c_0 \Rightarrow c_1 \ldots$.

Modelling programs. To model programs like the example at the start of this section, there are two main challenges that we need to overcome. Firstly, DCBP define possible changes to the variables via transducers, while programs instead use simple conditional statements and assignments. Secondly, DCBP have thread-local variables while programs usually have function-local ones.

Let us begin by arguing that for conditionals and assignments, it suffices to simulate `if`-statements over only single variables and assignments involving only the constant values 0 and 1. Regarding the latter, note that we can rewrite an assignment involving an arbitrary Boolean expression φ to two constant assignments by using an `if-else`-block over φ. Furthermore, `else`-statements can be easily rewritten to `if`-statements over a conjunction of the negations of all preceding Boolean expressions in the block. Finally, to rewrite an `if`-statement over φ, we can assume that φ is in negation normal form. Then $\varphi_1 \wedge \varphi_2$ can be rewritten as two nested `if`-statements over the two respective parts, while $\varphi_1 \vee \varphi_2$ can be rewritten as two non-nested `if`-blocks right after one another. We do not need to rewrite $\neg\varphi$, as in this case φ will be a single variable. To not duplicate the code inside of an `if`-block in the $\vee$-case we can make use of `goto`-statements, which our transducers can also simulate. This way we get at most a polynomial blow-up.

We continue by explaining how a transducer simulates the resulting `if`-statements, assignments, and `goto`-statements. Firstly, we extend the DCBP by finitely many additional local Boolean variables, to store the current line number of any given thread. Then an accepting run ρ in the transducer for a particular statement in the program takes the binary representation of the corresponding line number as input, and outputs either the line number directly below the statement, or the target line number in case of a `goto`. For an `if`-statement on variable v (respectively $\neg v$) the input on the remainder of ρ matches the output, while the transducer checks that there is a 1 (respectively 0) at the position corresponding to v. For an assignment of 1 (respectively 0) to v, the transducer instead outputs 1 (respectively 0) at the position corresponding to v, while keeping the rest of the input unchanged.

Now we also need to argue that we can simulate function-local variables by thread-local ones. The main trick is to push all values of local variables to the stack before pushing a function call, upon which we reset all values of local variables. When we return, we save the return value in a special variable followed by popping the function call from the stack, upon which we pop all variable values from the stack and store each one in its corresponding variable.

More formally, V_{gl} would contain the names of all global Boolean variables, and V_{loc} would contain all local variable names in all functions, all line numbers in the code, and one special variable `return`. The stack alphabet Γ would contain all function names and two copies of each function-local variable in the program, one for each truth value. New tasks would simply be spawned with one function name on the stack. Note that new tasks are always initialised with all local variables set to 0. Each task's current line number would be stored by setting the corresponding variable to true and all other such variables to false. The transducers would be defined in a way to facilitate all the program operations.

2.2 Decision Problems and Main Results

The *reachability problem* for DCBP asks, given an assignment $\vec{a}$ to the global variables of $\mathcal{D}$ (henceforth also called a *global state*), if there is a run $c_0 \Rightarrow c_1 \ldots \Rightarrow c_\ell$ such that $c_\ell.\vec{a} = \vec{a}$. Reachability of multi-threaded recursive programs is undecidable already with two fixed threads (see Section 3.2 in [Ramalingam 2000]).

We consider *context switch bounded* decision questions. Given $K \in \mathbb{N}$, a global state $\vec{a}$ of $\mathcal{D}$ is K-context switch bounded reachable if there is a K-context switch bounded run $c_0 \Rightarrow \ldots \Rightarrow c_\ell$ with $c_\ell.\vec{a} = \vec{a}$.

The *thread-pooled context-bounded reachability problem* is the following:

Given A DCBP $\mathcal{D}$, a global state $\vec{a}$, and two numbers K and N in *binary*;

Question Is $\vec{a}$ reachable in $\mathcal{D}$ in an N-thread-pooled, K-context switch bounded execution?

We show the following result.

Theorem 2.1 (Safety Verification with Thread-Pooling). *The thread-pooled context-bounded reachability problem is* EXPSPACE*-complete.*

EXPSPACE-hardness holds already for a DCBP $\mathcal{D}$, $K = 0$, and $N = 1$, even when the global state is restricted to be an explicitly given finite set (rather than as valuations to variables) and there are no additional local variables. This follows from results on *asynchronous programs* [Ganty and Majumdar 2012], which can be seen as the special case of a simpler model where the global states are specified as a finite set of possible values, and there is a single thread, which executes each task to completion before picking the next task. In other words, the case with non-succinctly defined states, a thread pool of size $N = 1$, and no context-switching (i.e. $K = 0$). Thus, our main aim is to show the upper bound.

In case $N = \infty$, there is no thread pooling and an arbitrary number of tasks can execute simultaneously. Then, the general *context-bounded reachability problem* takes as input a DCBP, a global state, and a context bound K in binary, and asks if the global state is reachable in a K-context switch bounded run, assuming there is no bound on the thread pool. If furthermore the global states are defined explicitly and there are no local variables, then this results in a setting, which has been studied before: Atig et al. [2009] showed that the context-bounded reachability problem is in 2EXPSPACE for a fixed K (or when K is given in unary). A matching lower bound for all fixed $K \geq 1$ (or equivalently, for K given in unary) was provided by Baumann et al. [2020]. We study the problem when K is given in binary and states are defined in a succinct manner via global and local Boolean variables. We show the following result.

Theorem 2.2 (Safety Verification without Thread Pooling). *The context-bounded reachability problem is* 3EXPSPACE*-complete.*

The 3EXPSPACE upper bound follows from a relatively simple analysis of the upper bound of Atig et al. [2009]. For the lower bound, we extend the 2EXPSPACE-hardness result of Baumann et al. [2020] with new encodings.

2.3 Outline of the Rest

As discussed in the Introduction, our EXPSPACE upper bound has three parts, following the general outline of the proof of decidability for context-bounded reachability [Atig et al. 2009].

The **first step** is to distill a succinct pushdown automaton for each task, and to construct a finite-state automaton that represents the downward closure of its language. In Section 3, we prove general results about succinct pushdown automata and doubly succinct representations of their downward closures. We then apply these results to DCBP in Section 4 to obtain succinct representations of the downward closures of tasks, where the number of context switches is preserved.

The **second step** is to construct a vector addition system with states (VASS), a model equivalent to Petri nets, that represents the state of a DCBP. Again, in order to overcome the exponential blow-up of representing all N threads in the thread pool (remember, N is in binary), we first introduce a succinct version of VASS and show the coverability problem for the succinct version can be solved in EXPSPACE (Section 5).

In the **third step**, we show a reduction from DCBP to succinct VASS, where we use the doubly succinct representation for the downward closures of all task languages.

Together, we obtain an EXPSPACE upper bound, as promised in Theorem 2.1.

Finally, in Section 6, we sketch the 3EXPSPACE lower bound in case the thread pool is unrestricted, and also give a very short argument as to why the matching upper bound follows from the results of Atig et al. [2009].

Remark 1. *We note that our model of* DCBP *assumes that tasks are not spawned with parameters that initialize the local variables; each local variable is always initialized to* 0 *instead. This in particular means that while we can model passing Boolean parameters as part of recursive function calls, we cannot model passing such parameters as part of spawning new tasks.*

If we allowed parameters in task creation, the thread-pooled context-bounded reachability problem would become 2EXPSPACE*-complete. For the membership, observe that there would be now exponentially many types of tasks in the buffer, one for each stack symbol γ and each assignment $\vec{b}$ to the local variables. Since our* EXPSPACE *algorithm for this problem relies on a polynomially-sized task buffer, we get a blow-up when using the same methods. For the hardness, one can apply a slightly altered version of the result from [Baumann et al. 2020, Remark 6]. While the unaltered version of this result does require a very large number of threads running at the same time, note that this is only necessary because each task needs to start its execution to receive its initial assignment to the local variables. With passing of parameters in task creation, initial assignments can be set for unstarted tasks as well, eliminating the need for a large thread pool.*

3 SUCCINCT PUSHDOWN AUTOMATA AND SUCCINCT DOWNWARD CLOSURES

Some of the detailed constructions for Sections 3-5 are quite technical, and therefore provided in the full version.

3.1 Preliminaries: Pushdown Automata

For an alphabet Γ, we write $\bar{\Gamma} = \{\bar{\gamma} \mid \gamma \in \Gamma\}$. Moreover, if $x = \bar{\gamma}$, then we define $\bar{x} = \gamma$. A *pushdown automaton* (PDA) $\mathcal{P} = (Q, \Sigma, \Gamma, E, q_0, \gamma_0, Q_F)$ consists of a finite set of *states* Q, a finite input alphabet Σ, a finite alphabet of *stack symbols* Γ, an *initial state* $q_0 \in Q$, an *initial stack symbol* $\gamma_0 \in \Gamma$, a set of *final states* $Q_F \subseteq Q$, and a transition relation $E \subseteq (Q \times \Sigma_\varepsilon \times \hat{\Gamma} \times Q)$, where $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$ and $\hat{\Gamma} = \Gamma \cup \overline{\Gamma} \cup \{\varepsilon\}$. For $(q, a, v, q') \in E$ we also write $q \xrightarrow{a|v} q'$.

The set of *configurations* of $\mathcal{P}$ is $Q \times \Gamma^*$. For a configuration (q, w) we call q its *state* and w its *stack content*. The *initial configuration* is (q_0, γ_0). The set of *final configurations* is $Q_F \times \Gamma^*$. For configurations (q, w) and (q', w'), we write $(q, w) \overset{a}{\Rightarrow} (q', w')$ if there is an edge (q, a, v, q') in E such that (i) if $v = \varepsilon$, then $w' = w$, (ii) if $v \in \Gamma$, then $w' = wv$, and (iii) if $v = \bar{\gamma}$ for $\gamma \in \Gamma$, then $w = w'\gamma$.

Informally, an edge $(q, a, v, q') \in E$ with $v \in \Gamma$ denotes the "push" of v onto the stack, and, for every $\gamma \in \Gamma$, the letter $\bar{\gamma}$ denotes the "pop" of γ from the stack.

For two configurations c, c' of $\mathcal{P}$, we write $c \Rightarrow c'$ if $c \overset{a}{\Rightarrow} c'$ for some a. Furthermore, we write $c \overset{u}{\Rightarrow}{}^* c'$ for some $u \in \Sigma^*$ if there is a sequence of configurations c_0 to c_n with

$$c = c_0 \overset{a_1}{\Longrightarrow} c_1 \overset{a_1}{\Longrightarrow} c_2 \cdots c_{n-1} \overset{a_n}{\Longrightarrow} c_n = c',$$

such that $a_1 \dots a_n = u$. We then call this sequence a *run* of $\mathcal{P}$ over u. We also write $c \Rightarrow^* c'$ if the word u does not matter. If w_i is the stack content of c_i, then the *stack height* of the run is defined as $\max\{|w_i| \mid 0 \le i \le n\}$. If the run has stack height at most h then we write $c \overset{u}{\Rightarrow}_h c'$. A run of $\mathcal{P}$ is

accepting if c is initial and c' is final. Given two configurations c, c' of $\mathcal{P}$ with $c \Rightarrow^* c'$, we say that c' is *reachable* from c and that c is *backwards-reachable* from c'. If c is the initial configuration, we simply say that c' is reachable.

The *language* accepted by $\mathcal{P}$, denoted $L(\mathcal{P})$ is the set of words in Σ^*, for which $\mathcal{P}$ has an accepting run. We also define a language accepted with bounded stack height. For $h \in \mathbb{N}$, we define

$$\mathsf{L}_h(\mathcal{P}) = \{u \in \Sigma^* \mid (q_0, \varepsilon) \overset{u}{\Rightarrow}_h (q_f, w) \text{ where } q_f \in Q_F\}.$$

Note that the *bounded stack language* $\mathsf{L}_h(\mathcal{P})$ is regular for any PDA $\mathcal{P}$ and any fixed h.

A *nondeterministic finite state automaton* (NFA) is a PDA where every edge $q \xrightarrow{a|v} q'$ satisfies $v = \varepsilon$. Equivalently, an NFA $\mathcal{A} = (Q, \Sigma, E, q_0, Q_F)$ is obtained by removing the stack and having edges of the form $q \xrightarrow{a} q'$ for $q, q' \in Q$ and $a \in \Sigma_\varepsilon$. The configurations now consist only of the state, and the concepts of runs, language, and reachability are appropriately modified.

3.2 Succinct PDAs and Doubly Succinct NFAs

We shall consider *succinct* representations of PDAs and NFAs. There are different possible choices for succinct representations; ours are based on *transducers*. A succinct representation is parameterized by a size parameter n, given in unary, and the goal is to define a PDA or NFA whose states are words of length n over a fixed alphabet Σ and whose transitions are succinctly described using transducers (of size at most logarithmically dependent on n). Thus, the number of states of the underlying machine is exponential in the size parameter. We call such a specification *singly succinct* or just *succinct*. When the words used to specify the states are of length 2^n (and transducers are of size at most doubly logarithmically dependent on n), then we call the specification *doubly succinct*.

For a stack alphabet Γ and an input alphabet Σ, let $\mathscr{C} = \{(a, \gamma) \mid a \in \Sigma_\varepsilon, \gamma \in \Gamma \cup \overline{\Gamma} \cup \{\varepsilon\}\}$. A *succinct* PDA is a tuple $C = ((T_c)_{c \in \mathscr{C}}, \Sigma, \Gamma, \Delta, w_0, w_f)$, where Σ is an input alphabet, Γ is a stack alphabet, Δ is a transducer alphabet, $w_0, w_f \in \Delta^n$ for some $n \geq 0$, and T_c, for each $c \in \mathscr{C}$, is a 2-ary transducer over the alphabet Δ. Note that in particular, n is implicitly given as the length of the words w_0 and w_f and is thus polynomial in the size of the succinct PDA.

A succinct PDA represents an (explicit) PDA $\mathcal{E}(C)$ with

- states $Q = \Delta^n$,
- initial state w_0 and final state w_f,
- input alphabet Σ and stack alphabet Γ, and
- transition relation $E \subseteq Q \times \Sigma_\varepsilon \times \hat{\Gamma} \times Q$, where $q \xrightarrow{a|\gamma} q'$ belongs to E iff $(q, q') \in L(T_{(a,\gamma)})$.

The runs, accepting runs, language, etc. of a succinct PDA C refer to those of the explicit PDA $\mathcal{E}(C)$. The size of C is defined as $|C| = |w_0| + |w_f| + \sum_{c \in \mathscr{C}} |T_c|$.

A *succinct* NFA is a succinct PDA where the stack is immaterial, and defined in the expected way.

A *doubly succinct* NFA (dsNFA in short) is a tuple $\mathcal{B} = ((T_a)_{a \in \Sigma_\varepsilon}, \Sigma, \Delta, M, w_0)$, where Σ is an input alphabet, Δ is a transducer alphabet, $M \in \mathbb{N}$ is a number given in *binary*, $w_0 \in \Delta^*$ is a prefix of the initial state, and T_a is a 2-ary transducer over Δ for each $a \in \Sigma_\varepsilon$. In the sequel, we will refer to w_0 as the *initial prefix*: the ability to specify w_0 will be helpful in uniformly specifying a collection of dsNFAs in a succinct mannner. We assume $\{0, 1\} \subseteq \Delta$. A dsNFA represents an (explicit) NFA $\mathcal{E}(\mathcal{B}) = (Q, \Sigma, E, q_0, \{q_f\})$ where

- the set of states $Q = \Delta^M$,
- the initial state q_0 is $w_0 0^{M - |w_0|}$ and the unique final state q_f is 1^M,
- there exists a transition $(p \xrightarrow{a} q) \in E$ for $a \in \Sigma_\varepsilon$ iff $(p, q) \in L(T_a)$.

The size of $\mathcal{B}$ is defined as $|\mathcal{B}| = |w_0| + \lceil \log M \rceil + \sum_{a \in \Sigma_\varepsilon} |T_a|$. Note that, for a dsNFA $\mathcal{E}(\mathcal{B})$, the number of states $|Q|$ is doubly exponential in the description of $\mathcal{B}$, since M is written in binary.

3.3 Succinct Downward Closures

Next, we move on to computing *downward closures* of languages.

The *subword* order $\preccurlyeq$ on finite words over an alphabet Σ is defined as: for all $u, v \in \Sigma^*$ with $u = u_1 \dots u_n$ where each $u_i \in \Sigma$, $u \preccurlyeq v$ if and only if there exist $w_0, w_1, \dots, w_n \in \Sigma^*$ such that $v = w_0 u_1 w_1 u_2 \dots w_{n-1} u_n w_n$. The subword order is a well-quasi-ordering [Higman 1952]. Given any $L \subseteq \Sigma^*$, its *downward closure* (also called *downclosure*) $L{\downarrow}$ is given by $L{\downarrow} = \{u \mid \exists v \in L, u \preccurlyeq v\}$. The downward closure of any language is regular [Haines 1969].

We show the following language theoretic result of independent interest.

Theorem 3.1 (Succinct Downward Closures). *For every succinct* PDA C*, there is a polynomial-time construction of a doubly succinct* NFA $\mathcal{B}$ *of size polynomial in* $|C|$ *such that* $\mathsf{L}(\mathcal{B}) = \mathsf{L}(C){\downarrow}$.

It is well known that an NFA for the downward closure of a PDA can be exponential in the size of the PDA [Bachmeier et al. 2015]. Moreover, it is not difficult to construct examples of succinct PDA for which a downward closure NFA is least doubly exponentially large[2]. A result of Majumdar et al. [2021] shows that NFAs for the downward closure of a PDA are (singly) compressible. Theorem 3.1 strengthens the result and shows that even for succinct PDAs, the NFAs are "doubly-compressible": there is a doubly succinct NFA for the downward closure.

We prove the theorem using the following two lemmas. The first lemma converts a succinct pushdown automaton C to a new succinct pushdown automaton C' whose bounded stack language (for a stack bound that is at most exponential in the size of the succinct PDA) is the same as the downward closure of C. The second lemma turns a succinct pushdown automaton into a doubly succinct NFA.

Lemma 3.2. *Given a succinct pushdown automaton* C*, one can construct in polynomial time a succinct pushdown automaton* C' *and a number* h *in binary such that* $\mathsf{L}_h(C') = \mathsf{L}(C){\downarrow}$.

Lemma 3.3. *Given a succinct pushdown automaton* C *and a number* h *in binary, one can construct in polynomial time a doubly succinct NFA* $\mathcal{B}$ *with* $\mathsf{L}(\mathcal{B}) = \mathsf{L}_h(C)$.

Theorem 3.1 follows by composing the constructions of the two lemmas.

Before proving the lemmas, we first recall the construction from [Majumdar et al. 2021] that takes an ordinary PDA $\mathcal{P} = (Q, \Sigma, \Gamma, \delta, q_0, Q_F)$ and constructs in polynomial time a PDA $\mathcal{P}^\top$ and a bound h such that the downclosure $\mathsf{L}(\mathcal{P}){\downarrow}$ is the bounded stack language of $\mathcal{P}^\top$.

From $\mathcal{P}$, we construct an *augmented automaton* $\mathcal{P}^\top = (Q^\top, \Sigma, \Gamma^\top, E^\top, q_0, Q_F)$ as follows. The states $Q^\top$ consist of the states in Q together with some additional states we describe below. The stack alphabet $\Gamma^\top$ of $\mathcal{P}^\top$ consists of the stack alphabet of $\mathcal{P}$ together with a fresh stack symbol $[p, q]$ for every $p, q \in Q$, i.e., $\Gamma^\top = \Gamma \cup \{[p, q] \mid p, q \in Q\}$.

The set $E^\top$ consists of all the edges in E together with the following additional edges, that we describe next. To begin, define the language

$$M_{p,q}(\mathcal{P}) = \{u \in \Sigma^* \mid \exists v \in \Gamma^* \colon (p, \varepsilon) \xrightarrow{u}{}^* (p, v),\ (q, v) \rightarrow^* (q, \varepsilon)\}.$$

In other words, $M_{p,q}$ consists of words which can be read on a cycle on p such that the stack content created is consumed by a cycle on q.

[2]Take, for example, the language $L_n = \{w w^{\mathrm{rev}} \mid w \in \{\mathrm{a}, \mathrm{b}\}^*,\ |w| = 2^n\}$. For each n, L_n is accepted by a succinct PDA of size polynomial in n, but a downward closure NFA requires at least $2^{2^n}/2^n$ states.

Furthermore, define

$$\eta_{p,q}(\mathcal{P}) = \{a \in \Sigma \mid \exists u \in M_{p,q}(\mathcal{P}),\ |u|_a \geq 1\}.$$

where $|u|_a$ denotes the number of occurrences of the letter $a \in \Sigma$ in u.

With this, we first add the following edges for each $p, q \in Q$ (the edges below use tuples $(R|v)$ where R is a regular language instead of $(a|v)$ where $a \in \Sigma$, but we can simply paste the automaton for R instead to convert to our current formulation):

$$p \xrightarrow{\eta_{p,q}(\mathcal{P})^* | [p,q]} p, \qquad\qquad q \xrightarrow{\eta_{q,p}(\overline{\mathcal{P}})^* | \overline{[p,q]}} q. \tag{1}$$

Here, $\overline{\mathcal{P}}$ denotes the *dual automaton* of $\mathcal{P}$, and is obtained from $\mathcal{P}$ by changing each edge $p \xrightarrow{a|v} q$ into $q \xrightarrow{a|\bar{v}} p$. Note that $\mathsf{L}(\overline{\mathcal{P}})$ is just $\mathsf{L}(\mathcal{P})^{\mathrm{rev}}$, i.e., the set of the reversals of words from $\mathsf{L}(\mathcal{P})$.

Second, we add an edge $p \xrightarrow{\varepsilon|\gamma} q$ for every edge $p \xrightarrow{a|\gamma} q$ for any $a \in \Sigma, \gamma \in \Gamma \cup \overline{\Gamma} \cup \{\varepsilon\}$. Here we essentially just drop the reading of input letters.

The sets $\eta_{p,q}(\mathcal{P})$ are constructed in polynomial time by checking emptiness of the languages $M_{p,q} \cap \Sigma^* a \Sigma^*$ for each $a \in \Sigma$, for which one can construct a PDA $\mathcal{P}_{a,p,q}$. To this end, the state set of PDA $\mathcal{P}_{p,q}$ (without a) for $M_{p,q}$ consists of two disjoint copies of Q, the states of $\mathcal{P}$. Hence, the state set is $\{0,1\} \times Q$. The start state of $\mathcal{P}_{p,q}$ is $(0,p) \in \{0\} \times Q$ and the final state is $(1,q) \in \{1\} \times Q$. The transitions in $\{0\} \times Q$ are the same as in $\mathcal{P}$ without any change. In the transitions in $\{1\} \times Q$, we drop the reading of input letters (replacing them with ε in the transitions). Finally, we add an ε-transition from $(0,p) \in \{0\} \times Q$ to $(1,q) \in \{1\} \times Q$ which does not change the stack. Intersecting this PDA with the regular language $\Sigma^* a \Sigma^*$ gives us the PDA $\mathcal{P}_{a,p,q}$.

In the construction of [Majumdar et al. 2021], the emptiness checks for $\mathcal{P}_{a,p,q}$ were computed separately in polynomial time. The main result of [Majumdar et al. 2021] shows that there is a bound $h = O(|Q^\top|^2)$ such that $\mathsf{L}_h(\mathcal{P}^\top) = \mathsf{L}(\mathcal{P})\downarrow$. However, for our construction on succinct PDAs, this leads to exponentially many checks, each potentially taking exponential time. (Note that the emptiness problem for succinct PDA is EXPTIME-complete.) Therefore, we have to modify the construction as follows.

We observe that the emptiness of the languages $M_{p,q} \cap \Sigma^* a \Sigma^*$, for $p, q \in Q$ and $a \in \Sigma$, can in fact be implemented directly within $\mathcal{P}^\top$ using its own stack on the fly by adding some additional states (corresponding to the union of all the states of all the $\mathcal{P}_{a,p,q}$). Any point when $\mathcal{P}^\top$ is in state p, it nondeterministically guesses a letter $a \in \Sigma$ and puts the special stack symbol \$ on its stack. Then, it runs the PDA $\mathcal{P}_{a,p,q}$ for $M_{p,q} \cap \Sigma^* a \Sigma^*$ using its own stack. If $\mathcal{P}_{a,p,q}$ accepts, the stack is popped all the way down including the \$ symbol and the computation is continued from there on. This requires augmenting the stack alphabet $\Gamma^\top$ with \$ and †. Since there is a run reaching the final state of $\mathcal{P}_{a,p,q}$ iff there is a run reaching the final state with a stack bounded by $O(|Q^\top|^2)$ by a hill cutting argument, we obtain $\mathsf{L}_h(\mathcal{P}^\top) = \mathsf{L}(\mathcal{P})\downarrow$ where $h = O(|Q^\top|^2)$.

We now proceed on to the proofs of the lemmas.

Proof of Lemma 3.2. The proof idea follows mostly from the adaptation of the construction from [Majumdar et al. 2021] to succinct PDAs that we discussed above.

The key idea is to think of state of a succinct machine as a tape on which polynomial space computations can be performed. By using a product alphabet $\Delta \times \Delta$ for the tape of C', we further think of it as having two separate tracks. The first track is used to simulate the transitions of C while the second is used to perform 'on the fly' computations in order to check for the emptiness of $M_{p,q} \cap \Sigma^* a \Sigma^*$, in order to decide if one is allowed to take edges of the form in Equation 1. We argue that polynomial amount of tape space suffices in order to accomplish this.

A detail here is that we can no longer utilize a fresh stack symbol $[p, q]$ for every $p, q \in Q$. The reason is the existence of now exponentially many choices for q and p, so this would result in a blow-up of the stack alphabet. As a remedy for this issue, we instead add the brackets '[' and ']' as fresh stack symbols, as well as adding the transducer alphabet Δ of the succinct PDA to the new stack alphabet. That way we can just push and pop the entire string $[p\#q]$ in $2n + 3$ steps, where # is another new stack symbol used as a separator. ∎

Proof of Lemma 3.3. Given a succinct PDA $\mathcal{C} = ((T_c)_{c\in\mathscr{C}}, \Sigma, \Gamma, \Delta, w_0, w_f)$ with $|w_0| = n$, we construct a doubly succinct NFA $\mathcal{B} = ((T_a)_{a\in\Sigma_\varepsilon}, \Sigma, \Delta', w_0', M)$ as follows. The states of the explicit NFA $\mathcal{E}(\mathcal{B})$ are used to store configurations (q, w), where $q \in \Sigma^n$ is a state of the explicit PDA $\mathcal{E}(\mathcal{C})$, and $w \in \Gamma^*$ with $|w| \leq h$ is the bounded stack.

We define the transducer alphabet $\Delta' = \Delta \cup \Gamma \cup \{\$, \#, \dagger\}$ where $\#, \dagger$ are new symbols, and declare $M = |w_0| + h + 3$. A configuration (q, w) is represented as the string $\$q\#w\dagger 0^{h-|w|}$, where the portion of the string between # and † encodes the stack contents. The prefix w_0' is defined to be $\$w_0\#\dagger$ so that the initial state then becomes $\$w_0\#\dagger 0^h$.

There is a transducer T_a for each $a \in \Sigma_\varepsilon$ that converts an input string representing the configuration (p, w) to an output string (q, w') for a transition $(p, w) \xrightarrow{a|v} (q, w')$ of the explicit PDA. The transducer nondeterministically guesses $c = (a, v)$ and applies T_c to p, converting it to q. This part ends when the letter # is seen. It then converts w to w' where the two strings are related by the stack operation v. The symbols on the stack that are not at the top are maintained as they are; the transducer nondeterministically guesses when the last stack symbol occurs in w and updates the stack based on whether $v \in \Gamma$, $v \in \bar{\Gamma}$, or $v = \varepsilon$, keeping the necessary information in its state.

The transducer T_ε nondeterministically checks letter by letter if the current state is $\$w_f\#w$ for some w and transforms the state to 1^M, the accepting state of $\mathcal{B}$. ∎

4 SUCCINCT DOWNWARD CLOSURES OF TASKS

4.1 Tasks in a DCBP

The DCBP model does not assign task identifiers. Nevertheless, it is convenient to be able to talk about the run of a single task along the execution. One can formally introduce unique task identifiers by modifying the thread step and the operational semantics rules to carry along the identifier in the local state. In this way, we can talk about the run of a single task, the multiset of tasks spawned by a given task, etc.

With this intuition, consider the run of a specific task t, that starts executing from some initial assignment $\vec{a}$ to the global variables (henceforth simply called *global state*) with an initial stack symbol γ and initial local variable values $\vec{b} = \vec{0}$ (henceforth called *local state*), from the moment it is started by the thread pool (by executing the Pick rule). In the course of its run, the thread executing the task t updates its own local stack and spawns new tasks, but it can also get swapped out and swapped back in.

The run of such a task t corresponds to the run of an associated succinct PDA $\mathcal{C}_{(\vec{a},\gamma)}$ (called a succinct task-PDA) that can be extracted from $\mathcal{D}$; our construction is a simple modification of the construction in [Atig et al. 2009] which now accounts for succinctness. The reason we require a succinct PDA in our setting is that the global states are now succinctly represented by a set of global Boolean variables and the number of possible assignments to these variables is exponential. Since the different succinct task-PDAs only differ in their initial state based on the value of $\vec{a}$ and otherwise have the same set of transducers defining them, the entire set of succinct task-PDAs have a small description. In the sequel, we will describe the succinct task-PDA without mentioning the initial and final states and simply denote it by $\mathcal{C}_\gamma = ((T_p')_{p\in\mathscr{C}}, \Sigma', \Gamma, \Delta')$. If Γ is the stack alphabet of

the DCBP, and $\Delta = \{0, 1\}$, then the alphabet Σ' of the succinct PDA is given by $\Sigma' = \Gamma \dot\cup \tilde\Gamma \dot\cup \Delta \dot\cup \{\bot\}$, where $\tilde\Gamma = \{\tilde\gamma \mid \gamma \in \Gamma\}$ is a decorated copy of Γ, and $\Delta' = \Gamma \dot\cup \Delta \dot\cup \{0', 1', \#, \$_1, \$_2, \$_3, \dagger, \bot_1, \bot_2\}$. When $\vec{a}$ is also specified, we will call $C_{(\vec{a},\gamma)}$ an initialised task-PDA. The initial state of the initialised task-PDA is then taken to be $\#\vec{a}\#\vec{\mathbf{0}}$ and the final state is the all 1s string.

The succinct PDA C_γ updates the global variables, local variables and the stack using the transducers in $\mathcal{T}_c$. The context switches are nondeterministically guessed and there are two kinds of state jumps possible. First, a jump $(\vec{a}_2, \tilde\gamma_2, \vec{a}_3)$ in C_γ corresponds to the thread being switched out while moving to global state $\vec{a}_2$ and later resuming at global state $\vec{a}_3$ with γ_2 on top of its stack (without being active in the interim). Second, a jump $(\vec{a}_2, \bot)$ corresponds to the last time the thread is swapped out during the execution of the associated task (leading to global state $\vec{a}_2$) or when the thread is terminated. Both of these kinds of jumps are made visible as part of the input alphabet by using the letters $\{0, 1, \bot\} \dot\cup \Gamma$ in Σ', one symbol at a time. Additionally, in the case of an initialised task-PDA $C_{(\vec{a},\gamma)}$, its specified global state $\vec{a}$ is also made visible (in the same manner as the jumps) at the start of its execution. We describe the details below.

The state of C is of the form $\square\vec{a}_1\#\vec{b}_1$ where $\vec{a}_1$ is a string in $\{0,1\}^m$ representing the global variables, $\vec{b}_1$ is a string in $\{0,1\}^n$ representing the local variables and $\square$ is a special symbol which is used to store information about whether the automaton is currently in the midst of outputting a jump transition or the initial global state.

Initialization. We want an initialised task-PDA $C_{(\vec{a},\gamma)}$ to output the specified global state $\vec{a}$, using the transducers $T'_{0,\varepsilon}$ and $T'_{1,\varepsilon}$. When seeing # as the first state symbol, either of these transducers will look for the first occurrence of its output symbol, 0 or 1, respectively, and replace it with the corresponding symbol $0'$ or $1'$, respectively. From the initial state $\#\vec{a}\#\vec{\mathbf{0}}$ this eventually results in $\#\vec{a}'\#\vec{\mathbf{0}}$, once all of $\vec{a}$ has been output. Here $\vec{a}'$ is the counterpart to $\vec{a}$, where every symbol in $\{0,1\}$ has been replaced with its primed version. Then finally the transducer $T'_{\varepsilon,\varepsilon}$ looks for a state of the form $\#\{0',1'\}^m\#0^n$ and changes each $0'$, respectively $1'$, back to 0, respectively 1, while also replacing the first # with †. This results in the state $\dagger\vec{a}\#\vec{\mathbf{0}}$.

Non-jump Transitions. For each $p = (\gamma, v)$ where $\gamma \in \Gamma \cup \{\varepsilon\}$, and $v \in \Gamma \cup \bar\Gamma \cup \{\varepsilon\}$, the transducer T'_p of the task-PDA simulates $T_p \in \mathcal{T}_c$ of the DCBP on $\vec{a}_1\vec{b}_1$ while ignoring the remaining symbols in $\square\vec{a}_1\#\vec{b}_1$. When simulating a non-jump transition, the first symbol of the state is always †.

Context Switching. Let $(\vec{a}_2, \gamma_2, \vec{a}_3)$ be a context switch, where the thread is swapped out when the global state is $\vec{a}_2$ with the top of stack being γ_2 and then swapped back in when global state is $\vec{a}_3$. Context switching takes place in three phases.

- **Phase 1** Let us assume that a thread is swapped when moving from global state $\vec{a}_1$ to $\vec{a}_2$. This is done by application of a transducer from $\mathcal{T}_s$ of the DCBP. While doing so, C changes the value of the first state symbol from † to $\$_1$ to indicate that it is Phase 1 of a context switch. Transducers $T'_{0,\varepsilon}$ and $T'_{1,\varepsilon}$ are used to output the string $\vec{a}_2$. Since this is done one symbol at a time, the exact position is remembered by changing the binary string $\vec{a}_2$ into its corresponding primed version $\vec{a}'_2$. For example 110 is converted to $1'10$, then $1'1'0$ and finally to $1'1'0'$. Once all bits have been output, the first state symbol is changed from $\$_1$ to $\$_2$ to signal the start of Phase 2.
- **Phase 2** The succinct PDA C uses its transducer $T'_{\tilde\gamma_2,\bar\gamma_2}$ in order to output $\tilde\gamma_2$ while at the same time checking that γ_2 is indeed the top of the stack, by popping it from the stack. During this, the global state $\vec{a}_3$ is guessed as a resumption point for the context switch, and its primed version $\vec{a}'_3$ is saved in the state in place of $\vec{a}'_2$. Additionally, the first state symbol is changed from $\$_2$ to $\$_3$ to move onto the next phase, and the # symbol between the global and local

state is rewritten to γ_2 in order to momentarily store this stack symbol. The state at this point is of the form $\$_3 \vec{a}'_3 \gamma_2 \vec{b}_1$.

Phase 3 The global state $\vec{a}_3$ is read out in binary similar to Phase 1, except the state symbols now change from the primed version to their non-primed counterparts, i.e. replacing $\vec{a}'_3$ with $\vec{a}_3$ in the state. Then C simulates the transducer $T_{\gamma_2} \in \mathcal{T}_r$ of the DCBP on global state $\vec{a}_3$ to essentially resume the thread. This is done as part of the transducer $T'_{\varepsilon,\gamma_2}$ in order to push γ_2 back onto the stack. Additionally, γ_2 in the state is changed back to # and the first symbol $\$_3$ is turned back to †.

Termination. This can take place in two ways: either a thread is switched out and not switched back in again (case 1), or a transition rule from T_t is applied (case 2).

Case 1 Here, C proceeds as in Phase 1 of context switching but can nondeterministically choose to write $\bot_1$ on the first cell instead of $\$_1$ to indicate imminent termination. The rest of Phase 1 is carried out as before, transitioning from $\vec{a}_1$ to $\vec{a}_2$, writing out the bits of $\vec{a}_2$, and rewriting the first symbol of the state to $\bot_2$. However, instead of the steps in Phase 2, C outputs $\bot$ and moves to its final state (the all 1s string) on seeing $\bot_2$ as the first state symbol.

Case 2 Here, C also proceeds like in Phase 1 of context switching, but applies $T_t \in \mathcal{T}_t$ of the DCBP to transition from $\vec{a}_1$ to $\vec{a}_2$, instead of a transducer from $\mathcal{T}_s$. During this, the first state symbol is also changed to $\bot_2$. Then C outputs $\bot$ and moves to its final state as in Case 1.

4.2 Alphabet-Preserving Downward Closures

For an alphabet $\Theta \subseteq \Sigma$, let $\pi_\Theta \colon \Sigma^* \to \Theta^*$ denote the projection onto Θ^*. In other words, for $u \in \Sigma^*$, the word $\pi_\Theta(u)$ is obtained from u by deleting all occurrences of letters in $\Sigma \setminus \Theta$. Let $L \subseteq \Sigma^*$, let $\Theta \subseteq \Sigma$ be a subset, and let $k \in \mathbb{N}$. We define

$$L\downarrow_{\Theta,k} = \{u \in \Sigma^* \mid \exists v \in L \colon u \preccurlyeq v \text{ and } \pi_\Theta(u) = \pi_\Theta(v) \text{ and } |v|_\Theta \leq k\}.$$

The following theorem shows that the alphabet-preserving downward closure has a doubly succinct representation.

Theorem 4.1. *Given a succinct* PDA C *over the input alphabet* Σ*, a subset* $\Theta \subseteq \Sigma$*, and* $K \in \mathbb{N}$ *written in binary, one can construct in polynomial time a doubly succinct* NFA $\mathcal{B}$ *with* $\mathsf{L}(\mathcal{B}) = \mathsf{L}(C)\downarrow_{\Theta,K}$.

Remark 2. *Recall that for the ordinary downward closure* $\mathsf{L}(C)\downarrow$*, one can construct a singly succinct NFA for* $\mathsf{L}(C)\downarrow$*. However, for the* Θ*-preserving downward closure* $\mathsf{L}(C)\downarrow_{\Theta,K}$*, this is not true. Consider, for example, the language* $L = \{ww^{\mathrm{rev}} \mid w \in \{\mathsf{a},\mathsf{b}\}^*\}$ *where* $w^{\mathrm{rev}} = w_n \dots w_1$ *is the reverse of the word* $w = w_1 \dots w_n$*. Then, for* $K = 2^n$ *and* $\Theta = \{\mathsf{a},\mathsf{b}\}$*, the set* $L\downarrow_{\Theta,K}$ *consists of all palindromes of length* 2^n*. Clearly, an NFA for* $L\downarrow_{\Theta,K}$ *requires at least* $2^{2^{n-1}}$ *states.*

We begin with a simple lemma that states that succinct PDAs can be modified to keep a count. The proof of the lemma follows by modifying the transducers to maintain and increment the count of the occurrences of Θ on the string.

Lemma 4.2. *Given a succinct* PDA C *over the input alphabet* Σ*, a subset* $\Theta \subseteq \Sigma$*, and a number* K *in binary, one can construct in polynomial time a succinct* PDA C' *over the alphabet* $\Sigma \cup \{\#\}$*, where* $\# \notin \Sigma$ *is a new symbol, such that*

$$L(C') = \{w\#^l \mid w \in L(C), |w|_\Theta + l = K\}$$

Proof. Let $u_K \in \{0,1\}^{\lceil \log K \rceil}$ be the binary encoding of K. We obtain C' from C as follows.

We extend the initial state w_0 of C to $w_0 \dagger u_K$ using a new separator symbol †. All transducers not corresponding to input symbols in Θ are extended to not change the part of the state after †. The transducers that do correspond to Θ implement binary subtraction by 1 on this part of

the state. We add a new transducer $T_\#$ for # that always checks whether the first part of the state matches the final state w_f of C without changing it. If this check succeeds, $T_\#$ also implements binary subtraction by 1 on the part after †. The final state of C' is then defined as $w_f \dagger 0^{\lceil \log K \rceil}$. ∎

Given a language $L \subseteq \Sigma^*$ and a subset $\Theta \subseteq \Sigma$ of the alphabet, define

$$L|_{\Theta,k} := \{w \mid w \in L, |w|_\Theta = k\},$$

i.e., those words that contain exactly k letters from Θ. In the next step, we extract precisely these words from the downward closure.

Lemma 4.3. *Given a doubly succinct* NFA $\mathcal{B} = ((T_a)_{a\in\Sigma_\varepsilon}, \Sigma, \Delta, w_0, M)$, $\Theta \subseteq \Sigma$ *and* $k \in \mathbb{N}$ *in binary, one can construct in polynomial time another doubly succinct* NFA $\mathcal{B}'$ *such that* $L(\mathcal{B}') = L(\mathcal{B})|_{\Theta,k}$.

Proof. $\mathcal{B}'$ simply keeps a counter in its state and increments it whenever a letter from Θ is read while simulating $\mathcal{B}$. When the final state of $\mathcal{B}$ is reached, $\mathcal{B}'$ can nondeterministically guess and subsequently check that the counter value is exactly k, moving to its own final state afterwards. Such a check is feasible by a transducer of $\mathcal{B}'$ since the bit representation of k is small. ∎

We are now ready to prove Theorem 4.1.

Proof. Suppose we are given a succinct PDA C and a number K written in binary. We build a succinct PDA C' which extracts those words of $L(C)$ that contain at most K letters from Θ (padded with #s) using Lemma 4.2. Then, using Theorem 3.1, we construct a doubly succinct NFA $\mathcal{B}$ such that $\mathsf{L}(\mathcal{B}) = \mathsf{L}(C')\downarrow$. Then, applying Lemma 4.3, we get a doubly succinct NFA $\mathcal{B}'$ such that

$$\mathsf{L}(\mathcal{B}') = \{w \mid w \in \mathsf{L}(\mathcal{B}) \text{ and } |w|_{\Theta\cup\{\#\}} = K\}.$$

Finally, modify $\mathcal{B}'$ by changing all transitions reading the input letter # to read ε instead. To this end we construct the doubly succinct NFA $\mathcal{B}''$ from $\mathcal{B}'$ by simply combining the two transducers $T'_\#$ and T'_ε of $\mathcal{B}'$ into a single transducer T''_ε. Observe that $\mathsf{L}(\mathcal{B}'') = \mathsf{L}(C)\downarrow_{\Theta,K}$. ∎

We can apply the theorem on any initialised task-PDA $C_{(\vec{a},\gamma)}$, giving us the following corollary.

Corollary 4.4 (Succinct Task Downclosure). *Given a* DCBP $\mathcal{D}$*, a succinct task-*PDA $C_{(\vec{a},\gamma)}$ *and a number* K *in binary, there is a polynomial time procedure to compute a (polynomial size) doubly succinct* NFA $\mathcal{B}$ *such that* $\mathsf{L}(\mathcal{B}) = \mathsf{L}(C_{(\vec{a},\gamma)})\downarrow_{\Theta,K}$*. Further, for a succinct task-*PDA $C_{(\vec{a}',\gamma)}$ *with* $\vec{a}' \neq \vec{a}$*, the corresponding* dsNFA $\mathcal{B}'$ *only differs from* $\mathcal{B}$ *in the initial prefix.*

5 FROM DCBP TO VASS COVERABILITY

Corollary 4.4 allows us to represent the state of each thread as a succinct machine. The next goal in the proof of Theorem 2.1 is to represent the entire configuration of a DCBP. Intuitively, the configuration will maintain the global state and the state of all the threads in the thread pool as "control states" and maintain a counter for each stack symbol that tracks the number of (unstarted) tasks of that type that have been spawned. The structure suggests the use of *vector addition systems with states* (VASS) as a representation. The main challenge is to represent the states of exponentially many (in the representation of N) threads in a concise way.

5.1 Vector Addition Systems with States and their Succinct Versions

A *vector addition system with states* (VASS) is a tuple $V = (Q, I, E, q_0, q_f)$ where Q is a finite set of *states*, I is a finite set of *counters*, $q_0 \in Q$ is the initial state, $q_f \in Q$ is the final state, and E is a

finite set of edges of the form $q \xrightarrow{\delta} q'$ where $\delta \in \{-1,0,1\}^I$.[3] A *configuration* of the VASS is a pair $(q,u) \in Q \times \mathbb{M}[I]$. The elements of $\mathbb{M}[I]$ and $\{-1,0,1\}^I$ can also be seen as vectors of length $|I|$ over $\mathbb{N}$ and $\{-1,0,1\}$, respectively, and we sometimes denote them as such. The edges in E induce a transition relation on configurations: there is a transition $(q,u) \xrightarrow{\delta} (q',u')$ if there is an edge $q \xrightarrow{\delta} q'$ in E such that $u'(p) = u(p) + \delta(p)$ for all $p \in I$. A *run* of the VASS is a finite or infinite sequence of configurations $c_0 \xrightarrow{\delta_0} c_1 \xrightarrow{\delta_1} \dots$ where $c_0 = (q_0, \vec{\mathbf{0}})$. A finite run is said to reach a state $q \in Q$ if the last configuration in the run is of the form $(q, \mathbf{m})$ for some multiset $\mathbf{m}$. An *accepting* run is a finite run whose final configuration has state q_f. The *language (of the* VASS*)* is defined as

$$\mathsf{L}(V) = \{w \in (\{-1,0,1\}^I)^* \mid w = w_1 \cdots w_l, \text{ there is a run } (q_0, \vec{\mathbf{0}}) = c_0 \xrightarrow{w_1} \dots \xrightarrow{w_l} c_l = (q_f, u)\}.$$

The size of the VASS V is defined as $|V| = |I| \cdot |E|$.

The *coverability problem* for VASS is

Given A VASS V.
Question Is there an accepting run?

The coverability problem for VASS is EXPSPACE-complete [Lipton 1976; Rackoff 1978].

A succinct variant of VASS. Let $I = \{1,2,\dots,d\}$ for some $d \in \mathbb{N}$, let $\bar{I}$ be a disjoint copy of I and define $\hat{I} := I \cup \bar{I}$ and $\hat{I}_\varepsilon := \hat{I} \cup \{\varepsilon\}$. A *transducer controlled vector addition system with states* (TCVASS, for short) is a tuple $\mathcal{V} = ((T_i)_{i \in \hat{I}_\varepsilon}, \Delta, M)$ where each T_i is a 2-ary transducer over $\Delta \supseteq \{0,1\}$, and M is a number in binary. This induces an explicit VASS $\mathcal{E}(\mathcal{V}) = (Q, I, E, q_0, q_f)$ given by

- $Q = \Delta^M$,
- $E \subseteq Q \times \mathbb{Z}^d \times Q$ is the set of triples (q,u,q') satisfying one of the following conditions:
  (1) $(q,q') \in L(T_\varepsilon)$ and $u = \vec{\mathbf{0}}$,
  (2) there exists $i \in I$, $(q,q') \in L(T_i)$ and $u = e_i$ where e_i denotes the vector with 1 in the i^{th} component and 0 otherwise, or
  (3) there exists $\bar{i} \in \bar{I}$, $(q,q') \in L(T_{\bar{i}})$ and $u = -e_i$.
- $q_0 = 0^M$ and $q_f = 1^M$.

We write $q \xrightarrow{u} q'$ to denote a transition of $\mathcal{E}(\mathcal{V})$ from its state q to state q' while executing the instruction u. If there exists a finite run $0^M \xrightarrow{u_1} q_1 \xrightarrow{u_1} q_2 \dots \xrightarrow{u_l} 1^M$, we say the word $u_1 \dots u_l$ is in the language of $\mathcal{V}$, which is denoted $\mathsf{L}(\mathcal{V})$.

Given a TCVASS $\mathcal{V} = ((T_i)_{i \in \hat{I}_\varepsilon}, \Delta, M)$, its *associated* dsNFA is given by $\mathcal{B}(\mathcal{V}) = ((T_i)_{i \in \hat{I}_\varepsilon}, \hat{I}, \Delta, M)$. In the sequel, we will associate i (resp. $\bar{i}, \varepsilon$) with e_i (resp. $-e_i, \vec{\mathbf{0}}$), so that the languages of $\mathcal{V}$ and $\mathcal{B}(\mathcal{V})$ are over the same alphabet $\hat{I}$. The size of $\mathcal{V}$ is simply defined as $|\mathcal{V}| = |\mathcal{B}(\mathcal{V})|$.

Our main result of this section is that, despite the succinctness, the coverability problem for TCVASS is EXPSPACE-complete.

Theorem 5.1. *The coverability problem for* TCVASS *is* EXPSPACE*-complete.*

Proof. The lower bound follows from the lower bound for VASS. To show the upper bound, given a TCVASS $\mathcal{V}$, we show that we can construct in polynomial time a VASS $\mathcal{V}'$ such that the final state is reachable in TCVASS if and only if the final state is reachable in $\mathcal{V}'$.

[3] A more general definition of VASS would allow each transition to add an arbitrary vector over the integers. We instead restrict ourselves to the set $\{-1,0,1\}$, since this suffices for our purposes, and the EXPSPACE-hardness result by Lipton [1976] already holds for VASS of this form.

We begin with a language-theoretic observation. Let the VASS V_I be defined as $V_I = (\{q_0\}, I, E, q_0, q_0)$ where for each $i \in I$ (resp. $i \in \bar{I}$) there is a transition $q_0 \xrightarrow{e_i} q_0$ (resp. $q_0 \xrightarrow{-e_i} q_0$); as well as a transition $q_0 \xrightarrow{\vec{0}} q_0$. For any TCVASS $\mathcal{V}$, note that

$$\mathsf{L}(\mathcal{V}) = \mathsf{L}(\mathcal{B}(\mathcal{V})) \cap \mathsf{L}(V_I).$$

To prove the theorem, it suffices to construct a VASS $\mathcal{V}''$ with $\mathsf{L}(\mathcal{V}'') = \mathsf{L}(\mathcal{B}(\mathcal{V}))$: This would enable us to build a VASS for $\mathsf{L}(\mathcal{V}'') \cap \mathsf{L}(V_I)$. The VASS $\mathcal{V}''$ can be constructed due to two facts:

(1) A classical result that an exponentially bounded automaton can be simulated by a polynomial size VASS, which can be constructed in polynomial time (see [Esparza 1998; Lipton 1976]) and
(2) the sequence of reductions used in the above result having a property we call *local simulation*. For machines $\mathcal{M}_1, \mathcal{M}_2$ we say $\mathcal{M}_1$ is simulated by $\mathcal{M}_2$ locally if every step of $\mathcal{M}_1$ is simulated by some fixed sequence of steps of $\mathcal{M}_2$ which only depends on a constant sized local part of the global configuration. By associating the last of the corresponding sequence of steps of $\mathcal{M}_2$ with the action of the single step of $\mathcal{M}_1$ and the rest of the steps of $\mathcal{M}_2$ with ε, we ensure that $\mathsf{L}(\mathcal{M}_2) = \mathsf{L}(\mathcal{M}_1)$. ∎

Remark 3. *An alternate proof that coverability of* TCVASS *is in* EXPSPACE *follows from the multi-parameter analysis of decision problems for* VASS *by Rosier and Yen [1986], where they show that the Rackoff argument can be modified to show that the coverability problem for a* VASS *with k counters, n states, and constants bounded by l can be solved in $O((l + \log n)2^{ck \log k})$ nondeterministic space. For* TCVASS*, the number of states n is doubly exponential in the description of the* TCVASS*, so this result shows that coverability is in* EXPSPACE*. We believe our argument is conceptually simpler.*

5.2 From DCBP to TCVASS

Theorem 5.2. *Given a* DCBP $\mathcal{D} = (V_{\mathrm{gl}}, V_{\mathrm{loc}}, \Gamma, \mathcal{T}_c, \mathcal{T}_s, \mathcal{T}_r, \mathcal{T}_t, \vec{a}_0, \gamma_0)$*, its global state $\vec{a}_f$ and two binary numbers K, N, we can construct in polynomial time a* TCVASS $\mathcal{V} = ((T_i')_{i \in \hat{I}_\varepsilon}, \Delta', M')$ *such that $\vec{a}_f$ is reachable in $\mathcal{D}$ via an N-thread-pooled, K-context switch bounded execution if and only if the final state of $\mathcal{E}(\mathcal{V})$ is reachable in $\mathcal{E}(\mathcal{V})$.*

The proof of the theorem requires us to simulate up to N different threads of the DCBP. To this end we compute dsNFAs for each type of task these threads could be executing, using Corollary 4.4. The simulation then needs to keep track of up to N different dsNFA states at the same time. Thus, we construct $\mathcal{V}$ in such a way that its state is comprised of exponentially many segments (since N was given in binary) each storing a dsNFA state, separated by special symbols. Initializing this segmentation is formalized in the following auxiliary result.

Lemma 5.3. *Given numbers M, M' in binary and m in unary such that $M' = M \cdot M''$, there exists a* dsNFA *$\mathcal{B}_{m,M,M'}$ over the empty alphabet, polynomial in the size of M' such that it can reach a state s of the form $s = \$^m(0^M\#)^{M''}$ from an initial state $0^{m+(M+1)M''}$. Moreover, s is the unique state in the set of reachable states of $\mathcal{B}_{m,M,M'}$ which ends with the symbol $\#$.*

Proof. Let us consider for simplicity the case when $m = 0$. We can think of the state of a dsNFA $\mathcal{B}$ as the tape of a Turing machine $\mathcal{M}$. While a transducer only reads its tape from left to right once, a Turing machine can read a particular cell and move either left or right. However, the transducer can simulate two-way movement as follows:
Let $\mathcal{M}$ operate over alphabet Σ, then $\mathcal{B}$ operates over alphabet $\Sigma \cup \tilde{\Sigma}$ where $\tilde{\Sigma}$ is a disjoint copy of Σ. The disjoint copy $\tilde{\Sigma}$ is used by $\mathcal{B}$ to keep track of the position of the head. For example, if the tape contents of $\mathcal{M}$ are w_1abw_2 for $w_1, w_2 \in \Sigma^*$ and $a, b \in \Sigma$ with the tape head on b, then the tape

contents of $\mathcal{B}$ are $w_1 a \tilde{b} w_2$. When $\mathcal{M}$ executes a transition where b is replaced by c and the head moves left, $\mathcal{B}$ guesses, at the time of reading a, that the tape head is on the next symbol and thus replaces a by $\tilde{a}$ and then replaces $\tilde{b}$ by c. (In case the transition simulated is one where the head is moved right, this guess is not required.) Thus in the rest of the proof, we will describe the actions of the required dsNFA as if it were a Turing machine.

Using a constant number n_1 of states, one can implement doubling, i.e. an input of the form $1^n 0^{n'}$ can be converted to $1^{2n} 0^{n'-n}$. The machine turns the first 1 into a decorated copy $\overline{1}$, then moves to the right till it finds the first 0, converting it into a $\overline{1}$. It then moves all the way back to the beginning of the tape, making sure that it encounters a 1. After i such moves, the state is of the form $\overline{1}^i 1^{n-i} \overline{1}^i 0^{n'-i}$. Finally we reach a point when no more 1s are present between the two blocks of $\overline{1}$s i.e. the state is $\overline{1}^{2n} 0^{n'-n}$. We now convert all $\overline{1}$s to 1s.

Let $N = (M+1)M''$. We initialize by converting 0^N to 10^{N-1}. We then implement a counter which repeatedly implements the doubling module. In order to create a string $1^{2^k} 0^{N-2^k}$, we need a counter which uses $k+1$ bits. Thus the counter can be implemented using n_2 many states where n_2 is a number that is linear in k, by first creating a string $u = 10^k$ and comparing the value of the counter with u after each increment. For an arbitrary number $M = \sum_{i=0}^{\ell} 2^{k_i}$, we can separately implement the above module for each power 2^{k_i}. This takes roughly n_2^2 many states. We can now create the string $0^M \# 0^{M'-M-1}$.

As the last step to obtain the required string $(0^M\#)^{M''}$, we create another counter which counts up to M'' using n_3 states, where n_3 is polynomial in the bit size of M''. Using this, we can repeat the above process M'' times to get the string $(0^M\#)^{M''}$. We have used a total of $n_1 + n_2^2 + n_3$ states which is polynomial in the bit size of M'. Note that the last symbol remains a 0 until the very end when it is changed into a #. Hence we also satisfy the condition that the target state is the only reachable state with a # as the last symbol.

Let us now consider the case that $m > 0$. Then, the dsNFA can simply count up to m using the states of the transducers since m is given in unary. It converts the string $0^{m+(M+1)M''}$ to $\$^m 0^{(M+1)M''}$ first and then runs the above procedure on the suffix $0^{(M+1)M''}$. ∎

Proof of Theorem 5.2. Application of Corollary 4.4 to the initialised succinct task-PDA $C_{(\vec{a},\gamma)}$ gives us a dsNFA $\mathcal{B}_{(\vec{a},\gamma)} = ((T_{\sigma,\gamma})_{\sigma\in\Sigma_\varepsilon}, \Sigma, \Delta, w_0(\vec{a}), M)$, where each $T_{\sigma,\gamma}$ has state set $Q_{\sigma,\gamma}$. We call $\mathcal{B}_{(\vec{a},\gamma)}$ a task-dsNFA. Note that for initialised succinct task-PDA $C_{(\vec{a}',\gamma)}$, the corresponding $\mathcal{B}_{(\vec{a}',\gamma)}$ only differs from $\mathcal{B}_{(\vec{a},\gamma)}$ in its initial prefix $w_0(\vec{a}')$. The tuple $(\vec{a},\gamma)$ is called the *type* of a task. Since we need a counter for each stack symbol (to track the unstarted tasks), we define $I = \{1,\ldots,d\}$, where $d = |\Gamma|$. We also write $\Gamma = \{\gamma_1,\gamma_2,\ldots,\gamma_d\}$.

The construction of $\mathcal{V}$ essentially involves running N different task-dsNFAs in parallel while keeping track of the tasks using its counters. The local information of the N different threads are stored in the global state (henceforth also called the *tape*) of $\mathcal{V}$, with separator symbols between consecutive threads. We call the space between two consecutive separators a *segment*. This means that the tape contents are always of the form $u\square_1 w_1 \square_2 \ldots \square_N w_N \square_{N+1}$ where u is a string of length m that is used to store the global state of the DCBP at schedule points to help coordinate the communication between the N different threads, the $\square_i$ are *separator* symbols which indicate whether a segment is currently *occupied* by a thread (in which case the string w_i contains the configuration of the thread) or *unoccupied* (in which case the segment is filled with 0s). We briefly outline the salient features of $\mathcal{V} = ((T'_i)_{i\in\hat{I}_\varepsilon}, \Delta', M')$ below:

- $\Delta' = \Delta \cup \{\$_1, \#_1, \dagger_1, \tilde{\dagger}\} \cup \Gamma \cup \{0, 1, \bot\}$, where
  - $\#_1, \dagger_1$ are separators which are used to indicate occupied and unoccupied segments respectively.

  - when the DCBP currently has an active thread then $u = \$_1^m$; otherwise it is at a schedule point and $u = \vec{a}$ for some $\vec{a} \in \{0,1\}^m$,
  - the symbols in $\Gamma \cup \{0,1\}$ are used to mark each thread with its state and top of stack in order to facilitate context switches,
  - $\tilde{\dagger}$ is a separator used to mark a segment being initialised,
  - $\perp$ is used to mark a thread meant to be terminated.
- $M' = m + 1 + N(M + 2m + 3)$,
- The transducers T_i' consist of different modules as outlined below.

(1) For each $i \in I$, T_i' has a single module which is used when a step performed by a particular thread spawns a new task γ_i.
(2) For each $\bar{i} \in \bar{I}$, $T_{\bar{i}}'$ has a single module which used to initialize a task of type γ_i.
(3) T_ε' has the following modules:
  (a) init, which creates the initial set of separators between segments,
  (b) start, which helps initialize a task-dsNFA correctly,
  (c) inter, which interrupts a thread,
  (d) resump, which resumes a thread,
  (e) term, which handles termination of threads,
  (f) clean, which cleans up a segment after termination of a thread,
  (g) final, which detects when the DCBP has reached global state $\vec{a}_f$, and
  (h) empty, which handles transitions on input ε.

The different modules in T_ε' are nondeterministically chosen to be run. In case the running of a module would cause a problem, this is prevented by appropriate flags in the global state on which no transitions are enabled in that particular module. We explain the working of $\mathcal{V}'$ in detail below.
Initialization: $\mathcal{V}$ initially moves to a state of the form $\$_1^m \dagger_1 (0^{M+2m+2}\dagger_1)^N$ where $\#_1, \$_1, \dagger_1$ are symbols not present in Δ (this is feasible by Lemma 5.3). We then simulate the dsNFA $\mathcal{B}_{(\vec{a}_0,\gamma_0)}$ in the first segment and the state is of the form $\$_1^m \#_1 \vec{a}_0 \gamma_0 0^m 1 0^M \dagger_1 (0^{M+2m+2}\dagger_1)^{N-1}$. We explain the contents of a segment next.

The first $2m+1$ letters in a segment are used to store context switch information: m letters for the global variables followed by the top of stack symbol, followed by m more letters. The global state of the thread can be inferred from this initial segment. The $(2m+2)$nd letter is a 1 or a 0 based on whether the segment contains the active thread or not. The final M many symbols are used to store the state of the dsNFA. Immediately after initialization, only the first segment contains a thread, which is active and starts running. Thus the separator symbol before the first segment at this point of time is #. In contrast, the other (unoccupied) segments are preceded by a †. These operations are handled by module init from (3a).
Spawning: The spawning of a new thread γ_i by a thread of type $(\vec{a}, \gamma)$ corresponds to a run of a transducer $T_{\gamma_i,\gamma}$ in a task-dsNFA and is simulated by T_i' in $\mathcal{V}$. Steps in the DCBP corresponding where the active thread does not spawn anything are handled by module empty from (3h). Having spawned some number of threads, the active thread is switched out for another.
Context Switching: In order to understand the simulation of the context switch behaviour, recall that the dsNFA $\mathcal{B}_{\vec{a},\gamma}$ which represents the succinct alphabet preserving downclosure has input alphabet $\Sigma = \Gamma \cup \tilde{\Gamma} \cup \{0,1,\perp\}$. Of these, Γ is used to indicate spawned tasks, the initial state is indicated by a string $\vec{a}_0 \in \{0,1\}^m$, context switches are indicated by strings of the form $\vec{a}_2\tilde{\gamma}_2\vec{a}_3$ where $\vec{a}_2, \vec{a}_3 \in \{0,1\}^m$ and $\tilde{\gamma} \in \tilde{\Gamma}$; and termination is indicated by strings of the form $\vec{a}_2\perp$.

The module inter in (3c) contains the transducers $T_{\sigma,\gamma}$ for $\sigma \in \tilde{\Gamma} \cup \{0,1\}$, for each $\gamma \in \Gamma$. In order to simulate a context switch, T_ε' can nondeterministically choose to run the module inter. Suppose the context switch is represented by the string $\vec{a}_2\tilde{\gamma}_2\vec{a}_3$. Since this string is output by the transducers,

it can be copied onto the first $2m+1$ cells of the segment s containing the active thread. On running inter, the global state is stored by rewriting $\$_1^m$ to $\vec{a}_2$, indicating that the DCBP is moving to a schedule point. The segment of the active thread has its $(2m+2)$nd symbol rewritten to 0. At this point, all segments have 0 as their $(2m+2)$nd symbol, indicating that there is no active thread.

Resumption of a different thread at the schedule point is executed as follows by the module resump in (3d). Two options are available at this point. The first option is to wake up some inactive thread. The state $\vec{a}_2$ stored in the first m cells is used to wake up a thread in segment s' by comparing symbol by symbol the global state stored in cells $m+2$ to $2m+2$ of s'. Segment s' is activated by writing 1 on the $(2m+2)$nd symbol in the segment and the first m cells are again replaced with $\$_1^m$.

The second option is to invoke a $T'_{\tilde{i}}$ transducer, starting a new task in one of the unoccupied segments s. The transducer rewrites the separator symbol in front of the segment s to $\tilde{\dagger}$ and writes $\tilde{\gamma}_i$ on the $(m+1)$st and 1 on the $(2m+2)$nd cells of the segment. The start module in (3b) then copies the global state $\vec{a}_2$ present on the first m cells of the tape onto the first m cells of segment s and writes the corresponding initial prefix $w_0(\vec{a}_2)$ starting from the $(2m+3)$rd cell. At the end of this operation, the separating symbol $\tilde{\dagger}$ in front of segment s is rewritten to $\#_1$, with the active thread contained in segment s at this point.

Termination: The termination of a thread is handled by module term in (3e), which contains transducers $T_{\sigma,\gamma}$ for $\sigma \in \{0, 1, \bot\}$. Whenever a string $\vec{a}_3\bot$ is output by the transducers, this string is copied onto the first $m+1$ cells of the segment containing the active thread. This signals that the task-dsNFA occupying this segment has terminated and now the clean module in (3f) is the only module allowed to run. This clean module replaces any segment containing $\bot$ with a string of 0s and also changes the symbol prior to the segment from $\#_1$ to $\dagger_1$ to indicate that the segment is now unoccupied.

Detecting $\vec{a}_f$ reachability of the DCBP**:** We note that we can modify the DCBP so that if there is a run reaching $\vec{a}_f$, then there is one where $\vec{a}_f$ occurs at a schedule point. To this end we add additional resumption rules whereby a context switch is allowed with any top of stack $\gamma \in \Gamma$ when the global state is $\vec{a}_f$. Thus it suffices to check if $\vec{a}_f$ occurs in the first m cells, at which point $\mathcal{V}$ moves to its final state. This is implemented by the module final in (3g). ∎

We can now prove Theorem 2.1. Given an input DCBP $\mathcal{D}$, its global state $\vec{a}$ and numbers K, N in binary, we construct task-dsNFAs $\mathcal{B}_{\vec{a}',\gamma}$ to represent the alphabet preserving downclosure $\mathsf{L}(\mathcal{P}_{\vec{a}',\gamma}(\mathcal{D}))\downarrow_{\Theta,m+(2m+1)K+(m+1)}$ of each task-PDA $C_{\vec{a}',\gamma}$ using Corollary 4.4. Note that the parameter $m+(2m+1)K+(m+1)$ occurs because in the language of an initialized task PDA, the initial assignment to the global variables is represented by a string of length m, each context switch is represented by a string of length $2m+1$, and the termination is represented by a string of length $m+1$. There are only polynomially many transducers corresponding to different dsNFA which need to be stored, one set for each γ, since the assignment $\vec{a}$ to the global Boolean variables only affects the initial prefix. These transducers are then used in Theorem 5.2 to reduce the thread-pooled safety problem to coverability of TCVASS. Finally, we use Theorem 5.1 which gives us an EXPSPACE procedure for coverability of TCVASS.

6 CONTEXT-BOUNDED REACHABILITY IS 3EXPSPACE-HARD

To put our results on thread-pooling in perspective, we show that we obtain 3EXPSPACE-completeness if we remove all restrictions on the thread-pool, as formalized by Theorem 2.2. We already mentioned that the upper bound of this theorem follows directly from Atig et al. [2009], and that for the lower bound we extend the 2EXPSPACE-hardness result of Baumann et al. [2020] by using succinctly encoded models in the intermediate proof steps.

The previous hardness result uses a more restrictive model called DCPS. Compared to DCBP it has no local Boolean variables, and uses a set of global states instead of global Boolean variables. This can be easily simulated by DCBP with (i) an empty set of local variables and (ii) using global variables to store the global state. Then we ensure that there is always exactly one global variable valued 1, corresponding to the current global state. Due to this simulation it suffices to only consider the model of DCPS for the lower bound, which we will do for the remainder of this section.

To also give a bit more detail on the upper bound, going from DCPS with unary context-switch bound k to DCBP with binary k incurs two exponential blow-ups, one for Boolean variables instead of states and one for the encoding of k. However, both these blow-ups occur multiplicatively with respect to one another, meaning the combined blow-up is still only singly exponential. Therefore the 2EXPSPACE-procedure by Atig et al. [2009] becomes a 3EXPSPACE-procedure in our setting.

We proceed by proving the 3EXPSPACE lower bound. The full proof is technical, and provided in the full version. We sketch the main challenges in this section.

Previous Result. Let us begin by recalling some details in the proof of the 2EXPSPACE lower bound of Baumann et al. [2020]. The hardness proof goes through a series of reductions; the main step is a reduction from the coverability problem for *transducer-defined Petri nets* (TDPN) to context-bounded reachability in DCPS for a unary context bound.

Recall that Petri nets are essentially a variant of VASS with just one state. TDPN are succinct representations of Petri nets. Similarly to TCVASS, the (exponentially many) counters of a TDPN are encoded by strings of polynomial length over an alphabet, and transitions are described by three transducers that encode the transitions between counters. Three transducers are needed because TDPN only consider three types of transitions: *Fork* transitions that subtract 1 from one counter and add 1 to each of two counters, *join* transitions that subtract 1 from two counters and add 1 to one counter, and *move* transitions that add 1 to and subtract 1 from a single counter each. It is known that any Petri net can be brought into a "normal form" in polynomial time, where each transition has one of these forms. Like for VASS, a configuration of a Petri net is a map from its counters to $\mathbb{N}$. In lieu of more than one state, the coverability problem for Petri nets takes a counter as input and asks whether a configuration with a nonzero value for that counter is reachable. Petri nets have an EXPSPACE-complete coverability problem. Baumann et al. [2020] showed 2EXPSPACE-hardness for the coverability problem for TDPN.

The succinctness in TDPN is different from that in TCVASS: in a TCVASS, the control state is encoded succinctly but the dimension (the number of counters) is maintained explicitly. In contrast, a TDPN has a dimension that is exponentially larger than the description.

The main step of the lower bound reduces the coverability problem for TDPN to context-bounded reachability for DCPS by representing the explicit Petri net configurations as tasks. (Since there is no thread pooling, each task runs in its own thread, simultaneously with all other spawned and partially executed tasks.) Each counter p of the TDPN, which is encoded as a string of length n, is assigned a number $u(p)$ by such a configuration. To encode this configuration, the DCPS maintains exactly $u(p)$ many tasks that each have p as their stack content. Transitions are then simulated by (i) emptying stacks of tasks whose corresponding counters were decremented, and (ii) spawning new tasks whose stacks are built up with corresponding counters that are incremented. During this, n many new tasks are spawned to hold the information of the string encoding the corresponding counter. These $2n$ or $3n$ tasks in the task buffer are then used to verify that the corresponding transition existed, with the three transducers being kept as part of the global states of the DCPS. Coverability for Petri nets then corresponds to reachability of a DCPS configuration that includes a task corresponding to a specific counter p. The DCPS can just check for stack

content p using n global states, and then move to a specific global state g, completing the reduction to context-bounded reachability.

Challenges. To lift the previous result to 3EXPSPACE we introduce the model of *succinct* TDPN (sTDPN), which use exponentially long strings for their encoding, as opposed to polynomially long ones. In our reduction to DCPS, we represent a configuration of (the underlying Petri net of) a sTDPN in the same way as before, the only difference being that stack contents are now of exponential length. This causes some challenges not present for the previous proof: Firstly, we cannot store the information of a single string of length m via m tasks in the task buffer. To be able to unambiguously reconstruct the string, each such task would need to remember the position of its letter, 1 to m. Since m is now exponential, and each such task just carries a single symbol, this would require an exponentially large alphabet. Secondly, we now also need to create a stack content of exponential length to initialize the whole simulation. For polynomial length n this could just be done using n many global states. Finally, the check for a specific counter p at the end is also more problematic, since it again needs to check an exponentially long string letter-by-letter.

However, since K is provided in binary, in contrast to the previous proof, we can now make exponentially many context switches per task. This allows our proof to overcome the above challenges as follows.

Verifying Transitions. Since we have exponentially many context switches, our simulation of a single transition can switch back and forth between the two or three tasks involved in a single transition of the sTDPN, removing or adding one stack symbol each time. For each pair or triple of such symbols we advance the state of the transducer to verify the existence of this transition. Because the verification now happens at the same time as the simulation, we do not need to store additional information for it in the task buffer. The problem here is that the stacks that are built-up in this way are in *reverse* order compared to the stacks that were emptied. This presents another challenge: in order to continue the simulation, we have to reverse an exponentially long stack.

Reversing Stacks. Another use case of exponentially many context switches is the transfer of an exponential length stack content from one task to another. This is done iteratively by popping a symbol from the first task, remembering it in the global state, switching to the second task, and then pushing the symbol before switching back. Doing so reverses the order of symbols on the stack, which is exactly what we need to remedy the issue mentioned above. If our stacks have exponential length m then this process requires m iterations and therefore m context switches. With the simulation of a transition acting on the task beforehand it results in $2m$ context switches in total per task, which is still exponential in the binary context bound K, and therefore can be performed in the reduction.

Initialization and Finalization. Starting the simulation was "easy" for TDPN: we could explicitly add the first stack of polynomial size using the global state. This is no longer possible as the stack is now exponential.

To create the exact exponentially long encoding of a specific counter p_0 at the start of the simulation, we do not encode it in the global states. Instead we simulate a context free grammar on the initial thread's stack. One can construct such a grammar of polynomial size that produces just a single exponentially long word, which in this case is p_0 (the grammar is a straight line program that performs the "iterated doubling" trick to accept exactly one word of exponential size). Due to the grammar's size, we only need to add polynomially many stack symbols to facilitate its simulation.

For the final part of the simulation, where we check for a specific counter p, we could also try and utilize a context free grammar. However, due to some technical details in our definitions for sTDPN, this is not even necessary: We only allow a certain shape for the p used as input to coverability,

and this shape can be easily described using a regular expression. Therefore we can check for the correct shape by using a finite automaton encoded in the global states of the DCPS.

7 DISCUSSION

We have characterized the complexity of safety verification in the presence of thread pooling. Surprisingly, thread pooling *reduces* the complexity of verification by a double exponential amount, even when all parameters are in binary. Along the way, we have introduced succinct representations and manipulations of succinct machines that may be of independent interest.

While we have focused on safety verification, we can consider liveness properties as well. The *thread-pooled context-bounded termination* problem asks, given a DCBP $\mathcal{D}$ and two numbers K and N in binary, does every N-thread-pooled, K-context switch bounded run terminate in a finite number of steps? Our constructions imply that checking termination is EXPSPACE-complete. The upper bound proceeds as with reachability, but we check termination rather than coverability for the constructed VASS. The lower bound follows from a "standard trick" of reducing reachability of a global state to non-termination: the program initially guesses the number of steps to reach a global state (by spawning that many threads) and entering an infinite loop iff the global state is reached. When $N = \infty$, the problem becomes 3EXPSPACE-complete, following the same ideas as for safety verification.

An infinite run is *fair* if, intuitively, the scheduler eventually picks an instance of every spawned task and any thread that can be executed is eventually executed. The thread-pooled, context-bounded *fair termination* problem asks if there is a fair non-terminating run. Baumann et al. [2021] study the problem in the case of explicitly specified states (no Boolean variables) without thread pooling and reduce it to VASS reachability. Here, the reduction can be performed in elementary time. By turning Boolean variable assignments into explicit states and keeping a counter of threads in the thread pool (in the global state), one can easily turn a DCBP into an exponentially large DCPS. Therefore, fair termination of DCBP also admits an elementary-time reduction to VASS reachability. On the other hand, VASS reachability can be reduced in polynomial time to fair non-termination of the special case ($K = 0$, $N = 1$) of asynchronous programs [Ganty and Majumdar 2012]. Together with complexity results on VASS-reachability [Czerwiński and Orlikowski 2022; Leroux 2022; Leroux and Schmitz 2019], we conclude the problem is Ackermann-complete with or without thread pooling.

In conclusion, we find it surprising that thread pooling results in a significant reduction in the theoretical complexity of safety verification. Whether this observation has practical implications remains to be explored.

ACKNOWLEDGMENTS

This research was sponsored in part by the Deutsche Forschungsgemeinschaft project 389792660 TRR 248–CPEC and by the European Research Council under the Grant Agreement 610150 (http://www.impact-erc.eu/) (ERC Synergy Grant ImPACT).

REFERENCES

Mohamed Faouzi Atig, Ahmed Bouajjani, and Shaz Qadeer. 2009. Context-Bounded Analysis for Concurrent Programs with Dynamic Creation of Threads. In *Proceedings of TACAS 2009*. 107–123. https://doi.org/10.1007/978-3-642-00768-2_11

Georg Bachmeier, Michael Luttenberger, and Maximilian Schlund. 2015. Finite Automata for the Sub- and Superword Closure of CFLs: Descriptional and Computational Complexity. In *9th International Conference on Language and Automata Theory and Applications, LATA 2015, Nice, France, March 2-6, 2015, Proceedings*. Springer, 473–485. https://doi.org/10.1007/978-3-319-15579-1_37

Thomas Ball and Sriram K. Rajamani. 2000. Bebop: A Symbolic Model Checker for Boolean Programs. In *SPIN Model Checking and Software Verification, 7th International SPIN Workshop, Stanford, CA, USA, August 30 - September 1, 2000,*

Proceedings (Lecture Notes in Computer Science, Vol. 1885), Klaus Havelund, John Penix, and Willem Visser (Eds.). Springer, 113–130. https://doi.org/10.1007/10722468_7

Thomas Ball and Sriram K. Rajamani. 2001. The SLAM Toolkit. In *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2102)*, Gérard Berry, Hubert Comon, and Alain Finkel (Eds.). Springer, 260–264. https://doi.org/10.1007/3-540-44585-4_25

Pascal Baumann, Rupak Majumdar, Ramanathan S. Thinniyam, and Georg Zetzsche. 2020. The Complexity of Bounded Context Switching with Dynamic Thread Creation. In *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, July 8-11, 2020, Saarbrücken, Germany (Virtual Conference) (LIPIcs, Vol. 168)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 111:1–111:16. https://doi.org/10.4230/LIPIcs.ICALP.2020.111

Pascal Baumann, Rupak Majumdar, Ramanathan S. Thinniyam, and Georg Zetzsche. 2021. Context-Bounded Verification of Liveness Properties for Multithreaded Shared-Memory Programs. *Proceedings of the ACM on Programming Languages (PACMPL)* 5, POPL, Article 44 (Jan. 2021), 31 pages. https://doi.org/10.1145/3434325

Peter Chini, Jonathan Kolberg, Andreas Krebs, Roland Meyer, and Prakash Saivasan. 2017. On the Complexity of Bounded Context Switching. In *25th Annual European Symposium on Algorithms, ESA 2017, September 4-6, 2017, Vienna, Austria (LIPIcs, Vol. 87)*, Kirk Pruhs and Christian Sohler (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 27:1–27:15. https://doi.org/10.4230/LIPIcs.ESA.2017.27

Bruno Courcelle. 1991. On constructing obstruction sets of words. *Bulletin of the EATCS* 44 (1991), 178–186.

Wojciech Czerwiński and Lukasz Orlikowski. 2022. Reachability in Vector Addition Systems is Ackermann-complete. In *Proceedings of the 62nd Annual IEEE Symposium on Foundations of Computer Science, FOCS 2021, Denver, CO, USA, February 7-10, 2022*. to appear.

Javier Esparza. 1998. Decidability and Complexity of Petri Net Problems – an Introduction. In *Lectures on Petri Nets I: Basic Models. Advances in Petri Nets (Lecture Notes in Computer Science, 1491)*, G. Rozenberg and W. Reisig (Eds.). 374–428. https://doi.org/10.1007/3-540-65306-6_20

Pierre Ganty and Rupak Majumdar. 2012. Algorithmic verification of asynchronous programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 34, 1 (2012), 6. https://doi.org/10.1145/2160910.2160915

Seymour Ginsburg. 1966. *The mathematical theory of context free languages.* McGraw Hill.

Patrice Godefroid and Mihalis Yannakakis. 2013. Analysis of Boolean Programs. In *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7795)*, Nir Piterman and Scott A. Smolka (Eds.). Springer, 214–229. https://doi.org/10.1007/978-3-642-36742-7_16

H. Gruber, M. Holzer, and M. Kutrib. 2009. More on the Size of Higman-Haines Sets: Effective Constructions. *Fundam. Inf.* 91(1) (2009), 105–121.

Leonard H Haines. 1969. On free monoids partially ordered by embedding. *Journal of Combinatorial Theory* 6, 1 (1969), 94–98.

Graham Higman. 1952. Ordering by divisibility in abstract algebras. *Proc. London Math. Soc.* (3) 2 (1952), 326–336.

Ranjit Jhala and Rupak Majumdar. 2007. Interprocedural Analysis of Asynchronous Programs. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*. ACM, 339–350. https://doi.org/10.1145/1190216.1190266

Salvatore La Torre, P. Madhusudan, and Gennaro Parlato. 2009. Reducing Context-Bounded Concurrent Reachability to Sequential Reachability. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5643)*, Ahmed Bouajjani and Oded Maler (Eds.). Springer, 477–492. https://doi.org/10.1007/978-3-642-02658-4_36

Salvatore La Torre, Parthasarathy Madhusudan, and Gennaro Parlato. 2010. The Language Theory of Bounded Context-Switching. In *LATIN 2010: Theoretical Informatics, 9th Latin American Symposium, Oaxaca, Mexico, April 19-23, 2010, Proceedings (Lecture Notes in Computer Science, Vol. 6034)*. Springer, 96–107. https://doi.org/10.1007/978-3-642-12200-2_10

Akash Lal and Thomas W. Reps. 2009. Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods in System Design* 35, 1 (2009), 73–97. https://doi.org/10.1007/s10703-009-0078-9

Jérôme Leroux. 2022. The Reachability Problem for Petri Nets is Not Primitive Recursive. In *Proceedings of the 62nd Annual IEEE Symposium on Foundations of Computer Science, FOCS 2021, Denver, CO, USA, February 7-10, 2022*. to appear.

Jérôme Leroux and Sylvain Schmitz. 2019. Reachability in Vector Addition Systems is Primitive-Recursive in Fixed Dimension. In *Proceedings of the 34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, Canada, June 24-27, 2019*. 1–13. https://doi.org/10.1109/LICS.2019.8785796

Jiaxin Li, Yuxi Chen, Haopeng Liu, Shan Lu, Yiming Zhang, Haryadi S. Gunawi, Xiaohui Gu, Xicheng Lu, and Dongsheng Li. 2018. Pcatch: automatically detecting performance cascading bugs in cloud systems. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, Rui Oliveira, Pascal Felber, and Y. Charlie Hu (Eds.). ACM, 7:1–7:14. https://doi.org/10.1145/3190508.3190552

Richard Lipton. 1976. The reachability problem is exponential-space hard. *Yale University, Department of Computer Science, Report* 62 (1976).

Rupak Majumdar, Ramanathan S. Thinniyam, and Georg Zetzsche. 2021. General Decidability Results for Asynchronous Shared-Memory Programs: Higher-Order and Beyond. In *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 12651)*, Jan Friso Groote and Kim Guldstrand Larsen (Eds.). Springer, 449–467. https://doi.org/10.1007/978-3-030-72016-2_24

Roland Meyer, Sebastian Muskalla, and Georg Zetzsche. 2018. Bounded Context Switching for Valence Systems. In *29th International Conference on Concurrency Theory, CONCUR 2018, September 4-7, 2018, Beijing, China (LIPIcs, Vol. 118)*, Sven Schewe and Lijun Zhang (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 12:1–12:18. https://doi.org/10.4230/LIPIcs.CONCUR.2018.12

Madanlal Musuvathi and Shaz Qadeer. 2007. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, PLDI 2007, San Diego, CA, USA, June 10-13, 2007*. ACM, 446–455. https://doi.org/10.1145/1250734.1250785

Christos H. Papadimitriou and Mihalis Yannakakis. 1986. A note on succinct representations of graphs. *Information and Control* 71, 3 (1986), 181–185. https://doi.org/10.1016/S0019-9958(86)80009-2

Shaz Qadeer and Jakob Rehof. 2005. Context-Bounded Model Checking of Concurrent Software. In *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3440)*. Springer, 93–107. https://doi.org/10.1007/978-3-540-31980-1_7

Charles Rackoff. 1978. The covering and boundedness problems for vector addition systems. *Theoretical Computer Science* 6, 2 (1978), 223–231.

Ganesan Ramalingam. 2000. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM TOPLAS* 22(2) (2000), 416–430. https://doi.org/10.1145/349214.349241

Louis E. Rosier and Hsu-Chun Yen. 1986. A Multiparameter Analysis of the Boundedness Problem for Vector Addition Systems. *J. Comput. System Sci.* 32, 1 (1986), 105–135. https://doi.org/10.1016/0022-0000(86)90006-1

Koushik Sen and Mahesh Viswanathan. 2006. Model Checking Multithreaded Programs with Asynchronous Atomic Methods. In *28th International Conference on Computer Aided Verification, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings (LNCS, Vol. 4144)*. Springer, 300–314. https://doi.org/10.1007/11817963_29

Jan van Leeuwen. 1978. Effective constructions in well-partially-ordered free monoids. *Discrete Mathematics* 21, 3 (1978), 237–252. https://doi.org/10.1016/0012-365X(78)90156-5