Improved On-The-Fly Livelock Detection:
Combining Partial Order Reduction and Parallelism for DFS$_{\text{FIFO}}$

Alfons Laarman[1] and David Faragó[2]

[1] Formal Methods and Tools, University of Twente, The Netherlands
a.w.laarman@cs.utwente.nl
[2] Logic and Formal Methods, Karlsruhe Institute of Technology, Germany
farago@kit.edu

Abstract. Until recently, the preferred method of livelock detection was via LTL model checking, which imposes complex constraints on partial order reduction (POR), limiting its performance and parallelization. The introduction of the DFS$_{\text{FIFO}}$ algorithm by Faragó et al. showed that livelocks can theoretically be detected faster, simpler, and with stronger POR. For the first time, we implement DFS$_{\text{FIFO}}$ and compare it to the LTL approach by experiments on four established case studies. They show the improvements over the LTL approach: DFS$_{\text{FIFO}}$ is up to 3.2 times faster, and it makes POR up to 5 times better than with SPIN's NDFS. Additionally, we propose a parallel version of DFS$_{\text{FIFO}}$, which demonstrates the efficient combination of parallelization and POR. We prove parallel DFS$_{\text{FIFO}}$ correct and show why it provides stronger guarantees on parallel scalability and POR compared to LTL-based methods. Experimentally, we establish almost ideal linear parallel scalability and POR close to the POR for safety checks: easily an order of magnitude better than for LTL.

1 Introduction

Context. In the *automata-theoretic* approach to *model checking* [27], the behavior of a system-under-verification is modeled, along with a property that it is expected to adhere to, in some concise specification language. This model $\mathcal{M}$ is then unfolded to yield a *state space* automaton $\mathcal{A}_{\mathcal{M}}$ (cf. Def. 1). *Safety properties*, e.g. *deadlocks* and *invariants*, can be checked directly on the states in $\mathcal{A}_{\mathcal{M}}$ as they represent all configurations of $\mathcal{M}$. This check can be done during the unfolding, *on-the-fly*, saving resources when a property violation is detected early on.

For more complicated properties, like *liveness properties* [1], $\mathcal{A}_{\mathcal{M}}$ is interpreted as an ω-automaton whose language $\mathcal{L}(\mathcal{A}_{\mathcal{M}})$ represents all infinite executions of the system. A property φ, expressed in linear temporal logic (LTL), is likewise translated to a Büchi or ω-automaton $\mathcal{A}_{\neg\varphi}$ representing all *undesired* infinite executions. The intersected language $\mathcal{L}(\mathcal{A}_{\mathcal{M}}) \cap \mathcal{L}(\mathcal{A}_{\neg\varphi})$ now consists of all counterexample traces, and is empty if and only if the system is correct with respect to the property. The emptiness check is reduced to the graph problem of finding cycles with designated accepting states in the *cross product* $\mathcal{A}_{\mathcal{M}} \otimes \mathcal{A}_{\neg\varphi}$ (cf. Sec. 2). The nested depth-first search (NDFS) algorithm [6] solves it in time linear to the size of the product and on-the-fly as well.

Motivation. The model checking approach is limited by the so-called *state space explosion* problem [1], which states that $\mathcal{A}_{\mathcal{M}}$ is exponential in the components of the system, and $\mathcal{A}_{\neg\varphi}$ exponential in the size of φ. Luckily, several remedies exist to this problem: patience, specialization and state space reduction techniques.

State space reduction via *partial order reduction* (POR) prunes $\mathcal{A}_{\mathcal{M}}$ by avoiding irrelevant interleavings of local components in $\mathcal{M}$ [16,26]: only a sufficient subset of successors, the *ample set*, is considered in each state (cf. Sec. 2). For safety properties, the ample set can be computed locally on each state. For liveness properties, however, an additional condition, the *cycle proviso*, is needed to avoid the so-called *ignoring problem* [9]. POR can yield exponential reductions.

Patience also pays of exponentially as Moore's law stipulates that the number of transistors available in CPUs and memory doubles every 18 months [22]. Due to this effect, model checking capabilities have increased from handling a few thousand states to covering billions of states recently (this paper and [5]). While this trend happily continues to increase memory sizes, it recently stopped benefitting the sequential performance of CPUs because physical limitations were reached. Instead, the available parallelism on the chips is rapidly increasing. So, for runtime to benefit from Moore's law, we must parallelize our algorithms.

Specialization towards certain subclasses of liveness properties, finally, can also help to solve them more efficiently. For instance, a limitation to the *CTL* and the *weak-LTL* fragments was shown to be efficiently parallelizable [25,3]. *In this paper, we limit the discourse to livelock properties*, an important subclass (used in about half of the case studies of [3] and a third of [24]) that investigates starvation, occurring if an infinite run does not make *progress* infinitely often. The definition of progress is up to the system designer and could for instance refer to an increase of a counter or access to a shared resource. The SPIN model checker allows the user to specify progress statements inside the specification of the model [12], which are then represented in the model by the state label 'progress' and referenced by the predefined *progress* LTL property [15]. Until 1996, SPIN used a specific livelock verification algorithm. Section 6 of [15] states that it was replaced by LTL model checking due to its incompatibility with POR.

Problem. LTL model checking can likely not be parallelized efficiently. The current state-of-the-art reveals that parallel cycle detection algorithms either raise the worst-case complexity to L^2 [3] or to $L \cdot P$ [8], where L is the size of the LTL cross product and P the number of processors. Moreover, its additional constraints on POR severely limit its reduction capabilities, even if implemented with great care (see models allocation, cs and p2p in Table 1 in the appendix of [9]). Last but not least, these constraints also limit the parallelization of POR [2].

We want to investigate whether better results can be obtained for livelocks, for which recently an efficient algorithm was proposed by Faragó et al. [11]: DFS_{FIFO}. In theory, it has additional advantages over the LTL approach:

1. It uses the progress labels in the model directly without the definition of an LTL property; avoiding the calculation of a larger cross product.

[3] PROMELA database: `http://www.albertolluch.com/research/promelamodels`.

2. It requires only one pass over the state space, while the NDFS algorithm, typically used for liveness properties, requires two.
3. It eliminates the need for the expensive cycle proviso with POR. Not only is the cycle proviso a highly limiting factor in state space reduction [9], it also complicates the parallelization of the problem [2].
4. It finds the shortest counterexample with respect to progress.

But DFS_{FIFO} is yet to be implemented and evaluated experimentally, so its practical performance is unknown. Additionally, a few hypotheses stand unproven:

1. The algorithm's strategy to delay progress as much as possible, may also be a good heuristic for finding livelocks early, making it more on-the-fly.
2. Its POR performance might be close to that of safety checks, because the cycle proviso is no longer required [11], and the visibility proviso (see Table 1) is also positively influenced by the postponing of progress.
3. The use of *progress transitions* instead of *progress states* is possible, semantically more accurate, and can yield better partial order reductions.

Furthermore, no parallelization exists for the DFS_{FIFO} algorithm.

Contributions. We implemented the DFS_{FIFO} algorithm in the LTSMIN [21,5], with both progress states and transitions. For the latter, we extended theory, algorithms, proofs, models and implementation. We compare the runtime and POR performance to that of LTL approaches using NDFS. For DFS_{FIFO}, we also investigate the effect of using progress transitions instead of states on POR.

Additionally, we present a parallel livelock algorithm based on DFS_{FIFO}, together with a proof of correctness. While the algorithm builds on previous efficient parallelizations of the NDFS algorithm [8,17,19], we show that it has stronger guarantees for parallel scalability due to the nature of the underlying DFS_{FIFO} algorithm. At the same time, it retains all the benefits of the original DFS_{FIFO} algorithm. This entails the redundancy of the cycle proviso, hence allowing for parallel POR with almost the same reductions as for safety checks.

Our experiments confirm the theoretical expectations: using DFS_{FIFO} on four case studies, we observed up to 3.2 times faster runtimes than with the use of an LTL property and the NDFS algorithm, even compared to measurements with the SPIN model checker. But we also confirm all hypotheses of Faragó et al.: the algorithm is more on-the-fly, and POR performance is closer to that of safety checks than the LTL approach, making it up to 5 times more effective than POR in SPIN. Our parallel version of the algorithm can work with POR and features the expected linear scalability. Its combination with POR easily outperforms other parallel approaches [3].

Overview. In Sec. 2, we recapitulate the intricacies of livelock detection via LTL and via non-progress detection, as well as POR. In Sec. 3, we introduce DFS_{FIFO} for progress transitions with greater detail and formality than in [11], as well as its combination with POR. Thereafter, in Sec. 4, we provide a parallel version of DFS_{FIFO} with a proof of correctness, implementation considerations, and an analysis on its scalability. Sec. 5 presents the experimental evaluation, comparing DFS_{FIFO}'s (POR) performance and scalability against the (parallel) LTL algorithms in SPIN [13,15], DIVINE [2,3], and LTSMIN [5,21]. We conclude in Sec. 6.

2 Preliminaries

Model checking of safety properties. Explicit-state model checking algorithms construct $\mathcal{A}_\mathcal{M}$ on-the-fly starting from the initial state s_0, and recursively applying the next-state function *post* to discover all reachable states $\mathcal{R}_\mathcal{M}$. This only requires storing states (no transitions). As soon as a counterexample is discovered, the exploration can terminate early, saving resources. To reason about these algorithms, it is however easier to consider $\mathcal{A}_\mathcal{M}$ structurally as a graph.

Definition 1 (State Space Automaton). *An automaton is a quintuple $\mathcal{A}_\mathcal{M} = (\mathcal{S}_\mathcal{M}, s_0, \Sigma, \mathcal{T}_\mathcal{M}, L)$, with $\mathcal{S}_\mathcal{M}$ a finite set of states, $s_0 \in \mathcal{S}_\mathcal{M}$ an initial state, Σ a finite set of action labels, $\mathcal{T}_\mathcal{M} \colon \mathcal{S}_\mathcal{M} \times \Sigma \to \mathcal{S}_\mathcal{M}$ the transition relation, and $L \colon \mathcal{S}_\mathcal{M} \to 2^{AP}$ a state labeling function, over a set of atomic propositions AP.*

We also use the recursive application of the transition relation $\mathcal{T}$: $s \xrightarrow{\pi}{}^{+} s'$ iff π is a path in $\mathcal{A}_\mathcal{M}$ from s to s', or $s \xrightarrow{\pi}{}^{*} s'$ if possibly $s = s'$. We treat a path π dually as a sequence of states and a sequence of actions, depending on the context. We omit the subscript $\mathcal{M}$ whenever it is clear from the context.

Now, we can define: the reachable states $\mathcal{R}_\mathcal{M} = \{s \in \mathcal{S}_\mathcal{M} \mid s_0 \to^* s\}$, the function $post \colon \mathcal{S}_\mathcal{M} \to 2^\Sigma$, such that $post(s) = \{\alpha \in \Sigma \mid \exists s' \in \mathcal{S}_\mathcal{M} : (s, \alpha, s') \in \mathcal{T}_\mathcal{M}\}$ and $\alpha(s)$ as the unique next-state for s, α if $\alpha \in post(s)$, i.e. the state t with $(s, \alpha, t) \in \mathcal{T}_\mathcal{M}$. Note that a state $s \in \mathcal{S}$ comprises the variable valuations and process counters in $\mathcal{M}$. Hence, we can use any proposition over these values as an atomic proposition representing a state label. For example, we may write $\mathsf{progress} \equiv \mathsf{Peterson0} = CS$ to have $\mathsf{progress} \in L(s)$ iff s represents a state where process instance 0 of Peterson is in its critical section CS. Or we can write $\mathsf{error} \equiv N > 1$ to express the mutual exclusion property, with N the number of processes in CS. These state labels can then be used to check safety properties using *reachability*, e.g., an *invariant* '$\neg error$' to check mutual exclusion in $\mathcal{M}$.

LTL model checking. For an LTL property, the property φ is transformed to an ω-automaton $\mathcal{A}_{\neg\varphi}$ as detailed in [27]. Structurally, the ω-automaton extends a normal automaton (Def. 1) with dedicated accepting states (see Def. 2). Semantically, these accepting states mark those cycles that are part of the ω-regular language $\mathcal{L}(\mathcal{A}_{\neg\varphi})$ as defined in Def. 3.

To check correctness of $\mathcal{M}$ with respect to a property φ, the *cross product* of $\mathcal{A}_{\neg\varphi}$ with the state space $\mathcal{A}_\mathcal{M}$ is calculated: $\mathcal{A}_{\mathcal{M}\times\varphi} = \mathcal{A}_\mathcal{M} \otimes \mathcal{A}_{\neg\varphi}$. The states of $\mathcal{S}_{\mathcal{M}\times\varphi}$ are formed by tuples (s, s') with $s \in \mathcal{S}_\mathcal{M}$ and $s' \in \mathcal{S}_{\neg\varphi}$, with $(s, s') \in \mathcal{F}$ iff $s' \in \mathcal{F}_{\neg\varphi}$. Hence, the number of possible states $|\mathcal{S}_{\mathcal{M}\times\varphi}|$ equals $|\mathcal{S}_\mathcal{M}| \cdot |\mathcal{S}_{\neg\varphi}|$, whereas the number of reachable states $|\mathcal{R}_{\mathcal{M}\times\varphi}|$ may be smaller. The transitions in $\mathcal{T}_{\mathcal{M}\times\varphi}$ are formed by synchronizing the transition labels of $\mathcal{A}_{\neg\varphi}$ with the state labels in $\mathcal{A}_\mathcal{M}$. For an exact definition of $\mathcal{T}_{\mathcal{M}\times\varphi}$, we refer to [1].

Definition 2 (Accepting states). *The set of accepting states $\mathcal{F}$ corresponds to those states with a label $\mathsf{accept} \in AP$: $\mathcal{F} = \{s \in \mathcal{S} \mid \mathsf{accept} \in L(s)\}$.*

Definition 3 (Accepting run). *A lasso-formed path $s_0 \xrightarrow{v}{}^{*} s \xrightarrow{w}{}^{+} s$ in $\mathcal{A}$, with $s \in \mathcal{F}$, constitutes an accepting run, part of the language of $\mathcal{A}$: $vw^\omega \in \mathcal{L}(\mathcal{A})$.*

As explained in Sec. 1, the whole procedure of finding counterexamples to φ for $\mathcal{M}$ is now reduced to the graph problem of finding *accepting runs* in $\mathcal{A}_{\mathcal{M}\times\varphi}$. This can be solved by the nested depth-first search (NDFS) algorithm, which does at most two explorations of all states $\mathcal{R}_{\mathcal{M}\times\varphi}$. Since $\mathcal{A}_{\mathcal{M}\times\varphi}$ can be constructed on-the-fly, NDFS saves resources when a counterexample is found early on.

Livelock detection. *Livelocks* form a specific, but important subset of the liveness properties and can be expressed as the *progress* LTL property: $\Box\Diamond\mathsf{progress}$, which states that on each infinite run, *progress* needs to be encountered infinitely often. As the LTL approach synchronizes the state labels of $\mathcal{A}_{\mathcal{M}}$ (see Def. 3), it requires that progress is defined on states as in Def. 4.

Definition 4 (Progress states). *The set of progress states $\mathcal{S}^{\mathcal{P}}$ corresponds to those states with a state label $\mathsf{progress} \in AP$: $\mathcal{S}^{\mathcal{P}} = \{s \in \mathcal{S} \mid \mathsf{progress} \in L(s)\}$.*

Definition 5 (Non-progress cycle). *A reachable cycle π in $\mathcal{A}_{\mathcal{M}}$ is a non-progress cycle (**NPcycle**) iff it contains no progress $\mathcal{P}$.*

We define $\mathcal{NP}$ as a set of states: $\mathcal{NP} = \{s \in \mathcal{S}_{\mathcal{M}} \mid \exists \pi : s \xrightarrow{\pi}{}^{+} s \wedge \pi \cap \mathcal{P} = \emptyset\}$.

Theorem 1. *Under $\mathcal{P} = \mathcal{S}^{\mathcal{P}}$, $\mathcal{A}_{\mathcal{M}}$ contains a **NPcycle** iff the crossproduct with the progress property $\mathcal{A}_{\mathcal{M}\times\Box\Diamond\mathsf{progress}}$ contains an accepting cycle.*

Livelocks can however also be detected directly on $\mathcal{A}_{\mathcal{M}}$ if we consider for a moment that a counterexample to a livelock is formed by an infinite run that lacks progress $\mathcal{P}$, with $\mathcal{P} = \mathcal{S}^{\mathcal{P}}$. By proving absence of such *non-progress* cycles (Def. 5), we do essentially the same as via the progress LTL property, as Th. 1 shows (see [15] for the proof and details). This insight led to the proposal of dedicates algorithms in [15,11] (cf. DFS_{FIFO} in Sec. 3), requiring $|\mathcal{R}_{\mathcal{M}}|$ time units to prove livelock freedom. The automaton $\mathcal{A}_{\neg\Box\Diamond\mathsf{progress}}$ consists of exactly two states [15], hence $|\mathcal{R}_{\mathcal{M}}| \cdot 2 \leq |\mathcal{R}_{\mathcal{M}\times\varphi}|$. This, combined with the revisits of the NDFS algorithm, *makes the LTL approach up to 4 times as costly as* DFS_{FIFO}.

Partial order reduction. To achieve the reduction as discussed in the introduction, POR replaces the *post* with an *ample* function, which computes a sufficient subset of *post* to explore only relevant interleavings w.r.t the property [16].

For deadlock detection, *ample* only needs to fulfill the *emptiness proviso* and *dependency proviso* (Table 1). The provisos can be deduced locally from s, $post(s)$, and dependency relations $D \subseteq \Sigma_{\mathcal{M}} \times \Sigma_{\mathcal{M}}$ that can be statically overestimated from $\mathcal{M}$, e.g. $(\alpha, \beta) \in D$ if α writes to those variables that β uses as guard [23]. For a precise definition of D consult [16,26].

In general, the model checking of an LTL property (or invariant) φ requires two additional provisos to hold: the *visibility proviso* ensures that traces included in $\mathcal{A}_{\neg\varphi}$ are not pruned from $\mathcal{A}_{\mathcal{M}}$, the *cycle proviso* prevents the so-called *ignoring problem* [9]. The strong variant **C3** (stronger than A4 in [1, Sec. 8.2.2]) is already hard to enforce, so often an even stronger condition, e.g. **C3'**, is implemented. While visibility can still be checked locally, the cycle proviso is a global property, that complicates parallelization [2]. Moreover, the NDFS algorithm revisits states, which might cause different ample sets for the same states, because the procedure is non-deterministic [15]. To avoid any resulting redundant explorations, additional bookkeeping is needed to ensure a deterministic ample set.

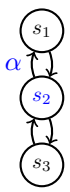
Table 1: POR provisos for the LTL model checking of $\mathcal{M}$ with a property φ

C0	*emptiness*	$ample(s) = \emptyset \Leftrightarrow post(s) = \emptyset$
C1	*dependency*	No action $\alpha \notin ample(s)$ that is dependent on another $\beta \in ample(s)$, i.e. $(\alpha, \beta) \in D$, can be executed in the original $\mathcal{A}_\mathcal{M}$ after reaching the state s and before some action in $ample(s)$ is executed.
C2	*visibility*	$ample(s) \neq post(s) \implies \forall \alpha \in ample(s) : \alpha$ is *invisible*, which means that α does not change a state label referred to by φ.
C3	*cycle*	For a cycle π in $\mathcal{A}_\mathcal{M}$, $\exists s \in \pi$: $post(s) = ample(s)$.
C3'	*cycle (impl.)*	$ample(s) \neq post(s) \Rightarrow \nexists \alpha \in ample(s)$ s.t. $\alpha(s)$ is on the DFS stack.

3 Progress Transitions and DFS$_{\text{FIFO}}$ for Non-Progress

In the current section, we refine the definition of progress to include transitions. We then present a new version of DFS$_{\text{FIFO}}$, an efficient algorithm for non-progress detection by Faragó et al. [11], which supports this broader definition. We also discuss implementation considerations and the combination with POR.

Progress transitions. As argued in [11], progress is more naturally defined on transitions (Def. 6) than on states. After all, the action itself, e.g. the increase of a counter in $\mathcal{M}$, constitutes the actual progress. This becomes clear considering the semantical difference between progress transitions and progress states for livelock detection: The figure on the right shows an automaton with $\mathcal{S}^\mathcal{P} = \{s_1\}$ and $\mathcal{T}^\mathcal{P} = \{(s_2, \alpha, s_1)\}$. Thus the cycle $s_2 \leftrightarrow s_3$ exhibits only *fake progress* when progress states are used ($\mathcal{P} = \mathcal{S}^\mathcal{P}$): the action performing the progress, α, is never taken. With progress transitions ($\mathcal{P} = \mathcal{T}^\mathcal{P}$), only $s_2 \leftrightarrow s_3$ can be detected as **NPcycle**. While fake progress cycles could be hidden by enforcing strong (A-)fairness [1], Spin's weak (A-)fairness [12] is insufficient [11]. But enforcing any kind of fairness is costly [1].



Definition 6 (Progress transitions/actions). *We define progress transitions as:* $\mathcal{T}^\mathcal{P} = \{(s, \alpha, s') \in \mathcal{T} \mid \alpha \in \Sigma^\mathcal{P}\}$*, with* $\Sigma^\mathcal{P} \subseteq \Sigma$ *a set of progress actions.*

Theorem 2. DFS$_{\text{FIFO}}$ *ensures:* $\mathcal{R} \cap \mathcal{NP} \neq \emptyset \Leftrightarrow$ *dfs-fifo*$(s_0) =$ ***report NPcycle***

DFS$_{\text{FIFO}}$. Alg. 1 shows an adaptation of DFS$_{\text{FIFO}}$ that supports the definition of progress on both states and transitions (actions), so $\mathcal{P} = \mathcal{S}^\mathcal{P} \cup \Sigma^\mathcal{P}$. Intuitively, the algorithm works by delaying progress as long as possible using a BFS and searching for **NPcycles** in between progress using a DFS. The correctness of this adapted algorithm follows from Th. 2, which is implied by Th. 4 with $P = 1$.

The FIFO queue F holds progress states, or immediate successors of progress transitions (which we will collectively refer to as after-progress states), with the exception of the initial state s_0. The outer *dfs-fifo* loop handles all after-progress states in breadth-first order. The *dfs* procedure, starting from a state in F then explores states up to progress, storing visited states in the set V (l.22), and after-progress states in F (l.21). The stack of this search is maintained in a set S (l.13 and l.23) to detect cycles at l.16. All states $t \in S$ and their connecting transitions are non-progress by l.18, except for possibly the starting state from F. next page $\longrightarrow$

Algorithm 1 DFS_{FIFO} for progress transitions and progress states

```
1:  procedure dfs-fifo(s_0)
2:      F := {s_0}                    ▷Frontier queue
3:      V := ∅                        ▷Visited set
4:      S := ∅                        ▷Stack
5:      repeat
6:          s := some s ∈ F
7:          if s ∉ V then
8:              dfs(s)
9:          F := F \ {s}
10:     until F = ∅
11:     report progress ensured
12: procedure dfs(s)
13:     S := S ∪ {s}
14:     for all t := α(s) s.t. α ∈ post(s) do
15:         if t ∈ S ∧ α, t ∉ P then
16:             report NPcycle
17:         if t ∉ V then
18:             if α, t ∉ P then
19:                 dfs(t)
20:             else if t ∉ F then
21:                 F := F ∪ {t}
22:     V := V ∪ {s}
23:     S := S \ {s}
```

The cycle-closing transition $s \xrightarrow{\alpha} t$ might also be a progress transition. Therefore, l.15 performs an additional check $\alpha, t \notin \mathcal{P}$. Furthermore, an after-progress state $s \notin \mathcal{S}^{\mathcal{P}}$ added to F, might be reached later via a non-progress path and added to V. Hence, we discard visited states in *dfs-fifo* at l.7.

Implementation. An efficient implementation of Alg. 1 stores F and V in one hash table (using a bit to distinguish the two) for fast inclusion checks, while F is also maintained as a queue F^q. S can be stored in a separate hash table as $|S| \ll |\mathcal{R}|$. Counterexamples can be reconstructed if for each state a pointer to one of its predecessors is stored [20]. Faragó et al. showed two alternatives [11], which are also compatible with lossy hashing [4].

Combination with POR. While the four-fold performance increase of DFS_{FIFO} compared to LTL (Sec. 2) is a modest gain, the algorithm provides even more potential as it relaxes conditions on POR, which, after all, might yield exponential gains. In contrast to the LTL method using NDFS, DFS_{FIFO} does not revisit states, simplifying the *ample* implementation. Moreover, Lemma 1 shows that DFS_{FIFO} does not require the cycle proviso using a visibility proviso from Table 2.

Lemma 1. *Under* $\mathcal{P} = \mathcal{S}^{\mathcal{P}}$, ***C2***$^{\mathcal{S}}$ *implies* ***C3****. Under* $\mathcal{P} = \Sigma^{\mathcal{P}}$, ***C2***$^{\mathcal{T}}$ *implies* ***C3****.*

Proof. If DFS_{FIFO} with POR traverses a cycle C which makes progress, i.e. $\exists s \in C\colon s \in \mathcal{S}^{\mathcal{P}} \vee \mathit{ample}(s) \cap C \cap \Sigma^{\mathcal{P}} \neq \emptyset$, $\mathbf{C2}^{\mathcal{S}}$/ $\mathbf{C2}^{\mathcal{T}}$ guarantees full expansion of s, thus fulfilling **C3**. If DFS_{FIFO} traverses a **NPcycle**, it terminates at l.16. □

Theorem 3. *Th. 2 still holds for* DFS_{FIFO} *with* ***C0****,* ***C1****,* ***C2***$^{\mathcal{S}}$ */* ***C2***$^{\mathcal{T}}$*.*

Proof. Lemma 1 shows that if the **C0**, **C1** and $\mathbf{C2}^{\mathcal{S}}$/ $\mathbf{C2}^{\mathcal{T}}$ hold, so does **C3**. Furthermore, **C0**, **C1** and $\mathbf{C2}^{\mathcal{S}}$/ $\mathbf{C2}^{\mathcal{T}}$ are independent of the path leading to s, so $\mathit{ample}(s)$ with DFS_{FIFO} retains stutter equivalence related to progress [14, p.6]. Therefore, the reduced state space has a **NPcycle** iff the original has one. □

Table 2: POR visibility provisos for DFS_{FIFO}

$\mathbf{C2}^{\mathcal{S}}$	$\mathit{ample}(s) \neq \mathit{post}(s) \implies s \notin \mathcal{S}^{\mathcal{P}}$
$\mathbf{C2}^{\mathcal{T}}$	$\mathit{ample}(s) \neq \mathit{post}(s) \implies \forall \alpha \in \mathit{ample}(s) : \alpha \notin \Sigma^{\mathcal{P}}$

4 A Parallel Livelock Algorithm based on DFS_{FIFO}

Alg. 2 presents a parallel version of DFS_{FIFO}. The algorithm does not differ much from Alg. 1: the *dfs* procedure remains largely the same, and only *dfs-fifo* is split into parallel *fifo* procedures handling states from the FIFO queue F concurrently. The technique to parallelize the $dfs(s, i)$ calls is based on successful multi-core NDFS algorithms [17,19,8]. Each worker thread $i \in 1..P$ uses a local stack S_i, while V and F are shared (below, we show how an efficient implementation can partially localize F). The stacks may overlap (see l.2 and l.9), but eventually diverge because we use a randomized next-state function: $post_i$ (see l.15).

Proof of Correctness. Th. 4 proves correctness of Alg. 2. We show that the propositions below hold after initialization of Alg. 2, and inductively that they are maintained by execution of each statement in the algorithm, considering only the lines that influence the proposition. Rather than restricting progress to either transitions or states, we prove the algorithm correct under $\mathcal{P} = \mathcal{S}^{\mathcal{P}} \cup \mathcal{T}^{\mathcal{P}}$. Hence, the dual interpretation of paths (see Def. 1) is used now and then. Note that a call to **report** terminates the algorithm and the callee does not return.

Lemma 2. *Upon return of $dfs(s, i)$, s is visited: $s \in V$.*

Proof. l.23 of $dfs(s, i)$ adds s to V. □

Lemma 3. *Invariantly, all direct successors of a visited state v are visited or in F: $\forall v \in V, \alpha \in post(v) : \alpha(v) \in V \cup F$.*

Proof. After initialization, the invariant holds trivially, as V is empty. V is only modified at l.23, where s is added after all its immediate successors t are considered at l.16–22: If $t \in V \cup F$, we are done. Otherwise, $dfs(s, i)$ terminates at l.17 or t is added to V at l.20 (Lemma 2) or to F at l.22. States are removed from F at l.12, but only after being added to V at l.11 (Lemma 2). □

Corollary 1. *Lemma 3 holds also for a state $v \notin V$ in $dfs(v, i)$ just before l.23.*

Algorithm 2 Parallel DFS_{FIFO} ($\text{PDFS}_{\text{FIFO}}$)

```
1: procedure dfs-fifo(s_0, P)
2:     F := {s_0}                          ▷Frontier queue
3:     V := ∅                              ▷Visited set
4:     S_i := ∅ for all i ∈ 1..P           ▷Stacks
5:     fifo(1) ‖ ... ‖ fifo(P)
6:     report progress ensured
7: procedure fifo(i)
8:     while F ≠ ∅ do
9:         s := some s ∈ F
10:        if s ∉ V then
11:            dfs(s, i)
12:        F := F \ {s}
13: procedure dfs(s, i)
14:    S_i := S_i ∪ {s}
15:    for all t := α(s) s.t. α ∈ post_i(s) do
16:        if t ∈ S_i ∧ α, t ∉ P then
17:            report NPcycle
18:        if t ∉ V then
19:            if α, t ∉ P then
20:                dfs(t, i)
21:            else if t ∉ F then
22:                F := F ∪ {t}
23:    V := V ∪ {s}
24:    S_i := S_i \ {s}
```

Lemma 4. *Invariantly, all paths from a visited state v to a state $f \in F \setminus V$ contain progress:* $\forall \pi, v \in V, f \in F \setminus V : v \xrightarrow{\pi} f \implies \mathcal{P} \cap \pi \neq \emptyset$.

Proof. After initialization of the sets V and F, the lemma is trivially true. These sets are modified at l.12, l.22, and l.23 (omitting the trivial case):

l.22 Let i be the first worker thread to add a state t to F in $dfs(s, i)$ at l.22. If some other worker j adds t to V, the invariant holds trivially, so we consider $t \notin V$. By l.19, all paths $v \to^* s \to t$ contain progress. By contradiction, we show that all other paths that do not contain s also contain progress: Assume that there is a $v \in V$ such that $v \xrightarrow{\pi} {}^+ t$ and $\mathcal{P} \cap \pi = \emptyset$. By induction on the length of the path π and Lemma 3, we obtain either $t \in V$, a contradiction, $t \in F \setminus V$, contradicting the assumption that worker i is first, or another $f \neq t$ with $f \in F \setminus V$, for which the induction hypothesis holds.

l.23 Assume towards a contradiction that i is the first worker thread to add a state s to V at l.23 of $dfs(s, i)$. So, we have $s \notin V$ before l.23. By Cor. 1, for all immediate successors t of s, i.e. for all $t = \alpha(s)$ such that $\alpha \in post(s)$, we have $t \in V$ or $t \in F \setminus V$. In the first case, since $s \neq t$, the induction hypothesis holds for t. In the second case, if $t = s$, the invariant trivially holds after l.23, and if $t \neq s$, we have $\alpha, t \in \mathcal{P}$, since otherwise $t \in V$ by l.19 and l.20 (Lemma 2). Thus the invariant holds for all paths $s \to^+ f$. □

Remark 1. Note that a state $s \in F$ might at any time be also added to V by some other worker thread in two cases: (1) $s \notin \mathcal{S}^{\mathcal{P}}$, i.e. it was reached via a progress transition (see l.19), but is reachable via some other non-progress path, or (2) another worker thread j takes s from F at l.9 and completes $dfs(s, j)$.

Lemma 5. *Invariantly, visited states do not lie on* ***NPcycles****:* $V \cap \mathcal{NP} = \emptyset$.

Proof. Initially, $V = \emptyset$ and the lemma holds trivially. Let i be the first worker thread to add s to V in $dfs(s, i)$ at l.23. So we have $s \in V$ just after l.23 of $dfs(s, i)$. Assume towards a contradiction that $s \in \mathcal{NP}$. Then there is a **NPcycle** $s \to t \to^+ s$ with $s \neq t$ since otherwise l.17 would have reported a **NPcycle**. Now by Lemma 3, $t \in V \cup F$. By the induction hypothesis, $t \notin V$, so $t \in F \setminus V$. Lemma 4 contradicts $s \to t$ making no progress. □

Lemma 6. *Upon return of dfs-fifo, all reachable states are visited:* $\mathcal{R} \subseteq V$.

Proof. After *dfs-fifo*(s_0, P), $F = \emptyset$ by l.8. By l.2, l.11 and Lemma 2, $s_0 \in V$. So by Lemma 3, $\mathcal{R} \subseteq V$. □

Lemma 7. *dfs-fifo terminates and reports an* ***NPcycle*** *or* ***progress ensured****.*

Proof. Upon return of a call $dfs(s, i)$ for some $s \in F$ at l.11, s has been added to V (Lemma 2), removed from F at l.12, and will never be added to F again. Hence the set V grows monotonically, but is bounded, and eventually $F = \emptyset$. Thus eventually all *dfs* calls terminate, and *dfs-fifo*(s_0, P) terminates too. □

Lemma 8. *Invariantly, the states in S_i form a path without progress except for the first state:* $S_i = \emptyset$ *or* $S_i = \pi \cap \mathcal{S}$ *for some* $s \xrightarrow{\pi} {}^* s'$ *and* $\pi \cap \mathcal{P} \subseteq \{s_1\}$.

Proof. By induction over the recursive $dfs(s, i)$ calls, we obtain π. At l.20, we have $\alpha, t \notin \mathcal{P}$, but at l.11 we may have $s \in \mathcal{S}^{\mathcal{P}}$ (by l.19 and l.22). □

Theorem 4. PDFS$_{\text{FIFO}}$ *ensures:* $\mathcal{R} \cap \mathcal{NP} \neq \emptyset \Leftrightarrow$ *dfs-fifo*$(s_0, P) =$ ***report NPcycle***

Proof. We split the equivalence into two cases:
$\Leftarrow$: We have a cycle: $s \xrightarrow{\alpha} t \xrightarrow{\pi} s$ s.t. $(\{\alpha\} \cup \pi) \cap \mathcal{P} = \emptyset$ by l.16 and Lemma 8.
$\Rightarrow$: Assume that *dfs-fifo*$(s_0, P) \neq$ **NPcycle** $\wedge\, \mathcal{R} \cap \mathcal{NP} \neq \emptyset$. However, at l.6, $\mathcal{R} \subseteq V$ by Lemma 6 and Lemma 7, hence $\mathcal{R} \cap \mathcal{NP} = \emptyset$ by Lemma 5. □

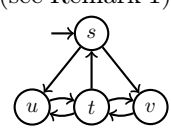
Implementation. For a scaling implementation, the hash table storing F and V (see Sec. 3) is maintained in shared memory using a lockless design [20,18]. Storing also the queue F^q in shared memory, however, would seriously impede scalability due to contention (recall that F is maintained as both hash table and queue F^q). Our more efficient implementation splits F^q into P local queues F_i^q, such that $F \subseteq \bigcup_{i \in 1..P} F_i^q$ (Remark 1 explains the $\subseteq$).

To implement load balancing, one could relax the constraint at l.21 to $s \notin F^q$, so that after-progress states end up on multiple local queues. Provided that $\mathcal{A}_\mathcal{M}$ is connected enough, which it usually is in model checking, this would provide good work distribution already. On the other hand, the total size of all queues F_i^q would grow proportional to P, wasting a lot of memory on many cores. Therefore, we instead opted to add explicit load balancing via work stealing. The code on the left illustrates this. Iff the local queue F_i^q is empty, the *steal* function grabs states from another random queue F_j^q and adds them to F_i^q, returning false iff it detects termination. Inspection of Lemma 3 and Lemma 7 shows that removing s from F is not necessary.

```
1: procedure fifo(i)
2:     F_i^q := {s_0}
3:     while steal(F_i^q) do
4:         F_i^q := F_i^q \ {s}
5:         if s ∉ V then
6:             dfs(s, i)
```

The proofs show that correctness of PDFS$_{\text{FIFO}}$ does not require F to be in strict FIFO order (as l.9 does not enforce any order). To optimize for scalability, we enforce a strict BFS order via synchronizations[4] between the BFS levels only optionally[5]. As trade-off, counterexamples are no longer guaranteed to be the shortest with respect to progress, and the size of F may increase (see Remark 1).

Analysis of scalability. Experiments with multi-core NDFS [8] demonstrated that these parallelization techniques make the state-of-the-art for LTL model checking. Because of the BFS nature of DFS$_{\text{FIFO}}$, we can expect even better speedups. Moreover, in [17], additional synchronization was needed to prevent workers from *early backtracking*; a situation in which two workers exclude a third from part of the state space. The figure on the right illustrates this: Worker 1 can visit s, v, t and u, and then halt. Worker 2 can visit s, u, t and v and backtrack over v. If now Worker 1 resumes and backtracks over u, both v and u are in V. A third worker will be excluded from visiting t, which might lead to a large part of the state space. Lemma 3 shows that this is impossible for PDFS$_{\text{FIFO}}$ as the successors of visited states are either visited or in F (treated in efficient parallel BFS), but never do successors lie solely on the stack S_i (as in CNDFS).



[4] Parallel BFS algorithms, with and without synchronization, are described in [7].
[5] The command line option --strict turns on strict PDFS$_{\text{FIFO}}$ in LTSMIN.

5 Experimental Evaluation

In the current section, we benchmark the performance of DFS_{FIFO}, and its combination with POR, using both progress states and progress transitions. We compare the results against the LTL approach with progress property using, inter alia, SPIN [12]. We also investigate the scalability of $\text{PDFS}_{\text{FIFO}}$, and compare the results against the multi-core NDFS algorithm CNDFS, the state-of-the-art for parallel LTL [8,5], and the piggyback algorithm in SPIN (PB). Finally, we investigate the combination of $\text{PDFS}_{\text{FIFO}}$ and POR, and compare the results with OWCTY [3], which uses a topological sort to implement paralel LTL and POR [2].

We implemented $\text{PDFS}_{\text{FIFO}}$ (*Alg. 2 with work stealing and both strict*[5]*/non-strict BFS order*) in LTSMIN [21] 2.0.[6] LTSMIN has a frontend for PROMELA, called SPINS [12], and one for the DVE language, allowing fair comparison [21,5] against SPIN 6.2.3 and DIVINE 2.5.2 [3]. To ensure similar state counts, we turned off control-flow optimizations in SPINS/SPIN, because SPIN has a more powerful optimizer, which can be, but is not yet implemented in SPINS. Only the GIOP model (described below) still yields a larger state count in SPINS/LTSMIN than in SPIN. We still include it as it nicely features the benefits of DFS_{FIFO} over NDFS.

We benchmarked on a 48-core machine (a four-way AMD Opteron 6168) with 128GB of main memory, and considered 4 publicly available[3] PROMELA models with progress labels, and adapted SPINS to interpret the labels as either progress states, as in SPIN, or progress transitions. leader_t is the efficient leader election protocol $\mathcal{A}_{timing}$ [10]. The *Group Address Registration Protocol* (*GARP*) is a datalink-level multicast protocol for a bridged LAN. *General Inter-Orb Protocol* (GIOP) models service oriented architectures. The model *i-Protocol* represents the GNU implementation of this protocol. We use a different leader election protocol (leader_{DKR}) from [24] for comparison against DIVINE. For all these models, the livelock property holds under $\mathcal{P} = \mathcal{S}^{\mathcal{P}}$ and $\mathcal{P} = \mathcal{T}^{\mathcal{P}}$.[7]

Performance. In theory, DFS_{FIFO} can be up to four times as fast as using the progress LTL formula and NDFS. To verify this, we compare DFS_{FIFO} to NDFS in LTSMIN and SPIN. In LTSMIN, we used the command line: `prom2lts-mc --state=tree -s28 --strategy=[dfsfifo/ndfs] [model]`, which replaces the shared table (for F and V) by a *tree* table for state compression [18]. In SPIN, we used compression as well (*collapse* [12]): `cc -O2 -DNP -DNOFAIR -DNOREDUCE -DNOBOUNDCHECK -DCOLLAPSE -o pan pan.c`, and `pan -m100000 -l -w28`, avoiding table resizes and overhead. In both tools, we also ran DFS reachability with similar commands. We write *oom* for runs that overflow the main memory.

Table 3 shows the results: As expected, $|\mathcal{R}_{\text{LTL}}|$ is 1.5 to 2 times larger than $|\mathcal{R}|$ for both SPIN and LTSMIN; GIOP fits in memory for DFS_{FIFO} but the LTL cross-product overflows (NDFS). T_{NDFS} is about 1.5 to 4 times larger than T_{DFS} for SPIN, 2 to 5 times larger for LTSMIN (cf. Section 2). $T_{\text{DFS}_{\text{FIFO}}}$ is 1.5 to 2 times larger than T_{DFS}, likely caused by its set inclusion tests on S and F. T_{NDFS} is 1.6 to 3.2 times larger than $T_{\text{DFS}_{\text{FIFO}}}$.

[6] LTSMIN is open source, available at: `http://fmt.cs.utwente.nl/tools/ltsmin`.

[7] Models that we modified are available at `http://doiop.com/leader4DFSFIFO`.

Table 3: Runtimes (sec) of (sequential) DFS, DFS_{FIFO}, NDFS in SPIN and LTSMIN

	LTSMIN					SPIN			
	$\lvert\mathcal{R}\rvert$	$\lvert\mathcal{R}_{\text{LTL}}\rvert$	T_{DFS}	$T_{\text{DFS}_{\text{FIFO}}}$	T_{NDFS}	$\lvert\mathcal{R}\rvert$	$\lvert\mathcal{R}_{\text{LTL}}\rvert$	T_{DFS}	T_{NDFS}
leader_t	4.5E7	198%	153.7	233.2	753.6	4.5E7	198%	304.0	1,390.0
garp	1.2E8	150%	377.1	591.2	969.2	1.2E8	146%	1,370.0	2,050.0
giop	2.7E9	oom	21,301.4	43,154.3	oom	8.4E7	181%	1,780.0	4,830.0
i-prot	1.4E7	140%	28.4	41.4	70.6	1.4E7	145%	63.3	103.0

Table 4: Runtimes (sec) / queue sizes of the parallel algorithms: DFS, $\text{PDFS}_{\text{FIFO}}$ and CNDFS in LTSMIN, and PB in SPIN

	DFS		$\text{PDFS}_{\text{FIFO}}$		CNDFS		PB		$\text{PDFS}_{\text{FIFO}}^{strict}$		$\text{PDFS}_{\text{FIFO}}^{non\text{-}strict}$		CNDFS	
	T_1	T_{48}	T_1	T_{48}	T_1	T_{48}	T_1	T_{min}	Q_1	Q_{48}	Q_1	Q_{48}	Q_1	Q_{48}
leader_t	153.7	3.8	233.2	5.7	925.7	51.4	228.0	25.9	1.0E6	1.2E6	1.2E6	1.4E6	2.7E6	3.6E7
garp	377.1	8.8	591.2	13.1	1061.0	58.6	1180.0	70.9	1.9E7	2.0E7	1.9E7	5.3E6	5.5E6	6.5E7
giop	2.1E4	463.3	4.3E4	9.7E2	oom	oom	1.2E3	57.8	1.1E9	8.4E8	1.1E9	8.4E8	oom	oom
i-prot	28.4	0.7	41.4	1.1	75.9	3.7	86.2	17.7	1.0E6	1.1E6	1.0E6	1.3E6	8.3E5	1.0E7

Table 5: POR (%) for $\text{DFS}_{\text{FIFO}}^{\mathcal{T}}$, $\text{DFS}_{\text{FIFO}}^{\mathcal{S}}$, DFS and NDFS in SPIN and LTSMIN

	LTSMIN				SPIN	
	DFS	$\text{DFS}_{\text{FIFO}}^{\mathcal{T}}$	$\text{DFS}_{\text{FIFO}}^{\mathcal{S}}$	NDFS	DFS	$\text{NDFS}^{\text{SPIN}}$
leader_t	0.32%	0.49%	99.99%	99.99%	0.03%	1.15%
garp	1.90%	2.18%	4.29%	16.92%	10.56%	12.73%
giop	1.86%	1.86%	3.77%	oom	1.60%	2.42%
i-prot	16.14%	31.83%	100.00%	100.00%	24.01%	41.37%

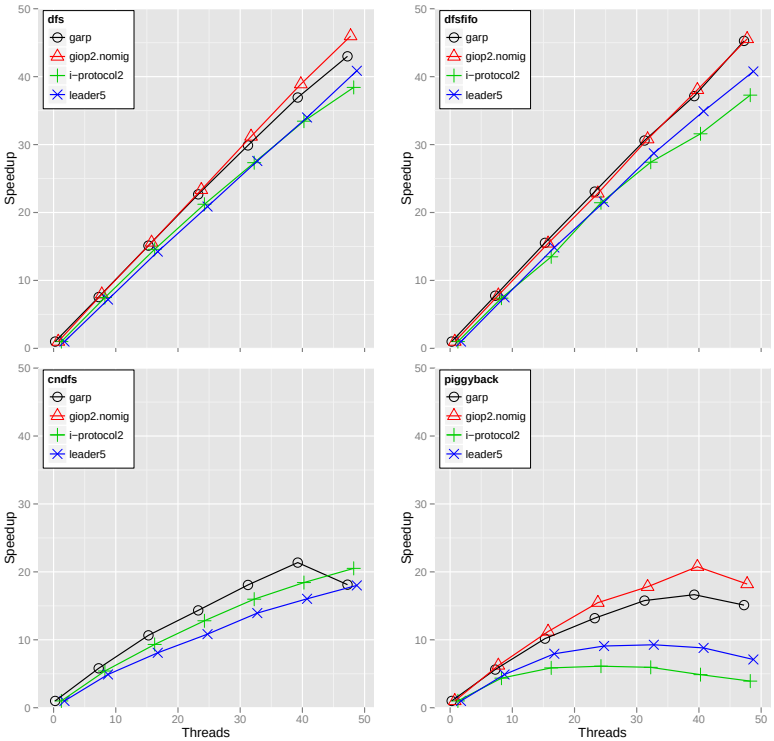


Fig. 1: Speedups of DFS, $\text{PDFS}_{\text{FIFO}}$ and CNDFS in LTSMIN, and piggyback in SPIN

Table 6: POR and speedups for leader$_{DKR}$ using PDFS$_{\text{FIFO}}$, CNDFS and OWCTY

N	Alg.	$\lvert\mathcal{R}\rvert$	$\lvert\mathcal{T}\rvert$	T_1	T_{48}	U	$\lvert\mathcal{R}^{\text{POR}}\rvert$	$\lvert\mathcal{T}^{\text{POR}}\rvert$	T_1^{POR}	T_{48}^{POR}	U^{POR}
9	CNDFS	3.6E7	2.3E8	502.6	12.0	41.8	27.9%	0.1%	211.8	n/a	n/a
9	PDFS$_{\text{FIFO}}$	3.6E7	2.3E8	583.6	14.3	40.8	1.5%	0.0%	12.9	3.6	3.5
9	OWCTY	3.6E7	2.3E8	498.7	51.9	9.6	12.6%	0.0%	578.4	35.7	16.2
10	CNDFS	2.4E8	1.7E9	~~30'~~	90.7	~~30'~~	19.3%	5.4%	1102.7	n/a	n/a
10	PDFS$_{\text{FIFO}}$	2.4E8	1.7E9	~~30'~~	109.3	~~30'~~	0.7%	0.1%	35.0	2.5	14.0
10	OWCTY	2.4E8	1.7E9	~~30'~~	663.1	~~30'~~	8.7%	2.2%	~~30'~~	156.3	~~30'~~
11	PDFS$_{\text{FIFO}}$	~~30'~~	~~30'~~	~~30'~~	~~30'~~	~~30'~~	5.1E6	7.1E6	109.8	5.3	20.7
11	OWCTY	~~30'~~	~~30'~~	~~30'~~	~~30'~~	~~30'~~	9.3E7	1.7E8	~~30'~~	1036.5	~~30'~~
12	PDFS$_{\text{FIFO}}$	~~30'~~	~~30'~~	~~30'~~	~~30'~~	~~30'~~	1.6E7	2.2E7	369.1	11.2	33.0
13	PDFS$_{\text{FIFO}}$	~~30'~~	~~30'~~	~~30'~~	~~30'~~	~~30'~~	6.6E7	9.2E7	1640.5	38.1	43.0
14	PDFS$_{\text{FIFO}}$	~~30'~~	~~30'~~	~~30'~~	~~30'~~	~~30'~~	2.0E8	2.9E8	~~30'~~	120.3	~~30'~~
15	PDFS$_{\text{FIFO}}$	~~30'~~	~~30'~~	~~30'~~	~~30'~~	~~30'~~	8.4E8	1.2E9	~~30'~~	527.5	~~30'~~

Parallel scalability. To compare the parallel algorithms in LTSMIN, we use the options --threads=P --strategy=[dfsfifo/cndfs], where P is the number of worker threads. In SPIN, we use -DBFS_PAR, which also turns on lossy state hashing [13], and run the pan binary with an option -uP. This turns on a parallel, linear-time, but incomplete, cycle detection algorithm called piggyback (PB) [13]. It might also be unsound due its combination with lossy hashing [4]. Fig. 1 shows the obtained speedups: As expected, reachability [20] and PDFS$_{\text{FIFO}}$ scale almost ideally, while CNDFS exhibits sub-linear scalability, even though it is the fastest parallel LTL solution [8]. PB also scales sub-linearly. Since LTSMIN sequentially competes with SPIN (Table 4, except for GIOP), scalability can be compared.

Parallel memory use. We expected few state duplication in F on local queues (see Remark 1). To verify this, we measured the total size of all local queues and hash tables using counters for strict[5] and non-strict PDFS$_{\text{FIFO}}$, and CNDFS. Table 4 shows $Q_P = \sum_{i \in 1..P} |F_i^q| + |S_i|$ averaged over 5 runs: Non-strict PDFS$_{\text{FIFO}}$ shows little difference from the strict variant, and Q_{48} is at most 20% larger than Q_1 for all PDFS$_{\text{FIFO}}$. Due to the randomness of the parallel runs, we even have $Q_{48} < Q_1$ in many cases. Revisits occurred at most 2.6% using 48 cores. In the case of CNDFS, the combined stacks typically grow because of the larger DFS searches. Accordingly, we found that PDFS$_{\text{FIFO}}$'s *total memory use with 48 cores* was between 87% and 125% compared to sequential DFS. In the worst case, PDFS$_{\text{FIFO}}$ (with tree compression) used 52% of the memory use of PB (collapse compression and lossy hashing) [18,5] – GIOP excluded as its state counts differ.

POR performance. LTSMIN's POR implementation (option --por) is based on stubborn sets [26], described in [23], and is competitive to SPIN's [5]. We extended it with the alternative provisos for DFS$_{\text{FIFO}}$: $\mathbf{C2}^{\mathcal{S}}$ and $\mathbf{C2}^{\mathcal{T}}$. Table 5 shows the relative number of states, using the different algorithms in both tools: For all models, both LTSMIN and SPIN are able to obtain reductions of multiple orders of magnitude using their DFS algorithms. We also observe that much of this benefit disappears when using the NDFS LTL algorithm due to the cycle proviso, although SPIN often performs much better than LTSMIN in this respect. Also DFS$_{\text{FIFO}}$ with progress states (column DFS$^{\mathcal{S}}_{\text{FIFO}}$), performs poorly: apparently, the $\mathbf{C2}^{\mathcal{S}}$ proviso is so restrictive that many states are fully expanded. But DFS$_{\text{FIFO}}$ with progress transitions (column DFS$^{\mathcal{T}}_{\text{FIFO}}$) retains DFS's impressive POR with at most a factor 2 difference.

Scalability of parallelism and POR. We created multiple instances of the leader$_{DKR}$ models by varying the number of nodes N and expressed the progress LTL property in DiVinE. We start DiVinE's state-of-the-art parallel LTL-POR algorithm, OWCTY, by: divine owcty [model] -wP -i30 -p. With the options described above, we turned on POR in LTSmin and ran PDFS$_{\text{FIFO}}$, and CNDFS, for comparison. We limited each run to half an hour (~~30'~~ indicates a timeout). Piggyback reported contradictory memory usage and far fewer states (e.g. <1%) compared to DFS with POR, although it must meet more provisos. Thus we did not compare against piggyback and suspect a bug.

Table 6 shows that PDFS$_{\text{FIFO}}$ and POR complement each other rather well: Without POR (left half of the table) the almost ideal speedup ($U = \frac{T_1}{T_{48}} = 40.8$) allows to explore one model more: $N \leq 10$ instead of only $N = 9$. When enabling POR (right half of the table), we see again multiple orders of magnitude reductions, while parallel scalability reduces to $U = 3.5$ for $N = 9$, because of the small size of the reduced state space ($|\mathcal{R}^{\text{POR}}|$). When increasing the model size to $N = 13$ the speedup grows again to an almost ideal level ($U = 43$). With POR, the parallelism allows us to explore two more models within half an hour, i.e., $N \leq 15$. While OWCTY and NDFS also show this effect, it is less pronounced due to their cycle proviso, allowing $N \leq 11$ for OWCTY and $N \leq 9$ for NDFS.

As livelocks are disjoint from the class of weak LTL properties, OWCTY could become non-linear [3], but it required only one iteration for leader$_{DKR}$.

As PDFS$_{\text{FIFO}}$ revisits states, the random next-state function could theoretically weaken POR (as for NDFS, see Sec. 2). But for all our 5 models, this did not occur.

On-the-fly performance. We created a leader election protocol with early (*shallow*) and another with late (*deep*) injected **NPcycles** (see [7], [10]).

	CNDFS		PDFS$^{\mathcal{T}}_{\text{FIFO}}$		CNDFS		PDFS$^{\mathcal{T}}_{\text{FIFO}}$	
	T_1	T_{48}	T_1	T_{48}	C_1	C_{48}	C_1	C_{48}
shallow	~~30'~~	7	12	4	~~30'~~	16	16	16
deep	16 (once ~~30'~~)	2	~~30'~~	451	577	499	~~30'~~	51

The table on the right shows the average runtime in seconds (T) and counterexample length (C) over five runs. Since PDFS$_{\text{FIFO}}$ finds shortest counterexamples[5], it outperforms CNDFS for *shallow* (more relevant in practice) and pays a penalty for *deep*. Both algorithms benefit greatly from massive parallelism (see also [19]).

6 Conclusions

We showed, in theory and in practice, that model checking livelocks, an important subset of liveness properties, can be made more efficient by specializing on it. For our PDFS$_{\text{FIFO}}$ implementation with progress transitions, POR becomes significantly stronger (cf. Table 5), parallelization has linear speedup (cf. Fig. 1), and both can be combined efficiently (cf. Table 6).

Acknowledgements. We thank colleagues Mark Timmer, Mads Chr. Olesen, Christoph Scheben and Tom van Dijk for their useful comments on this paper.

References

1. C. Baier and J.-P. Katoen. *Principles of Model Checking.* The MIT Press, 2008.

2. J. Barnat, L. Brim, and P. Rockai. Parallel Partial Order Reduction with Topological Sort Proviso. In *SEFM*, pages 222–231. IEEE Computer Society, 2010.
3. J. Barnat, L. Brim, and P. Ročkai. A Time-Optimal On-the-Fly Parallel Algorithm for Model Checking of Weak LTL Properties. In *ICFEM 2009*, volume 5885 of *LNCS*, pages 407–425. Springer, 2009.
4. J. Barnat, J. Havlíček, and P. Ročkai. Distributed LTL Model Checking with Hash Compaction. In *PASM/PDMC*, ENTCS. Elsevier, 2012.
5. F. v. d. Berg and A. Laarman. SpinS: Extending LTSmin with Promela through SpinJa. In *PASM/PDMC*, ENTCS. Elsevier, 2012.
6. C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory-Efficient Algorithms for the Verification of Temporal Properties. *FMSD*, 1(2):275–288, 1992.
7. A. Dalsgaard, A. Laarman, K. Larsen, M. Olesen, and J. van de Pol. Multi-Core Reachability for Timed Automata. In *Formats*, LNCS 7595. Springer, 2012.
8. S. Evangelista, A. Laarman, L. Petrucci, and J. van de Pol. Improved Multi-core NDFS. In *ATVA*, volume 7561 of *LNCS*, pages 269–283. Springer, 2012.
9. S. Evangelista and C. Pajault. Solving the Ignoring Problem for Partial Order Reduction. *STTF*, 12:155–170, 2010.
10. D. Faragó. Model Checking of Randomized Leader Election Algorithms. Master's thesis, Universität Karlsruhe, 2007.
11. D. Faragó and P. Schmitt. Improving Non-Progress Cycle Checks. In *SPIN*, volume 5578 of *LNCS*, pages 50–67. Springer, 2009.
12. G. Holzmann. *The* SPIN *Model Checker: Primer&Ref. Man.* Addison-Wesley, 2011.
13. G. Holzmann. Parallelizing the Spin Model Checker. In *SPIN*, volume 7385 of *LNCS*, pages 155–171. Springer, 2012.
14. G. Holzmann and D. Peled. An Improvement in Formal Verification. In *Proceedings of the Formal Description Techniques*, pages 197–211. Chapman & Hall, 1994.
15. G. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In *SPIN*, pages 23–32. American Mathematical Society, 1996.
16. S. Katz and D. Peled. An Efficient Verification Method for Parallel and Distributed Programs. In *REX workshop*, volume 398 of *LNCS*, pages 489–507. Springer, 1988.
17. A. Laarman, R. Langerak, J. van de Pol, M. Weber, and A. Wijs. Multi-Core NDFS. In *ATVA*, volume 6996 of *LNCS*, pages 321–335. Springer, 2011.
18. A. Laarman, J. v. d. Pol, and M. Weber. Parallel Recursive State Compression for Free. In *SPIN*, LNCS, pages 38–56. Springer, 2011.
19. A. Laarman and J. van de Pol. Variations on Multi-Core Nested Depth-First Search. In *PDMC*, volume 72 of *EPTCS*, pages 13–28, 2011.
20. A. Laarman, J. van de Pol, and M. Weber. Boosting Multi-Core Reachability Performance with Shared Hash Tables. In *FMCAD*. IEEE Computer Society, 2010.
21. A. Laarman, J. van de Pol, and M. Weber. Multi-Core LTSmin: Marrying Modularity and Scalability. In *NFM*, LNCS 6617, pages 506–511. Springer, 2011.
22. G. Moore. Cramming more Components onto Integrated Circuits. *Electronics*, 38(10):114–117, 1965.
23. E. Pater. Partial Order Reduction for PINS, Master's thesis. Uni. of Twente, 2011.
24. R. Pelánek. BEEM: Benchmarks for Explicit Model Checkers. In *SPIN*, volume 4595 of *LNCS*, pages 263–267. Springer, 2007.
25. R. Saad, S. Zilio, and B. Berthomieu. An experiment on parallel model checking of a ctl fragment. In *ATVA*, volume 7561 of *LNCS*, pages 284–299. Springer, '12.
26. A. Valmari. Stubborn Sets for Reduced State Space Generation. In *APN*, volume 483 of *LNCS*, pages 491–515. Springer, 1991.
27. M. Vardi and P. Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *LICS*, pages 332–344, 1986.