BFS-Based Model Checking of Linear-Time Properties with an Application on GPUs

Anton Wijs(✉)

Eindhoven University of Technology, Eindhoven, The Netherlands
A.J.Wijs@tue.nl

Abstract. Efficient algorithms have been developed to model check liveness properties, such as the well-known Nested Depth-First Search, which uses a depth-first search (DFS) strategy. However, in some settings, DFS is not a suitable option. For instance, when considering distributed model checking on a cluster, or many-core model checking using a Graphics Processing Unit (GPU), Breadth-First Search (BFS) is a more natural choice, at least for basic reachability analysis. Liveness property checking, however, requires the detection of (accepting) cycles, and BFS is not very suitable to detect these on-the-fly. In this paper, we consider how a model checker that completely runs on a GPU can be extended to efficiently verify whether finite-state concurrent systems satisfy liveness properties. We exploit the fact that the state space is the product of the behaviour of several parallel automata. The result of this work is the very first GPU-based model checker that can check liveness properties.

1 Introduction

Model checking [2] is a formal verification technique to ensure that a model satisfies desired functional properties. Some of these properties may address infinite system behaviour. Such properties are called *liveness* properties; they express that some desired behaviour eventually happens [1].

In finite-state systems, infinite behaviour is only possible if some of the states are visited infinitely often. Therefore, in the state space of such a system, infinite behaviour is represented by a cycle and a path from the initial state leading to it. A counter-example to a liveness property is also some infinite behaviour through the product of the system behaviour and an automaton accepting any infinite behaviour that violates the property [2]. Linear-time properties, for instance expressed in the LTL temporal logic, can be represented by such automata, for instance Büchi and Rabin automata.

For explicit-state model checking, there are efficient algorithms to find counter-examples to liveness properties, e.g. [10,14,17,29,30]. All of these perform a search through the state space using a Depth-First Search (DFS) strategy. However, in some settings, Breadth-First Search (BFS) is a more natural choice

A. Wijs—We gratefully acknowledge the support of NVIDIA Corporation with the donation of the GeForce Titan X used for this research.

© Springer International Publishing Switzerland 2016
S. Chaudhuri and A. Farzan (Eds.): CAV 2016, Part II, LNCS 9780, pp. 472–493, 2016.
DOI: 10.1007/978-3-319-41540-6_26

for graph traversals than DFS. For instance, when model checking is done by several machines collaboratively in a network [5,7,11], and when General Purpose Graphics Processing Units (GPGPUs or GPUs) are employed for model checking. In the former case, information obtained through DFS traversals cannot efficiently by synchronised between workers. In the latter case, the availability of thousands of threads, and the requirement to occupy these to fully harness the computing power of the GPU, is at odds with a DFS-based search, in which a stack is used and the focus is always on the state at the top of the stack. One might consider running multiple randomised DFSs in parallel [12,23], but maintaining thousands of stacks would require too much overhead.

GPUs are being used to dramatically speed up computations. For explicit-state model checking, GPUs are used for on-the-fly state space exploration and safety property checking [6,35,36,39], offline property checking [3,8,9,34], counterexample generation [38], and state space decomposition and minimisation [33,37].

The next challenge in GPU model checking is the on-the-fly checking of general linear-time properties. In this paper, we consider doing this using a BFS-based approach. The best known way to find cycles using a BFS-like search is by *topological sorting* of the states [19], but it does not work on-the-fly, i.e. while discovering the state space. A more suitable algorithm is a heuristic search called *piggybacking* [13,16], in which a state that may be in a cycle of a counterexample, is carried (piggybacked) along the search. If at any point, a state is visited which is at that time also piggybacked, then a cycle is found. However, the algorithm is not complete; in many situations, it may fail to detect a cycle.

Contributions. Firstly, we define, to the best of our knowledge, for the first time a variant of piggybacking using *Rabin automata* to express properties. Secondly, we propose several new algorithms that can efficiently analyse the input model before the checking is started, to extract structural information that tends to make the piggybacking algorithm more effective. Thirdly, in earlier work [13], a post-processing phase was proposed to further analyse particular states that can be identified as 'promising' during piggybacking. This phase makes piggybacking complete for *bounded suffix model checking*, in which cycle detection is limited to cycles of a predetermined number of transitions. In [13], a depth-bounded DFS was initiated from each of those promising states to search for a cycle containing them. In this paper, we show that with the extracted structural information, the number of promising states can be reduced, thereby potentially speeding up the post-processing phase. Fourthly, we discuss how piggybacking can be implemented in a GPU model checking approach, focussing primarily on the data structures. Finally, we validate the effectiveness of our approach as a whole, and the application of the different algorithms in particular, on an implementation in the GPU model checker GPUExplore [35,36].

It should be stressed that, even though we focus on using GPUs, our proposed enhancements of piggybacking may be of use in other contexts as well. For instance, in [25], several heuristics are used to incrementally verify liveness properties symbolically using saturation. Our proposed heuristics seem to be

more refined than the ones they use; for instance, they consider every visiting of a state in a non-trivial strongly connected component to be an event that may close a cycle in the state space, whereas we can distinguish situations where this cannot be the case from cases where it can.

Piggybacking was first proposed in [16] and extended in [13], and a similar approach is used as an on-the-fly heuristic in the OTF-OWCTY algorithm [4]. Various different strategies to select and remove piggybacking states have been proposed, including piggybacking not one, but multiple states, to improve the algorithm. However, these variations tend to not fundamentally improve the original algorithm. We are the first to try to exploit the fact that the state space is the combination of behaviour of several interacting processes in the input model. One of the new algorithms, to identify so-called essential states, is inspired by [22]. They statically analyse cycles in processes to improve partial order reduction, and they consider state-based models. as opposed to our algorithm, which works for action-based models.

The structure of the paper is as follows. In Sect. 2, we discuss the basic notions. Section 3 presents the piggybacking algorithm and various algorithms to statically analyse input models. How these algorithms affect GPU model checking is explained in Sect. 4. Experimental results are given in Sect. 5, and finally, our conclusions are presented in Sect. 6.

2 Preliminaries

In this section, we discuss the basic notions involved to understand the problem, namely Lts, network of Ltss, and an action-based version of Rabin automata.

Labelled Transition Systems. We use Labelled Transition Systems (Ltss) to represent the semantics of finite-state systems. They are action-based descriptions, indicating how a system can change state by performing particular actions.

Definition 1 (Labelled Transition System). *An* Lts $\mathcal{G}$ *is a tuple* $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \underline{s} \rangle$, *with*

- $\mathcal{S}$ *a finite set of states;*
- $\mathcal{A}$ *a set of action labels;*
- $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ *a transition relation;*
- $\underline{s} \in \mathcal{S}$ *the initial state.*

Actions in $\mathcal{A}$ are denoted by a, b, c, etc. We use $s_1 \xrightarrow{a} s_2$ to denote $\langle s_1, a, s_2 \rangle \in \mathcal{T}$. If $s_1 \xrightarrow{a} s_2$, this means that an action a can be performed in state s_1, leading to state s_2; we call s_2 a *successor* of s_1. We refer with $\rightarrow^*$ and $\rightarrow^+$ to the reflexive, transitive closure, and the transitive closure of $\rightarrow$, respectively. We call $s_1 \rightarrow^* s_2$ and $s_1 \rightarrow^+ s_2$ *paths* through $\mathcal{G}$, and for a path π, we refer with $S(\pi)$ to the set of states that are part of π. A *cycle* is a path consisting of at least one transition, from a state s_1 to itself, i.e. $s_1 \rightarrow^+ s_1$. Finally, $\mathit{inf}(\pi)$ is the set of states in $S(\pi)$ that are part of a cycle in π. In particular, if π is a cycle, we have $S(\pi) = \mathit{inf}(\pi)$. In finite-state Ltss, a path involving a cycle is called an *infinite* path.

LTS *Networks.* We use LTS networks (Definition 2) to describe concurrent systems. They consist of a finite number of concurrent process LTSs and a set of synchronisation rules that define the possible interaction between the processes. We write $1..n$ for the set of integers ranging from 1 to n. A vector $\bar{v}$ of size n contains n elements indexed from 1 to n. For all $i \in 1..n$, $\bar{v}_i$ represents the ith element of vector $\bar{v}$.

Definition 2 (LTS network). *An* LTS *network* $\mathcal{N}$ *of size* n *is a pair* $\langle \Pi, \mathcal{V} \rangle$, *where*

- Π *is a vector of* n *concurrent process* LTSs. *For each* $i \in 1..n$, *we write* $\Pi_i = \langle \mathcal{S}_i, \mathcal{A}_i, \mathcal{T}_i, \underline{s}_i \rangle$.
- $\mathcal{V}$ *is a finite set of synchronisation rules. A* synchronisation rule *is a tuple* $\langle a, T \rangle$, *where* a *is an action label and* $T \subseteq 1..n$.

The synchronisation rules define how the processes can synchronise with each other. These rules allow m among n synchronisation between transitions of the same label. The fact that they must have the same label is a restriction of our definition of LTS networks, by which it differs from the one in [24]. Our rules can, for instance, express that a-transitions of different processes need to synchronise, and not that a- and b-transitions must synchronise with each other, and produce a transition with some other label, say c. The reason for having the restriction is due to our desire to compactly represent synchronisation rules in the GPU model checker [35,36]. It does not restrict our ability to specify concurrent system behaviour, since any network with rules that do not adhere to this restriction can be rewritten to a network without such rules. It does remove the ability to rename transition labels, but on the other hand, this is not a limitation for the current application, in which renaming is never applied.

The explicit behaviour of an LTS network is defined by its *system* LTS (Definition 3).

Definition 3 (System LTS). *Given an* LTS *network* $\mathcal{N} = \langle \Pi, \mathcal{V} \rangle$, *its* system LTS *is defined by* $\mathcal{N}_\mathcal{G} = \langle \mathcal{S}_\mathcal{N}, \mathcal{A}_\mathcal{N}, \mathcal{T}_\mathcal{N}, \underline{s}_\mathcal{N} \rangle$, *with*

- $\mathcal{S}_\mathcal{N} = \mathcal{S}_1 \times \cdots \times \mathcal{S}_n$;
- $\mathcal{A}_\mathcal{N} = \bigcup_{i \in 1..n} \mathcal{A}_i$;
- $\underline{s}_\mathcal{N} = \langle \underline{s}_1, \ldots, \underline{s}_n \rangle$, *and*
- $\mathcal{T}_\mathcal{N}$ *is the smallest relation satisfying:*

$$\forall \bar{s} \in \mathcal{S}_\mathcal{N}, \langle a, T \rangle \in \mathcal{V}.(\forall i \in 1..n. \begin{pmatrix} (i \notin T \wedge \bar{s}_i = \bar{s'}_i) \\ \vee (i \in T \wedge \bar{s}_i \xrightarrow{a} \bar{s'}_i) \end{pmatrix}) \implies \bar{s} \xrightarrow{a} \bar{s'}$$

The system LTS is obtained by combining the processes in Π according to the synchronisation rules in $\mathcal{V}$. In the following, we assume that the so-called *independent* transitions of a process, i.e. those that do not require synchronisation with other transitions, are always enabled. In other words, for each process Π_i ($i \in 1..n$) and independent action $a \in \mathcal{A}_i$, we assume that $\langle a, \{i\} \rangle \in \mathcal{V}$.

Rabin Automata. Model checking linear-time functional properties involves checking whether a system satisfies a property φ, written in some linear-time temporal logic. The property can be a liveness property addressing infinite system behaviour ("something good eventually happens"). Since we use the action-based LTS network formalism to express system behaviour, we also require an action-based way to express our properties (which can, for instance, be done using action-based LTL). For this, we extend the definition of LTS to obtain an action-based Rabin automaton [28].

Definition 4 (Rabin automaton). *A* Rabin automaton *(RA)* $\mathcal{R}$ *is a tuple* $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \underline{s}, \mathcal{F} \rangle$*, with*

- $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \underline{s} \rangle$ *an* LTS*;*
- $\mathcal{F}$ *a set of* k *pairs of state sets* $\{(L_i, K_i) \mid 0 \leq i < k\}$*, with* $L_i, K_i \subseteq \mathcal{S}$*.*

An RA has an *acceptance condition*: an infinite path π is *accepted* iff there exists a pair $(L_i, K_i) \in \mathcal{F}$ such that $\mathit{inf}(\pi) \cap L_i = \emptyset$ and $\mathit{inf}(\pi) \cap K_i \neq \emptyset$ [2,21]. RAs are as expressive as ω-regular languages (and many linear-time properties are ω-regular). In fact, contrary to Büchi automata, deterministic versions of RAs already have the full power of ω-regular languages [15]. A deterministic automaton defines for each state and action label at most one successor state.

Verifying Linear-Time Properties. When checking functional properties, one must solve the emptiness problem for the product of $\mathcal{N}_{\mathcal{G}}$ and the property automaton $\mathcal{R}_{\neg\varphi}$, where $\mathcal{R}_{\neg\varphi}$ refers to the RA accepting all infinite paths described by the negation of φ [32]. In fact, this product is the system LTS $\mathcal{P}_{\mathcal{G}}$ of an LTS network $\mathcal{P} = \langle \Pi', \mathcal{V}' \rangle$ combining $\mathcal{N}$ and $\mathcal{R}_{\neg\varphi}$, in which for each $i \in 1..n$, $\Pi'_i = \Pi_i$, $\Pi'_{n+1} = \mathcal{R}_{\neg\varphi}$, and the synchronisation rules of $\mathcal{N}$ have been extended in $\mathcal{V}'$ to always involve synchronisation with $\mathcal{R}_{\neg\varphi}$, i.e. $\mathcal{V}' = \{\langle a, T \cup \{n+1\}\rangle \mid \langle a, T \rangle \in \mathcal{V}\}$.

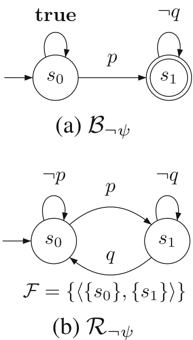

Fig. 1. Request-response automata

A counter-example for φ in $\mathcal{P}_{\mathcal{G}}$ exists iff there exists some path $\pi = \langle \underline{s}_{\mathcal{N}}, \underline{t} \rangle \rightarrow^* \langle \bar{s}, t \rangle$, with $\underline{t}$ the initial state of $\mathcal{R}_{\neg\varphi}$, and a path $\pi' = \langle \bar{s}, t \rangle \rightarrow^+ \langle \bar{s}, t \rangle$, that is accepted by $\mathcal{R}_{\neg\varphi}$. Let $C = \{t' \mid \langle \bar{s}', t' \rangle \in S(\pi')\}$, then the combination of π and π' is accepted iff there exists an $(L_i, K_i) \in \mathcal{F}$ such that $C \cap L_i = \emptyset$ and $C \cap K_i \neq \emptyset$. If this is the case, then π' is referred to as an *accepting cycle*. In other words, solving the emptiness problem boils down to checking for the reachability of accepting cycles.

Figure 1 shows a Büchi automaton $\mathcal{B}_{\neg\psi}$ (a), which accepts all paths that traverse through s_1 infinitely often, and a Rabin automaton $\mathcal{R}_{\neg\psi}$ (b). Both express the negation of a property ψ = "after p, eventually q happens".[1] This property is of a very commonly used type, namely *request-response*. For our application, Rabin automata are often more suitable than Büchi automata. In GPU model checking (Sect. 4), the outgoing transitions of each state vector are constructed using multiple threads. In general, for this

[1] Note that this is an action-based property referring to the transition-labels, as opposed to a state-based property referring to state predicates.

construction, the more independent transitions are present in the processes, the more potential there is to explore them in parallel, which speeds up the successor construction. When using non-deterministic Büchi automata, we observe that the potential to avoid synchronisation is often smaller than when using (deterministic) Rabin automata. For instance, in $\mathcal{R}_{\neg\psi}$, we can ignore the self-loops, and combine $\mathcal{R}_{\neg\psi}$ with $\mathcal{N}_{\mathcal{G}}$ in such a way that p and q-transitions of $\mathcal{N}_{\mathcal{G}}$ need to synchronise with p and q-transitions of $\mathcal{R}_{\neg\psi}$, but other transitions do not, thereby avoiding many synchronisations. In $\mathcal{B}_{\neg\psi}$, however, we have a self-loop $\neg q$ at state s_1, which cannot be ignored in the same way, since we depend on q-transitions in $\mathcal{N}_{\mathcal{G}}$ being blocked whenever $\mathcal{R}_{\neg\psi}$ is in state s_1 in $\mathcal{P}_{\mathcal{G}}$. This could be solved by extending $\mathcal{B}_{\neg\psi}$, but an additional drawback is that unlike $\mathcal{R}_{\neg\psi}$, $\mathcal{B}_{\neg\psi}$ is non-deterministic for p at state s_0, thereby creating two branches in $\mathcal{P}_{\mathcal{G}}$ whenever a p-transition can be fired, which may increase the state space size. On the other hand, it should be noted that in general, Rabin automata tend to be larger than Büchi automata expressing the same property, but multiple techniques exist to keep the former reasonably small [20].

3 On-The-Fly BFS-Based Property Checking

3.1 A Piggyback Algorithm for Rabin Automata

It is well-known that BFS-based graph search algorithms are not as efficient in detecting cycles as DFS-based algorithms. This is because unlike in DFS, where a stack is employed that keeps track of the currently explored path, information on the individual paths in the state space is not maintained during a BFS. However, there is one exception to this, relating to the state s from which the BFS is initiated: if at any point in the BFS, s is reached again, we know that a cycle involving s exists.

Algorithm 1. Piggyback BFS

Require: network $\mathcal{P} = \langle \Pi', \mathcal{V}' \rangle$
  $\alpha \leftarrow \bullet$; $I \leftarrow \{i \mid (L_i, K_i) \in \mathcal{F} \wedge \underline{s}_{n+1} \in K_i\}$
2: **if** $I \neq \emptyset$ **then** $\alpha \leftarrow \underline{s}_{\mathcal{P}}$
  - *Open*, *Closed* $\leftarrow \{\langle \underline{s}_{\mathcal{P}}, [\alpha, I] \rangle\}$
4: **while** *Open* $\neq \emptyset$ **do**
  - $\langle \bar{s}, [\alpha, I] \rangle \leftarrow$ *Open*; *Open* $\leftarrow$ *Open* $\setminus \{\langle \bar{s}, [\alpha, I] \rangle\}$
6: **for all** $\langle a, \bar{s}' \rangle \in \textsc{out}(\bar{s})$ **do**
  if $\bar{s}' = \alpha$ **then**
8: **return false**
  $I' \leftarrow \{i \mid (L_i, K_i) \in \mathcal{F} \wedge \bar{s}'_{n+1} \notin L_i \wedge i \in I\}$
10: **if** $I' = \emptyset$ **then**
  $\alpha \leftarrow \bullet$; $I' \leftarrow \{i \mid (L_i, K_i) \in \mathcal{F} \wedge \bar{s}'_{n+1} \in K_i\}$
12: **if** $I' \neq \emptyset$ **then**
  $\alpha \leftarrow \bar{s}'$
14: **if** $\neg\exists \beta, J. \langle \bar{s}', [\beta, J] \rangle \in$ *Closed* **then**
  - *Closed* $\leftarrow$ *Closed* $\cup \{\langle \bar{s}', [\alpha, I'] \rangle\}$
16: - *Open* $\leftarrow$ *Open* $\cup \{\langle \bar{s}', [\alpha, I'] \rangle\}$
  else if $\alpha \neq \bullet$ **then**
18: **if** $\beta = \bullet$ **then**
  - *Closed* $\leftarrow$ (*Closed* $\setminus \{\langle \bar{s}', [\beta, J] \}) \cup \{\langle \bar{s}', [\alpha, I'] \rangle\}$
20: - *Open* $\leftarrow$ *Open* $\cup \{\langle \bar{s}', [\alpha, I'] \rangle\}$
  else
22: Mark blocking occurrence in $\bar{s}'$
  Post-process blocking occurrences; return **false** if cycle detected, else **true**

This observation is the basis for the *piggyback* algorithm in [16], which was initially designed to perform bounded liveness checking, and improved in [13] to handle liveness properties with a bounded suffix (i.e. the cycle in the state space is bounded in size, as opposed to the cycle in the property automaton). In Algorithm 1, we present a version of the piggyback algorithm tailored for the use of Rabin automata, which is, to the best of our knowledge, the first time this is done. At lines 1, 9 and 12, the acceptance condition for Rabin automata is checked, and the results of those checks are, where applicable, piggybacked along with the piggybacked state. This is the key difference between this and previous versions of the piggyback algorithm.

In Algorithm 1, α refers to the state that is piggybacked along with the state being visited or explored. The notation $\bullet$ refers to a value representing 'no state'. At lines 1–2, the initial state of $\mathcal{P}_{\mathcal{G}}$ is selected as a piggyback value if the initial state of $\mathcal{R}_{\neg\varphi}$ is in at least one K_i (indicated by I), i.e. potentially makes a cycle involving that state accepting. State α, together with the indices in I of the involved (L_i, K_i) are combined with the state. The combination is added to sets *Open* and *Closed* (line 3), where *Open* is the set of states that have been visited, but not yet explored, and *Closed* is the set of explored states.

Next, while *Open* is not empty, states are taken from that set, to be explored (lines 4–5). At line 6, the OUT function returns the outgoing transitions of $\bar{s}$, i.e. $\text{OUT}(\bar{s}) = \{\langle a, \bar{s}'\rangle \mid \bar{s} \xrightarrow{a} \bar{s}'\}$. At line 7, the early termination condition of the algorithm is checked: if at any time, one of the successors of a state is the state being piggybacked, we have clearly detected a cycle. At lines 9–10, it is checked whether successor $\bar{s}'$ can still be in an accepting cycle. This is the case if for at least one of the $(L_i, K_i) \in \mathcal{F}$ relevant for the piggybacked state, $\bar{s}'$ is not in L_i. If such an $(L_i.K_i)$ no longer exists, we remove the current piggybacked state, and consider $\bar{s}'$ as a new piggybacked value (lines 11–13). At line 14, the presence of $\bar{s}'$ in *Closed* is checked. If it is not present, $\bar{s}'$ is added to *Open* and *Closed* with the updated piggybacked information (lines 15–16). Else, if currently a state is being piggybacked, we have to consider re-exploration of $\bar{s}'$: if the piggybacked value of $\bar{s}'$ encountered in *Closed* is $\bullet$, we reopen the state and update its information (lines 19–20). Else, we indicate at line 22 that a so-called *blocking situation* has occurred (which we discuss next). In the end, these situations are further analysed at lines 23–24. Note that re-exploration of a state can be done at most once: if it is in *Closed* without a piggybacked value, and it is reached from a state with a piggybacked value, it will be explored again, and added to *Closed* with the piggybacked value. Hence, the complexity of the algorithm is $O(|\mathcal{S}_{\mathcal{P}}|)$.

Unfortunately, the piggyback algorithm is not complete; in many situations, it fails to detect accepting cycles. Figure 2 illustrates two types of problems, where, for convenience, we consider one pair of state sets $(L, K) \in \mathcal{F}$. In the figure, double-lined states are in K, while no states are in L. Piggybacked states are listed between square brackets, and the search enters via incoming transitions. In Fig. 2a, when reaching $\bar{s}_1 \in K$, this state cannot be piggybacked, since $\bar{s}_0$ already is. This may cause a situation called *shadowing* [13]: the accepting cycle $\bar{s}_1 \rightarrow^+ \bar{s}_1$ is not recognised due to a different state being piggybacked.

Figure 2b illustrates a problem called *blocking* [13]. The search enters the cycle from $\bar{s}_0$ and $\bar{s}_1$, and piggybacks both. This means that when exploring $\bar{s}_2$, $\bar{s}_0$ is encountered with a different piggybacked value, and hence, the search is blocked (line 22 of Algorithm 1), i.e. it cannot continue along that path. Similarly, the search from $\bar{s}_0$ is blocked. The result is that the cycle is not detected.

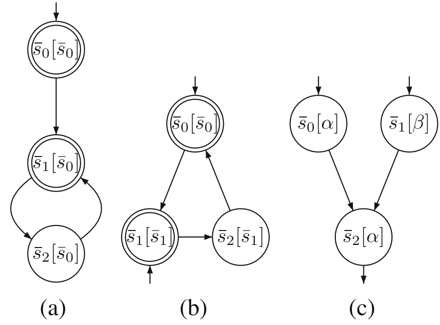

Fig. 2. Shadowing and blocking

However, the occurrence of blocking does not necessarily indicate the presence of cycles. Another cause for blocking is due to the confluence of paths, which tends to occur very frequently in state spaces, due to the interleaving semantics of the input models [27]. This is illustrated in Fig. 2c: from $\bar{s}_0$, value α is piggybacked to $\bar{s}_2$, due to which a search from $\bar{s}_1$ with value $\beta \neq \alpha$ is blocked.

In a way, one can interpret shadowing as a specific form of blocking; in Fig. 2a, the search is blocked at $\bar{s}_2$ due to reaching state $\bar{s}_1$ which already has a piggybacked value not equal to $\bar{s}_1$ itself. In [13], it is proposed to keep track of blocking occurrences during piggybacking, and post-processing those occurrences by starting a DFS with depth-bound b is launched from every point in the state space where blocking occurs, to determine whether or not a cycle of at most size b exists. However, not all blocking occurrences require further analysis. We observe the following, which is formalised in a lemma.

Lemma 1. *Consider an accepting cycle $\pi = s_1 \rightarrow^* s_2 \rightarrow s_3 \rightarrow^* s_1$ with, for some $(L_i, K_i) \in \mathcal{F}$, $S(\pi) \cap K_i \neq \emptyset$ and $S(\pi) \cap L_i = \emptyset$. Say that a search enters π via s_1, and piggybacks a value α along π, but is being blocked at s_3, because the latter already has a piggyback value β. Then, eventually, blocking will also occur at s_1, or an accepting cycle containing s_1 is discovered.*

Proof. By induction on the length m of $s_3 \rightarrow^* s_1$.

- $m = 0$: we have $s_3 = s_1$, therefore blocking at s_1 happened when considering the successors of s_2.
- $m = 1$: s_1 is a successor of s_3. We either have $\beta = s_1$, in which case an accepting cycle containing s_1 is discovered (line 7 of Algorithm 1), or the search is blocked at s_1, since s_1 is in *Closed*, $\beta \neq \bullet$, and $\alpha \neq \bullet$ (line 22 of Algorithm 1).
- $m = m' + 1$: let s_4 be the successor of s_3 in π. There are now two possibilities:
  1. either s_4 has no piggyback value, in which case s_4 will be added with piggyback value β to *Open*, either at line 16 or 20 of Algorithm 1, and by the induction hypothesis, eventually blocking will occur at s_1, or another accepting cycle through s_1 is discovered.
  2. or s_4 has a piggyback value $\gamma \neq \beta$. Then, another search along $s_4 \rightarrow^* s_1$ (with length m') is being conducted with piggyback value γ. By the induction hypothesis, either blocking will occur at s_1, or, in case $\gamma = s_1$ and exploration continues to s_1, an accepting cycle containing s_1 and s_4 will be discovered. □

In Sect. 4, we discuss how we analyse blocking occurrences on the GPU. Lemma 1 is a key observation to reduce the number of occurrences that need to be resolved. In particular, it suffices to look at those blocking occurrences at states by which the search has entered a cycle and has closed process-local cycles. Distinguishing these from others depends on knowing where the cycles are, which we do not. However, in the next section, several new algorithms are presented to statically analyse Lts networks to determine via which states in the Π'_i a search enters a local cycle. If in $\mathcal{P}_\mathcal{G}$ a given state does not enter a local cycle in any of the Π'_i, then it also cannot enter a cycle in $\mathcal{P}_\mathcal{G}$. This relation between cycles in the Π'_i and cycles in $\mathcal{P}_\mathcal{G}$ is discussed next.

3.2 Static Analysis of Lts Networks

In this section, we present several algorithms to statically analyse networks under analysis, with the goal to extract information that can improve the effectiveness of the piggyback algorithm. These algorithms analyse the different Π'_i in isolation, and have a complexity which is at most quadratic in the size of an individual Π'_i. This is often much lower than the complexity of computing $\mathcal{P}_\mathcal{G}$, which is $O(|\mathcal{S}_1| \times \cdots \times |\mathcal{S}_{n+1}|)$. Lemma 2 serves as the starting point for our reasoning.

Lemma 2. *Consider an* Lts *network* $\mathcal{P} = \langle \Pi', \mathcal{V}' \rangle$. *For every cycle* $\pi = \bar{s} \rightarrow^+ \bar{s}$ *in* $\mathcal{P}_\mathcal{G}$, *the following holds for each of the* Π'_i $(i \in 1..n+1)$*:*

1. either $\forall \bar{s}' \in S(\pi). \bar{s}'_i = \bar{s}_i$
2. or $\exists \pi_i = \bar{s}_i \rightarrow^+ \bar{s}_i. \forall s \in S(\pi_i). \exists \langle \bar{s}', s \rangle \in S(\pi)$

Furthermore, for at least one Π'_i*, case 2 holds.*

Lemma 2 follows from the following observation. Consider a state vector $\bar{s} = \langle s_0, \ldots, s_n \rangle \in \mathcal{S}_\mathcal{P}$ that is part of a cycle π. If we traverse π all the way back to $\bar{s}$, then locally in each Π'_i, we have moved from s_i to s_i, meaning that either we did not move at all (case 1) or we have traversed a local cycle in Π'_i (case 2). Of course, for at least one of the Π'_i, case 2 must hold, otherwise π would not contain any transitions, and hence not be a cycle.

Independent Cycles. Lemma 2 implies that if for at least one of the Π_i, s_i is contained in a *non-trivial, independent strongly connected component* (SCC), then $\bar{s}$ (and, in fact, any state vector containing s_i) is part of a cycle in $\mathcal{N}_\mathcal{G}$. An *SCC* is a subgraph in which every state is reachable from every other state in the SCC. It is *non-trivial* if it contains at least one transition.[2] Finally, it is *independent* if it consists of transitions that require no synchronisation with other transitions in the network. Since we avoid the need to synchronise with the property automaton whenever possible (see Sect. 2), there can be SCCs in a

[2] Alternatively, an SCC is called trivial if it contains exactly one state, but here we use the criterion that an SCC is trivial if it contains no transitions.

process LTS that require no synchronisation with any other process LTS, including the property automaton. Being aware of reaching independent cycles during model checking can be very helpful: if we ever reach a state vector containing at least one state in an independent cycle, and an accepting property state, then we immediately know that there is an accepting cycle.

Independent SCCs can be detected in each Π'_i by employing a variant of the well-known SCC algorithm of Tarjan [31], in which we limit ourselves to the exploration of independent transitions, and we launch Tarjan's algorithm once from each of the states $s_i \in \mathcal{S}'_i$, while never reexploring those states that were visited during previous explorations.

This approach runs in a time linear to the number of states, since Tarjan's algorithm runs in linear time, and each state is explored at most once. In the end, trivial SCCs are removed from the results.

Cycle Entry States. From Lemma 2, we know that during a search, a cycle in $\mathcal{P}_{\mathcal{G}}$ can only have been completely traversed as soon as at least one local cycle in one of the Π'_i is traversed. Because of that fact, we would like to analyse the Π'_i to determine which states are possible entry points to cycles (and therefore points where a cycle can also be closed).

There exist algorithms to identify all elementary cycles, i.e. cycles in which no state is present more than once, in a directed graph, for instance [18,31]. However, these algorithms have a time complexity which is exponential in the number of vertices, and therefore larger than we can allow, i.e. larger than computing $\mathcal{P}_{\mathcal{G}}$. Because of this, we propose a new algorithm (Algorithm 2), which does not identify all elementary cycles individually, but instead identifies all states by which at least one elementary cycle can be entered via some path from the initial state. Besides this, for every such state, it keeps track of the transition labels that require synchronisation in $\mathcal{P}$ and are associated to at least one transition of a cycle entered via that state. These labels are elements of what we call the *action dependency set* associated with a state. The algorithm has a worst case complexity of $O(n^2)$, with n the number of states.

As an example, consider the LTS in Fig. 3. The cycle entry states are marked black. If a search starts from s_0, then s_1, s_4, and s_3 are entry states: path $s_0 \rightarrow s_1$ leads to cycles $s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_1$ and $s_1 \rightarrow s_2 \rightarrow s_4 \rightarrow s_3 \rightarrow s_1$, path $s_0 \rightarrow s_4$ leads to $s_4 \rightarrow s_3 \rightarrow s_1 \rightarrow s_2 \rightarrow s_4$, and path $s_0 \rightarrow s_4 \rightarrow s_3$ leads to $s_3 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3$. The action dependency sets are $\{a, b, c, d, e\}$ for s_1, $\{e, c, a, d\}$ for s_4, and $\{c, a, b\}$ for s_3. Note that an SCC detection algorithm does not provide us the desired result. For instance, Tarjan's algorithm will identify $s_1, \ldots, s_4$ as an SCC, with either s_1 or s_4 as root state, depending on the DFS-order, but it will not distinguish s_2 from the other states.

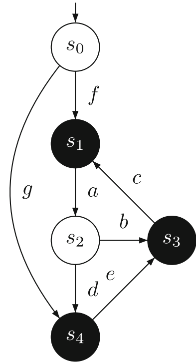

Fig. 3. Cycle entries

The DFS-based algorithm is described in procedure DFSCHECKENTRIES (line 5), which is applied to each of the initial states in $\mathcal{P}$ (line 6), after some global sets have

been initialised. There are three *Closed* sets, i.e. sets consisting of fully explored states: $Closed_{co}$, containing closed states that are in a cycle which is currently open, i.e. partially on the DFS stack, $Closed_{cc}$, containing states that are on a closed cycle, and *Closed*, containing states not on a cycle. Furthermore, for each of the Π'_i, there are several sets:

- $cycleentries_i$ contains the states in $\mathcal{S}'_i$ that have been identified as cycle entry states;
- $creps_i$ contains a subset of $cycleentries_i$ of so-called *representatives*. For each cycle in Π'_i, at least one of its states is representative;
- $sdeps_i$ records prioritised connections between states and representatives, to keep track of which states are directly connected via a transition to a representative, and which states are indirectly connected via states that are directly connected;
- $asets_i$ contains the action dependency sets of the cycle entry states, i.e. states in $cycleentries_i$.

Finally, *cclosing* contains cycle entry states of open cycles, and *curdeps* contains the currently recorded connections between the state at the top of the DFS stack and representatives.

For each transition $s \xrightarrow{a} s'$ of state s for which DFSCHECKENTRIES is called, i.e. which is at the top of the DFS stack, a number of distinct situations is considered (we refer to the DFS stack as *stack* in Algorithm 2):

1: (Line 10): if s' is on the DFS stack ($s' \in stack$), then s' is clearly a cycle entry state, since $\underline{s}_i \rightarrow^* s \rightarrow s'$ leads to cycle $s' \rightarrow^+ s'$. Hence, we add s' to *cclosing* (the cycle is still open) and $cycleentries_i$. We record in *curdeps* that s relates to (representative) s' with priority 1, indicating a direct connection.

2: (Line 13): if s' is in $Closed_{co}$, then we relate s to all representatives that have been related to s', but now with an incremented priority, representing an indirect connection ($sdeps_i(s)(t) = \perp$ at line 15 means that $sdeps_i(s)(t)$ is undefined). For example, if through the LTS of Fig. 3, the algorithm first explores $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3$, then identifies s_1 as cycle entry, records $\langle s_1, 1 \rangle$ as a direct connection for s_3, backtracks over s_3, and then reaches s_3 again via s_4, then s_4 gets the connection $\langle s_1, 2 \rangle$, signifying that s_4 relates to s_1 via s_3.

3: (Line 17): if s' is in $Closed_{cc}$, then s' is a cycle entry state. At line 19, D records those representatives to which s' has a more direct connection than s (with 'undefined' being the weakest connection). If D is not empty, then at lines 21–22, the relevant representatives get connected to s'. If any of the involved connections of s' is not direct, then reopening of s' is required (line 24). Finally, at lines 25–26, those connections are set to 1, by which repeated reopening of s' based on those connections is prevented.

Continuing the example from the previous item, if the search backtracks all the way back to s_0, then $s_1, \ldots, s_4$ are all in $Closed_{cc}$. If then, s_4 is reached again from s_0, s_4 is identified as cycle entry. However, the path $s_0 \rightarrow s_4$ can be extended such that s_3 is cycle entry, but this still needs to be detected. Since $\langle s_1, 2 \rangle$ is an

indirect connection of s_4, it will be reexplored. This leads us to s_3, which will now be identified as cycle entry, at which point the search backtracks, since s_3 has no indirect connections.

4: (Line 27): if s' is neither on the stack, nor in any *Closed* set, then the search continues via s', and the connections of s' are connections for s as well.

Once s' has been explored, a is added to the action dependency set of the states in *curdeps*, if a-transitions require synchronisation (lines 30–31). Finally, *curdeps* is used to update $sdeps_i(s)$: for every representative in either of the two sets, we keep the strongest recorded connection in $sdeps_i(s)$.

When s is fully explored, s is added to *Closed* if there are no connections, i.e. s is not part of a cycle (lines 33–34). Otherwise, s is part of an open cycle (lines 35–36), and hence added to $Closed_{co}$. Finally, if s represents at least one open cycle, we move s to the set of representatives $creps_i$. Backtracking over s means that the associated open cycles are now closed. All states no longer related to an open cycle are moved to $Closed_{cc}$ (lines 37–41).

Algorithm 2. Cycle entry detection

Require: network $\mathcal{P} = \langle \Pi', \mathcal{V}' \rangle$
  procedure IDENTIFYCYCLEENTRIES($\mathcal{P}$):
2: **for all** $i \in 1..n+1$ **do**
  - $Closed, Closed_{co}, Closed_{cc}, cycleentries_i \leftarrow \emptyset$
4: - $sdeps_i, asets_i, creps_i \leftarrow \emptyset$
  - $cclosing, curdeps \leftarrow \emptyset$
6: - DFSCHECKENTRIES($\underline{s}_i, i$)
  procedure DFSCHECKENTRIES(s, i):
8: **for all** $\langle a, s' \rangle \in$ OUT(s) **do**
  - $curdeps \leftarrow \emptyset$
10: **if** $s' \in stack$ **then**
  - add s' to $cclosing$ and $cycleentries_i$
12: - add $\langle s', 1 \rangle$ to $curdeps$
  else if $s' \in Closed_{co}$ **then**
14: **for all** $\langle t, c \rangle \in sdeps_i(s')$ **do**
  if $sdeps_i(s)(t) = \bot$ **then**
16: - add $\langle t, c+1 \rangle$ to $curdeps$
  else if $s' \in Closed_{cc}$ **then**
18: - add s' to $cycleentries_i$
  - $D \leftarrow \{t \mid \langle sdeps_i(s')(t) < sdeps_i(s)(t)\}$
20: **if** $D \neq \emptyset$ **then**
  for all $t \in D$ **do**
22: - $sdeps_i(t)(s') \leftarrow 1$
  if $\exists t \in D.sdeps_i(s')(t) > 1$ **then**
24: - DFSCHECKENTRIES(s', i)
  for all $t \in D$ **do**
26: - set $curdeps(t)$ and $sdeps_i(s')(t)$ to 1
  else if $s' \notin Closed$ **then**
28: - DFSCHECKENTRIES(s', i)
  - $curdeps \leftarrow sdeps_i(s')$
30: **if** a requires synchr. **then**
  - add a to $asets_i$ of the states in $curdeps$
32: - update $sdeps_i(s)$ with $curdeps$
  if $sdeps_i(s) = \emptyset$ **then**
34: - add s to $Closed$
  else if $s \notin Closed_{cc}$ **then**
36: - add s to $Closed_{co}$
  if $s \in cclosing$ **then**
38: - move s from $cclosing$ to $creps_i$
  for all $t \in Closed_{co}$ **do**
40: **if** no states in $sdeps_i(t)$ are in $cclosing$ **then**
  - move t from $Closed_{co}$ to $Closed_{cc}$

Essential Cycle Entries. Finally, having detected all cycle entry states for each of the Π'_i, we can consider

how the Π'_i may synchronise. For instance, if a local cycle $\pi = s_0 \xrightarrow{a} s_0$ exists in Π'_1, and a requires synchronisation, and there is only one other cycle π' containing an a-transition in Π'_2, then any cycle containing a in $\mathcal{P}_\mathcal{G}$ must be the result of combining π and π'. Therefore, to be aware of the possibility of a cycle occurring in $\mathcal{P}_\mathcal{G}$ during a search, it suffices to keep track of only one of the two cycles π and π', i.e. only the cycle entry states of one of the two.

Algorithm 3 selects among the cycle entry states a subset containing what we call the *essential* cycle entry states, or essential states. At lines 1–2, we construct an integrated and sorted version of all the $creps_i$, in which the representatives of all Π'_i are sorted by the number of actions in their action dependency set, from small to large. Then, all representatives are at first selected (lines 3–5). The selection procedure is covered at lines 6–8: if for a representative s of Π'_i, none of its actions is shared with a removed representative from some other Π'_j, then s can be safely removed. Finally, at lines 9–10, all cycle entry states with connections to essential representatives are marked as being essential as well.

Algorithm 3. Selecting essential states

Require: *creps*, *sdeps*, *asets*, $\mathcal{V}'$
- *sorted-creps* $\leftarrow \{\langle i, s\rangle \mid i \in 1..n+1 \wedge s \in creps_i\}$
2: - sort the $\langle i, s\rangle$ in *sorted-creps* on $|asets_i(s)|$
for all $i \in 1..n+1$ **do**
4: - $essential_i \leftarrow creps_i$
- $removed_i \leftarrow \emptyset$
6: **for all** $\langle i, s\rangle \in$ *sorted-creps* **do**
if $\forall a \in asets_i(s). \exists \langle a, T\rangle \in \mathcal{V}', j \in T \setminus \{i\}. a \notin removed_j$ **then**
8: - move s from $essential_i$ to $removed_i$
for all $i \in 1..n+1, s \in essential_i$ **do**
10: - add the states in $sdeps_i(s)$ to $essential_i$

Note that the approach suggested here only approximates the real relation between elementary cycles in the network, since multiple cycles may share cycle entry states, and in those cases, their synchronising actions may be lumped together in one action dependency set. However, improving on this would involve the detection of all individual elementary cycles, which is too time-consuming.

3.3 Using the Static Analysis Results in the Piggyback Algorithm

The algorithms described in Sect. 3.2 can be used in a pre-processing phase to provide additional structural information for the piggyback algorithm. We describe here how and when that information can be used, referring back to Algorithm 1.

First of all, detecting independent cycles in the process LTSs results in a set of states C_i, for each Π'_i, consisting of all the states that are in at least one independent cycle. Whenever the piggyback algorithm reaches a state vector $\bar{s}'$ for which 1) there exists at least one $i \in 1..n$ such that $\bar{s}'_i \in C_i$ and 2) $\{j \mid (L_j, K_j) \in \mathcal{F} \wedge \bar{s}'_{n+1} \in K_j\} \neq \emptyset$, then an accepting cycle has been found, since one can construct from $\bar{s}'$ a cycle by following the transitions in the independent cycle in Π'_i. This can be checked directly after line 6 in Algorithm 1.

Second of all, knowing the cycle entry states for each Π'_i can help to distinguish blocking situations occurring due to the confluence of paths from situations occurring due to the closing of a cycle. When blocking occurs on a state $\bar{s}$ in

which no process state enters a process-local cycle, i.e. no state is a cycle entry state, we can conclude that the blocking cannot have been caused by closing a global (system-level) cycle. This check can be added to Algorithm 1 at line 21. Only if the check evaluates to true, will the blocking occurrence be marked. This can be even further improved if we also keep track of when we are closing process-local cycles, starting from the moment when the current piggyback value was picked up. Only in those cases where at least one such cycle is closed, is it possible that we are also closing a cycle in $\mathcal{P}_\mathcal{G}$. Besides this advantage, one can as well improve the selection of piggyback values in those cases where the Rabin automaton only contains cycles in which the cycle entry states are all in some K_i ($1 \leq i \leq |\mathcal{F}|$): only when a state $\bar{s}$ contains an accepting property state *and* has at least one cycle entry state should it be selected for piggybacking. This can be used as an additional condition in Algorithm 1 at lines 2 and 12.

Finally, knowing the essential states for each Π'_i helps us to further distinguish promising from unpromising situations: only blocking situations involving essential states are interesting for post-processing, since a cycle in $\mathcal{P}_\mathcal{G}$ must be closed by revisiting a state $\bar{s}$ containing an essential state. Furthermore, under the same assumption about the Rabin automaton as described above, information on the essential states can help to make more informed decisions regarding the selection of piggyback values. This affects Algorithm 1 at the same lines as the ones mentioned above for cycle entry information.

4 GPU Model Checking

GPUExplore [35,36] is an explicit-state model checker that practically runs entirely on a GPU (only the general progress is checked on the host side, i.e. by a thread running on the Central Processing Unit (CPU)). It is written in CUDA C, an extension of C offered by Nvidia. It provides the Compute Unified Device Architecture interface to write applications for Nvidia's GPUs. GPUExplore takes an Lts network as input, and can construct the system Lts using many threads in a BFS-based exploration, while checking on-the-fly for the presence of deadlocks and violations of safety properties. A safety property can be added as an automaton to the network. The general approach of GPUExplore is discussed here, leaving out many of the details that are not relevant for understanding the current work. The interested reader is referred to [35,36].

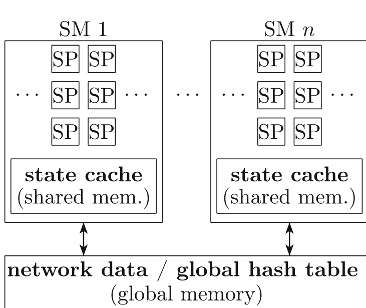

Fig. 4. GPUExplore overview

In a CUDA program, the host launches CUDA functions called *kernels*, that are to be executed many times in parallel by a specified number of GPU threads. Usually, all threads run the same kernel using different parts of the input data, although some GPUs allow multiple different kernels to be executed simultaneously (GPUExplore does not use this feature). Each thread is executed by a streaming processor (SP), see Fig. 4. Threads are grouped in *blocks* of a predefined size. Each block is assigned to a streaming multiprocessor (SM).

Each thread has a number of on-chip registers that allow fast access. The threads in a block together share memory to exchange data, which is located in the (on-chip) *shared memory* of an SM. Finally, the blocks can share data using the *global memory* of the GPU, which is relatively large, but slow, since it is off-chip. The global memory is used to exchange data between the host and the kernel. The GTX Titan X, which we used for our experiments, has 12 GB global memory and 24 SMs, each having 128 SPs, which is in total 3,072 SPs.

Writing well-performing GPU applications is challenging, due to the execution model of GPUs, which is *Single Instruction Multiple Threads*. Threads are partitioned in groups of 32 called *warps*. The threads in a warp run in lock-step, sharing a program counter, so they always execute the same program instruction. Hence, thread divergence, i.e. the phenomenon of threads being forced to execute different instructions (e.g., due to if-then-else constructions) or to access physically distant parts of the global memory, negatively affects performance.

Model checking tends to introduce divergences frequently, as it requires combining the behaviour of the processes in the network, and accessing and storing state vectors of the system state space in the global memory. In GPUexplore, this is mitigated by combining relevant network information as much as possible in 32-bit integers, and storing these as textures, that only allow read access and use a dedicated cache to speed up random accesses. Furthermore, in the global memory, a hash table is used to store state vectors (Fig. 4). The hash table has been designed to optimise accesses of entire warps: the space is partitioned into buckets consisting of 32 integers, precisely enough for one warp to fetch a bucket with one combined memory access. State vectors are hashed to buckets, and placed within a bucket in an available slot. If the bucket is full, another hash function is used to find a new bucket. Each block accesses the global hash table to collect vectors that still require exploration. To each state vector with n process states, a *group* of n threads is assigned to construct its successors using fine-grained parallelism. Since access to the global memory is slow, each block uses a dedicated state cache (Fig. 4). It serves to collect newly produced state vectors, that are subsequently stored in the global hash table in batches. With the cache, block-local duplicates can be detected. The approach allows to work with vectors that require any number of integers smaller than 32 to be stored.

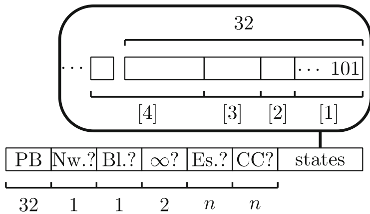

Fig. 5. Encodings of an Lts and a state vector with bookkeeping

The data structures used are illustrated in Fig. 5. At the top, an encoding of the Lts in Fig. 3 is given. The *State offsets* array contains the offsets needed to read the outgoing transitions of a state i in *Transitions*: they are stored from position *State offsets*$[i]$ up to (not including) *State offsets*$[i+1]$. A transition is a pair of integers, the first being the (index of) a transition label, and the second being the target state. When possible, multiple target states are listed in one entry (in the case of non-determinism). Transitions are

stored as compactly as possible: for a transition of Π_i', the $\log_2(|\mathcal{A}_i|)$ bits needed to store the label index are combined with the $\log_2(|\mathcal{S}_i|)$ bits needed to store the target state in one 32-bit integer.

Below the LTS encoding, the encoding of a state vector can be seen. The top part corresponds to how GPUEXPLORE originally used to encode state vectors: the state ID's for the individual process LTSs are concatenated. For each of the Π_i', $\log_2(|\mathcal{S}_i|)$ bits are reserved in the vector. If required, multiple 32-bit integers are used to store a vector. At the bottom of Fig. 5, the structure of a state vector extended with bookkeeping bits is displayed. Besides the states of the Π_i':

- n bits are reserved to indicate which of the processes have already fully traversed a local cycle since the last piggyback value was picked up (CC?);
- n bits are reserved to indicate which of the states are essential (Es.?);
- Two bits are reserved to indicate whether the property automaton is in a state that requires infinite visits, finite visits, or neither (∞?);
- One bit is used to indicate whether blocking occurred on the vector (Bl.?);
- One bit indicates whether the vector is new, i.e. requires exploration (Nw.?);
- Thirty-two bits are reserved to store a piggyback value (PB).

Instead of piggybacking vectors, which is costly space-wise, pointers to the global hash table are piggybacked. One 32-bit integer suffices: in 27 bits, we store the index of a bucket (in 12 GB, about 100 million buckets can be stored), and with the remaining 5 bits, the position of a vector within the bucket is given.

GPUEXPLORE is extended as follows to support the checking of linear-time properties. First of all, the network data contains the information we obtained through static analysis, plus the status of the property states (requires infinite visits, finite visits, or neither). The current implementation supports the use of Rabin automata in which $\mathcal{F}$ is a singleton; this can be straightforwardly extended by increasing the number of bookkeeping bits. Furthermore, transitions that lead to a cycle entry state and close a cycle are marked, and essential states are marked. Second of all, using that information, we mark state vectors in the obvious way, i.e. the status of the individual process states is kept in the vector. To do this correctly, it is important that these markings are at times merged; different groups may construct the same successor state with different markings. Both when storing states in the local state cache, and in the global hash table, this merging is performed.

The overall approach follows Algorithm 1 and the extensions discussed in Sect. 3.3. If all cycles in the Rabin automaton can only be entered via states in K ($\mathcal{F} = \{(L, K)\}$), then visiting a vector state with a property state in K is only selected as a piggyback value if at least one state in the vector is essential. This can avoid some of the issues of standard piggybacking, in particular shadowing such as in Fig. 2a, since $\bar{s}_0$ will not be selected as a piggyback value if no state in it is essential.

When blocking occurs on a state vector $\bar{s}$, this is marked (in Bl.?) *only* if $\bar{s}$ contains an essential state (Es.?) and at least one local cycle has been traversed (CC?). By Lemma 1, this is sufficient. Once the state space has been completely explored without detection of a counter-example, post-processing of the blocking

occurrences must be conducted. Since DFSs cannot be performed efficiently on a GPU, we need to perform this post-processing in a way different from the technique in [13]. Instead, we launch up to sixteen parallel BFSs at a time, each performed by several thread blocks from one of the marked states. During each BFS, visited and explored states are marked in the global hash table using two bits. The space previously occupied to store a piggyback value pointer can be reused for this purpose, hence the space for 32/2 parallel BFSs. Each BFS is bounded by the fact that the global hash table is scanned completely for open states up to a predefined number of times. The search in each BFS can be limited to those processes for which it was indicated that they fully traversed a local cycle (in the CC? bits). The sixteen groups of thread blocks process all marked states in this way, until either none are left, or an accepting cycle has been detected.

5 Experimental Results

GPUExplore is equipped with a separate preprocessor, written in Python. It can read Lts networks and produces an output file that configures the GPU model checker. To the preprocessor, we added the ability to read Rabin automata, which are stored as normal Lts*s* with some additional information regarding $\mathcal{F}$. Also, we implemented the algorithms explained in Sect. 3. Results on cycle entry states and essential states is added to the configuration file, while results in independent SCCs are added by adapting the Lts*s* in the network: any state appearing in such an SCC is equipped with an independent selfloop. This allows the model checker to efficiently determine whether a state is in an independent SCC or not, just by looking at the outgoing transitions of that state.

We conducted experiments using input models from various sources, namely the BEEM database [26], the VLTS benchmark suite[3], the mCRL2 website[4], and two models (`ABP` and `broadcast`) designed by us.

To conduct the experiments, a machine with an AMD A6-3670 CPU, 16 GB memory, and an Nvidia GeForce Titan X GPU, running Linux Mint 17.2 was used. GPUExplore used 3120 blocks, 512 threads per block, which turned out to be optimal for reachability analysis [35,36]. Table 1 provides the runtime results in seconds. Besides reachability analysis (**Rch.**) and standard piggybacking (**PB**), several variants of the algorithm have been used, namely:

- **+iSCC**: A version using only the information on independent SCCs;
- **+SPB**: +iSCC plus smart piggyback value selection based on essential states;
- **+SBR**: +SPB with smart blocking resolution, i.e. only blockings with essential states and the closing of a local cycle are considered.

[3] http://cadp.inria.fr/resources/vlts.
[4] http://www.mcrl2.org.

Table 1. GPU runtimes (in seconds) for various piggyback variants

Model	Prop.	$\lvert\mathcal{S}_{\mathcal{P}_{\mathcal{G}}}\rvert$	Rch.	PB		+iSCC		+SPB		+SBR	
				Time	?	Time	?	Time	?	Time	?
1394	request-response	200K	2.68	5.01	✓	5.05	✓	4.10	✓	**3.02**	✓
1394.1	request-response	36.9M	10.42	16.39	✓	16.37	✓	13.20	✓	**12.11**	✓
acs	lock eventually freed	4.8K	1.61	0.40	✗	0.37	✗	0.38	✗	**0.35**	✗
acs.1	lock eventually freed	200K	2.14	0.45	✗	0.45	✗	0.46	✗	**0.43**	✗
wafer stepper.1	□◇ all wafers exposed	3.8M	6.61	35.34	✓	35.32	✓	23.51	✓	**12.63**	✓
ABP	request-response	481.8M	968.44	253.57	✗	258.21	✗	230.58	✗	**180.35**	✗
broadcast	□◇ communication succeeds	105.4M	151.05	53.43	✗	57.36	✗	48.35	✗	**47.36**	✗
transit	□◇ message sent or buffered	4.4M	9.06	0.99	✗	**0.81**	✗	t.o.p.	-	t.o.p.	-
asyn3	□◇ leader announced or reset	17.2M	49.65	2.84	✗	2.75	✗	2.75	✗	**2.70**	✗
asyn3.1	□◇ leader announced or reset	215.4M	472.43	**2.39**	✗	2.52	✗	2.40	✗	2.43	✗
ODP	□◇ WORK executed	178K	4.65	1.14	✗	**1.11**	✗	**1.11**	✗	1.12	✗
ODP.1	□◇ WORK executed	10.1M	13.49	1.99	✗	2.02	✗	2.03	✗	**1.91**	✗
lamport.8	$\square\lozenge P0@CS$	35.1M	35.89	2.33	✗	2.32	✗	**2.26**	✗	2.27	✗
lann.6	$\square\lozenge P0@CS$	144M	136.23	**1.62**	✗	1.64	✗	4.03	✗	1.64	✗
lann.7	$\square\lozenge P0@CS$	160M	202.46	1.56	✗	**1.50**	✗	1.51	✗	1.51	✗
peterson.7	$\square P0$ wait $\Longrightarrow \lozenge P0@CS$	142.5M	4223.36	**368.30**	✗	384.12	✗	502.53	✗	712.63	✗
szymanski.5	$\square P0$ wait $\Longrightarrow \lozenge P0@CS$	79.5M	323.34	**91.30**	✗	102.27	✗	165.52	✗	181.54	✗

In Table 1, for each model, a brief description of the property is given, where '□' and '◇' are shorthand for 'always' and 'eventually', in line with the LTL operators. For each experiment, its result is reported, where ✓ indicates that the property is satisfied, and ✗ means that it is not.

We do not compare the GPUexplore runtimes with those of a standard CPU model checker. It has been established that on the same benchmarks, GPUexplore outperforms state-of-the-art (single-core) explicit-state model checkers by one to two orders of magnitude [35,36]. If we can establish that the checking of linear-time properties can be done in comparable runtimes, then we can safely conclude that the GPU can be effectively applied for this as well.

Table 1 does not provide the runtimes of the preprocessing steps for each experiment, which involve the relevant algorithms described in Sect. 3. In practically all the cases, preprocessing took only a fraction of the subsequent exploration time. An exception to this is the `transit` model, for which the detection of cycle entry states and essential states caused a time-out (t.o.p. = time-out during preprocessing in Table 1). In that model, one process Lts is much larger than the others, making it costly to analyse it in comparison to directly exploring the system Lts.

First of all, we experienced that for most of the analysed models, the standard piggyback algorithm is already very efficient, and we can conclude that it is suitable to check liveness properties with a GPU. Concerning the proposed

extensions, there are two types of situations where we experienced improved runtimes. The first situation is when post-processing has to be performed. In the cases `1394`, `1394.1`, and `wafer stepper.1`, the extensions were more efficient, due to being less prone to mark blocking occurrences. The second situation is when the infinitely visitable states in the Rabin automaton are not immediately reached in the state space search. More precisely, in those cases where there is a non-empty prefix in the property automaton before the suffix expressing the infinite behaviour part, the extensions made more informed decisions. This was the case for `acs` and `broadcast`. The ability to identify independent SCCs was also helpful in some cases: for `transit`, `ODP`, and `lann.7`, +iSCC was the most efficient option.

Finally, it should be noted that for `peterson.7` and `szymanski.5`, two cases from the BEEM database, the extensions actually lead to worse results. It turns out that the infinitely visitable states of the Rabin automata are reachable already very early on during the state space search, leading to many possible candidates for piggybacking. This holds for all the extensions. In general, on the GPU, the extensions have one major drawback compared to standard piggybacking, which is the fact that vector markings have to be maintained in the global hash table. This means that additional writes to the global memory must be done frequently. In case this additional marking work does not productively contribute to finding a counter-example, the runtime only increases. It would be interesting to investigate whether the same can be observed when model checking is done on the CPU.

Concluding, the piggyback algorithm is very suitable to efficiently check liveness properties with a GPU. Under certain circumstances, the suggested extensions improve on these results, in particular when post-processing needs to be performed, and when the property has a non-empty prefix. In several cases, however, applying the extensions actually negatively influenced the runtimes. To overcome this, it seems that a good strategy would be to first run the standard piggyback algorithm, and stop it if post-processing would be required. Instead of post-processing, one could then launch the most informed variant (+SBR), to try to find a counter-example in that way. In case the latter is also not successful, post-processing is still possible.

6 Conclusions

We presented a method to check linear-time properties using a BFS-based search technique. It employs the piggybacking algorithm, adapted to take new insights regarding blocking into account. Furthermore, it uses structural information of the input model obtained by applying new preprocessing algorithms. Like standard piggybacking, it is complete for bounded-suffix model checking, i.e. for finding counter-examples involving cycles no longer than a given bound.

We demonstrated that both the original piggyback algorithm and the proposed extensions generally work effectively in a GPU model checker, but we expect that the extensions also work well in other settings where BFS-based techniques need to be applied.

Future Work. We plan to involve fairness constraints, to rule out specific counter-examples, and reduction techniques, such as partial order reduction [2].

References

1. Alpern, B., Schneider, F.: Defining liveness. Inform. Process. Lett. **21**(4), 181–185 (1985)
2. Baier, C., Katoen, J.P.: Principles of Model Checking. The MIT Press, Cambridge (2008)
3. Barnat, J., Bauch, P., Brim, L., Češka, M.: Designing fast LTL model checking algorithms for many-core GPUs. J. Parall. Distrib. Comput. **72**, 1083–1097 (2012)
4. Barnat, J., Brim, L., Ročkai, P.: A time-optimal on-the-fly parallel algorithm for model checking of weak LTL properties. In: Breitman, K., Cavalcanti, A. (eds.) ICFEM 2009. LNCS, vol. 5885, pp. 407–425. Springer, Heidelberg (2009)
5. Barnat, J., Brim, L., Stříbrná, J.: Distributed LTL model-checking in SPIN. In: Dwyer, M.B. (ed.) SPIN 2001. LNCS, vol. 2057, pp. 200–216. Springer, Heidelberg (2001)
6. Bartocci, E., DeFrancisco, R., Smolka, S.: Towards a GPGPU-parallel SPIN model checker. In: SPIN, pp. 87–96. ACM (2014)
7. Blom, S., van de Pol, J., Weber, M.: LTSmin: distributed and symbolic reachability. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 354–359. Springer, Heidelberg (2010)
8. Bošnački, D., Edelkamp, S., Sulewski, D., Wijs, A.: Parallel probabilistic model checking on general purpose graphic processors. STTT **13**(1), 21–35 (2011)
9. Ceška, M., Pilar, P., Paoletti, N., Brim, L., Kwiatkowska, M.: PRISM-PSY: precise GPU-accelerated parameter synthesis for stochastic systems. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 367–384. Springer, Heidelberg (2016). doi:10.1007/978-3-662-49674-9_21
10. Courcoubetis, C., Vardi, M., Wolper, P., Yannakakis, M.: Memory efficient algorithms for the verification of temporal properties. In: Clarke, E.M., Kurshan, R.P. (eds.) CAV 1990. LNCS, vol. 531, pp. 233–242. Springer, Heidelberg (1990)
11. Dill, D.: The murphi verification system. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 390–393. Springer, Heidelberg (1996)
12. Evangelista, S., Laarman, A., Petrucci, L., van de Pol, J.: Improved multi-core nested depth-first search. In: Chakraborty, S., Mukund, M. (eds.) ATVA 2012. LNCS, vol. 7561, pp. 269–283. Springer, Heidelberg (2012)
13. Filippidis, I., Holzmann, G.: An improvement of the piggyback algorithm for parallel model checking. In: SPIN, pp. 48–57. ACM (2014)
14. Geldenhuys, J., Valmari, A.: Tarjan's algorithm makes on-the-Fly LTL verification more efficient. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 205–219. Springer, Heidelberg (2004)
15. Grädel, E., Thomas, W., Wilke, T. (eds.): Automata, Logics, and Infinite Games. LNCS, vol. 2500. Springer, Heidelberg (2002)
16. Holzmann, G.J.: Parallelizing the spin model checker. In: Donaldson, A., Parker, D. (eds.) SPIN 2012. LNCS, vol. 7385, pp. 155–171. Springer, Heidelberg (2012)
17. Holzmann, G., Peled, D., Yannakakis, M.: On nested depth first search. In: SPIN, pp. 23–32. American Mathematical Society (1996)
18. Johnson, D.: Finding all the elementary circuits of a directed graph. SIAM J. Comput. **4**(1), 77–84 (1975)

19. Kahn, A.: Topological sorting of large networks. Commun. ACM **5**(11), 558–562 (1962)
20. Klein, J., Baier, C.: Experiments with deterministic ω-automata for formulas of linear temporal logic. TCS **363**(2), 182–195 (2006)
21. Kupferman, O.: Automata theory and model checking. In: Clarke, E.M., Henzinger, T.A., Veith, H. (eds.) Handbook of Model Checking. Springer, New York (2015)
22. Kurshan, R.P., Levin, V., Minea, M., Peled, D.A., Yenigün, H.: Static partial order reduction. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 345–357. Springer, Heidelberg (1998)
23. Laarman, A., Langerak, R., van de Pol, J., Weber, M., Wijs, A.: Multi-core nested depth-first search. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 321–335. Springer, Heidelberg (2011)
24. Lang, F.: Refined interfaces for compositional verification. In: Najm, E., Pradat-Peyre, J.-F., Donzeau-Gouge, V.V. (eds.) FORTE 2006. LNCS, vol. 4229, pp. 159–174. Springer, Heidelberg (2006)
25. Molnár, V., Darvas, D., Vörös, A., Bartha, T.: Saturation-based incremental LTL model checking with inductive proofs. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 643–657. Springer, Heidelberg (2015)
26. Pelánek, R.: BEEM: benchmarks for explicit model checkers. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 263–267. Springer, Heidelberg (2007)
27. Pelánek, R.: Properties of state spaces and their applications. STTT **10**(5), 443–454 (2008)
28. Rabin, M.: Decidability of second order theories and automata on infinite trees. Trans. AMS **141**, 1–35 (1969)
29. Schwoon, S., Esparza, J.: A note on on-the-fly verification algorithms. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 174–190. Springer, Heidelberg (2005)
30. Sun, J., Liu, Y., Dong, J.S., Wang, H.H.: Specifying and verifying event-based fairness enhanced systems. In: Liu, S., Araki, K. (eds.) ICFEM 2008. LNCS, vol. 5256, pp. 5–24. Springer, Heidelberg (2008)
31. Tarjan, R.: Depth-first search and linear graph algorithms. SIAM J. Comput. **1**(2), 146–160 (1972)
32. Vardi, M., Wolper, P.: An automata-theoretic approach to automatic program verification. In: LICS, pp. 332–344. IEEE (1986)
33. Wijs, A.: GPU accelerated strong and branching bisimilarity checking. In: Baier, C., Tinelli, C. (eds.) TACAS 2015. LNCS, vol. 9035, pp. 368–383. Springer, Heidelberg (2015)
34. Wijs, A.J., Bošnački, D.: Improving GPU sparse matrix-vector multiplication for probabilistic model checking. In: Donaldson, A., Parker, D. (eds.) SPIN 2012. LNCS, vol. 7385, pp. 98–116. Springer, Heidelberg (2012)
35. Wijs, A., Bošnački, D.: GPUexplore: many-core on-the-fly state space exploration using GPUs. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014 (ETAPS). LNCS, vol. 8413, pp. 233–247. Springer, Heidelberg (2014)
36. Wijs, A., Bošnački, D.: Many-Core On-The-Fly Model Checking of Safety Properties Using GPUs. STTT (2016)
37. Wijs, A., Katoen, J.-P., Bošnački, D.: GPU-based graph decomposition into strongly connected and maximal end components. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 310–326. Springer, Heidelberg (2014)

38. Wu, Z., Liu, Y., Liang, Y., Sun, J.: GPU accelerated counterexample generation in LTL model checking. In: Merz, S., Pang, J. (eds.) ICFEM 2014. LNCS, vol. 8829, pp. 413–429. Springer, Heidelberg (2014)
39. Wu, Z., Liu, Y., Sun, J., Shi, J., Qin, S.: GPU accelerated on-the-fly reachability checking. In: ICECCS, pp. 100–109. IEEE (2015)