Improving the Scalability of Automatic Linearizability Checking in SPIN

Patrick Doolan[1], Graeme Smith[2], Chenyi Zhang[3(✉)], and Padmanabhan Krishnan[1]

[1] Oracle Labs, Brisbane, Australia
[2] The University of Queensland, Brisbane, Australia
[3] Jinan University, Guangzhou, China
chenyi_zhang@jnu.edu.cn

Abstract. Concurrency in data structures is crucial to the performance of multithreaded programs in shared-memory multiprocessor environments. However, greater concurrency also increases the difficulty of verifying correctness of the data structure. Model checking has been used for verifying concurrent data structures satisfy the correctness condition 'linearizability'. In particular, 'automatic' tools achieve verification without requiring user-specified linearization points. This has several advantages, but is generally not scalable. We examine the automatic checking used by Vechev et al. in their 2009 work to understand the scalability issues of automatic checking in SPIN. We then describe a new, more scalable automatic technique based on these insights, and present the results of a proof-of-concept implementation.

1 Introduction

How efficiently data structures are shared is a crucial factor in the performance of multithreaded programs in shared-memory multiprocessor environments [14]. This motivates programmers to create objects with fewer safety mechanisms (such as locks) to achieve greater concurrency. However, as noted by [14], any enhancement in the performance of these objects also increases the difficulty of verifying they behave as expected. Several published concurrent data structures – often with manual proofs of correctness – have been shown to contain errors (e.g., [7,18]). This has resulted in a wealth of research on proving the safety of these objects with minimal input from programmers.

To verify concurrent data structures it is necessary to have a suitable definition of correctness. The general consensus of the literature is that linearizability, first introduced in [10], is the appropriate notion of correctness. The definition of linearizability given by Vechev et al. [24] is summarised below.

Definition 1. *A concurrent data structure is* ***linearizable*** *if every concurrent/overlapping history of the data structure's operations has an equivalent sequential history that*

The corresponding author was at Oracle Labs, Australia during the initial stages of this work.

Z. Duan and L. Ong (Eds.): ICFEM 2017, LNCS 10610, pp. 105–121, 2017.
https://doi.org/10.1007/978-3-319-68690-5_7

1. *meets a sequential specification of the data structure, and*
2. *respects the ordering of non-overlapping operations.*

Note that condition (2) is also referred to as the *partial ordering condition*. When discussing linearizability the sequential specification is often referred to as the *abstract specification*, and the implementation of the concurrent data structure the *concrete implementation*. The equivalent sequential history generated from a concurrent history is referred to as the *linearization* or *sequential witness*.

Given a sequential specification, a history can be checked for a linearization. This requires examining permutations of the history to identify whether any one of them is a linearization. This process is called *linearization checking* (not to be confused with the overall process of linearizability checking).

Example 1. Figure 1 shows a history of operations for a concurrent queue. By enumerating all permutations, it can be seen that this history has the linearization [enqueue(1), dequeue() → 1, enqueue(2)].

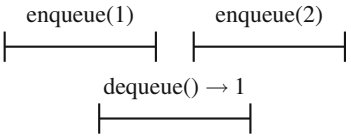

Fig. 1. A sample concurrent history with a linearization.

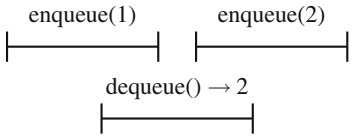

Fig. 2. A sample concurrent history with no linearization.

Conversely, consider Fig. 2, which is also a history of a concurrent queue. This does not have a linearization, because, by the partial ordering condition, enqueue(2) must linearize after enqueue(1). It follows that dequeue() can only correctly return 1 (if it linearizes after enqueue(1)) or 'empty' (if it linearizes before enqueue(1)). No sequential equivalent of this history will satisfy the sequential specification of a queue. This history is in fact a behaviour of the 'buggy queue' from [18].

Linearizability is useful for programmers because it allows them to view a concurrent data structure's operations as happening at a single point in time (called the *linearization point*) [14]. Furthermore, [9] proves that linearizability generally coincides with 'observational refinement', meaning that when a linearizable data structure replaces a correct but sub-optimal data structure, the new program produces a subset of its previous, acceptable behaviour.

In this paper we identify reasons why some of the techniques to verify linearizability are not scalable and present a technique that overcomes some of these hurdles. We also present experimental results to demonstrate the feasibility of our ideas.

1.1 Related Work

There are a wide variety of approaches used to verify linearizability of data structures. These range from manual proofs, possibly with the help of a theorem prover (see [16,22] respectively for examples with and without a theorem prover), to static and runtime analysis (e.g., [23,27], respectively) and model checking (e.g., [4,12,19,24]).

Model checkers give a high degree of automation because they work by exhaustive checking of behaviour, but are limited compared to other approaches because their verification is typically within bounds on the number of threads, arguments and other factors. We distinguish two approaches to model checking linearizability:

- *linearization point-based checking* requires the user to specify the linearization points (see [19] for an example), whereas
- *automatic checking* does not require user specification of linearization points (see [4,12,24]).

The latter has two advantages, viz., greater flexibility for data structures with non-fixed linearization points, and certainty that reported failures are from bugs in the data structure and not incorrectly identified linearization points.

There is a substantial literature on automatic checking which illustrates that many different model checkers and techniques have been used for this purpose. Vechev et al. [24] describe a tool for examining many potential versions of a data structure and determining which are linearizable. To this end they use both automatic and linearization point-based methods in SPIN [11]. They note, importantly, that automatic checking can be used to cull a large number of potential implementations but that its inherent scalability issues make it intractable for thorough checking.

Similarly, Liu et al. [12] use the model checker PAT [17,20] for automatic checking of linearizability. Both the implementation (the concurrent data structure) and the specification (the sequential behaviour) are modelled in the process algebra CSP, and the verification is carried out as checking observational refinement with certain optimizations. The verification process is, generally, automatic checking, though the results can be enhanced if linearization points are known. This result was further improved on by Zhang [28] by combining partial order reduction and symmetry reduction to narrow the potential state space, and in doing so they were able to verify concurrent data structures (albeit simple ones) for three to six threads. In contrast, automatic checking reported by Vechev et al. [24] only allows two threads, though the comparison may not be fair, as SPIN does not have built-in support for symmetry reduction.

Burckhardt et al. [4] describe the tool Line-Up, built on top of the model checker CHESS [15], for automatically checking linearizability of data structures. It is one of the most automated approaches to date; it does not require user-specified linearization points nor an abstract specification of the data structure (a specification is instead automatically extracted from the implementation). It also operates on actual code, as opposed to a model of the code.

The compromise for this convenience, as pointed out by [28], is that Line-Up is "only sound with respect to its inputs". Specifically, a user must specify which sequences of operations Line-Up checks, whereas other model checking techniques generate all possible sequences of operations (within bounds). Line-Up also requires that a specification be deterministic, as otherwise the extracted specification will misrepresent the actual abstract specification.

Regarding the complexities of linearizability checking, the problem has been shown decidable for a special class of concurrent linked-list, by a reduction to reachability of method automata [5]. As an observational refinement problem, checking linearizability is in general undecidable, and it is EXPSPACE-complete even with fixed linearization points [2]. More recently, Bouajjani et al. discovered that for a class of concurrent objects and data structures such as stacks and queues, the linearizability property can be reduced to the control state reachability problem [3].

1.2 Contributions

A notable theme in the related work is that automatic methods are considered to have inherent scalability issues for verification [12,24], though they can be used effectively when limits are placed on types or numbers of operations checked [4,24] or advanced state compression techniques are used [28]. However, the exact causes of the scalability issues are not discussed in detail, and there is some disagreement in the literature.

This paper explores in detail the causes of scalability issues in automatic checking, using the work of Vechev et al. [24] as our starting point. The insights derived are then used to describe a technique for improving the scalability of automatic checking methods using SPIN. Our solution, as currently implemented, is not sound and hence can only be used to find bugs. However, we describe how the technique can be extended to support verification.

The paper is structured as follows. In Sect. 2 we present our analysis of the scalability issues in the work of Vechev et al. [24]. A technique for overcoming these issues is presented in Sect. 3, and the results of applying an implementation of this technique to data structures from the literature with known bugs is described in Sect. 4. Also in Sect. 4 we discuss the main limitation of our technique which restricts it to bug finding, rather than full verification. Section 5 then describes how this limitation can be overcome and how the technique can be integrated into SPIN.

2 Scalability Issues of Automatic Checking with SPIN

To understand the scalability issues of automatic checking in [24], we first describe their methods. We will refer to their approach as using 'global internal recordings' since a (global) list of all invocations and responses of operations by any thread is recorded (internally) as part of the model checker's state.

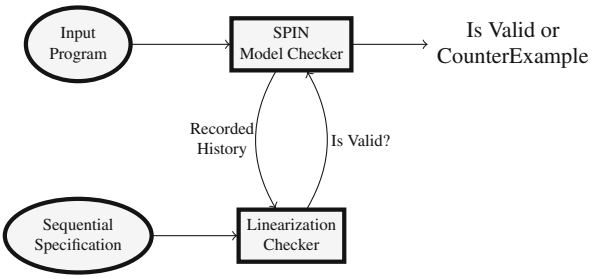

Fig. 3. Checking linearizability using global internal recordings.

Figure 3 depicts the process of checking with global internal recordings (based on the top right section of [24, Fig. 1]). In the input program, data structure models to be tested are instrumented so that client threads non-deterministically invoke operations on the data structure. Invocations and responses of operations are recorded during the state-space exploration. These recordings are then passed to an (external) linearization checker which searches for a valid linearization of the history. It searches by generating a permutation of the history, and then checking whether it satisfies conditions (1) and (2) of Definition 1. Note that condition (1) requires that the linearization checker has its own sequential specification of the data structure, separate from the model checker. If no such linearization can be found, the value returned by the linearization checker causes a local assertion to fail in the model checker.

2.1 Existing Explanations for the Scalability Issues of Automatic Checking

Though well-acknowledged in the literature, explanations for the scalability issues of automatic checking in [24] are not comprehensive. In [24], the authors assert that storing history recordings in the state during model checking limits the state space which can be explored, because "every time we append an element into [*sic*] the history, we introduce a new state".

In contrast, the authors of [12] consider linearization checking, not model checking, to be the performance-limiting factor of automatic checking in [24], stating that:

> "Their approach needs to find a linearizable sequence for each history ... [and] may have to try all possible permutations of the history. As a result, the number of operations they can check is only 2 or 3."

Long and Zhang [13] describe heuristics for improving linearization checking. Their approach also suggests that linearization checking is a performance-limiting factor of automatic linearizability checking. Though their results show the effectiveness of their optimisations, they only test their methods on pre-generated traces; that is, without doing model checking to generate the traces.

As a result, the impact of these optimisations on overall linearizability checking is unclear.

2.2 Testing Explanations for the Scalability Issues of Automatic Checking

To test these different hypotheses, we conducted several preliminary experiments on a concurrent set provided as supplementary material by Vechev et al. [25]. All experiments were performed on a machine running Ubuntu 14.04.3 with 32 GB RAM and a 4-core Intel Core i7-4790 processor. The first compared the performance of automatic checking with and without the linearization checker; see Tables 1 and 2. Without the linearization checker, histories are explored by SPIN but not checked for linearizability. Checking with a linearization point-based approach is also shown for comparison.

In this experiment, two threads invoked operations on the data structure. For 6 operations, both automatic methods were given a moderate state compression setting (the built-in COLLAPSE flag in SPIN – see [11]) but failed to complete due to memory requirements. All times shown are the average of 10 executions. Note that SPIN was used with a single core to avoid time overhead for small tests and memory overhead for large tests.

The results clearly indicate model checking is the performance-limiting factor, since disabling linearization checking does not lead to performance comparable to checking with linearization points.

Table 1. Comparison of execution times for automatic and non-automatic checking methods of Vechev et al. [24]. All times in milliseconds.

Method	History length (# operations)		
	2	4	6
Linearization points	22	257	2160
Global internal recordings	33	10 590	Out of memory (30 GB)
Global internal recordings without linearization checker	33	10 240	Out of memory (30 GB)

Table 2. Comparison of memory use for automatic and non-automatic checking methods of Vechev et al. [24]. All measurements in MB.

Method	History length (# operations)		
	2	4	6
Linearization points	131.0	204.4	773.3
Global internal recordings	136.2	3780.80	Out of memory (30 GB)
Global internal recordings without linearization checker	136.2	3744.2	Out of memory (30 GB)

A second experiment investigated scalability issues in the model checking process. The number of states and histories explored in the same concurrent set were compared; see Tables 3, 4 and 5. For global internal recordings, histories were recorded by modifying the linearization checker. Each time the linearization checker was invoked, the history it was acting on was recorded. When checking with linearization points, the SPIN model was instrumented to output each operation as it was checked. The histories checked were then reconstructed from the output list of recordings.[1]

Note that states 'stored' refers to the number of distinct states in the state space, whereas states 'matched' refers to how many times a state was revisited [11]. Together they give an indication of how much state space exploration occurred.

Table 3. Comparison of states stored by global internal recordings and linearization points methods.

Method	History length (# operations)		
	2	4	6
Linearization points	21 198	1 215 501	12 899 275
Global internal recordings	25 740	12 693 435	Out of memory (30 GB)

Table 4. Comparison of states matched by global internal recordings and linearization points methods.

Method	History length (# operations)		
	2	4	6
Linearization points	4514	329 884	3 765 699
Global internal recordings	4699	2 570 412	Out of memory (30 GB)

Table 5. Comparison of histories checked by global internal recordings and linearization points methods.

Method	History length (# operations)		
	2	4	6
Linearization points	165	2876	9783
Global internal recordings	296	133 536	Out of memory (30 GB)

Tables 3 and 4 confirm the statement of [24] – many more states are explored using automatic checking. However, the magnitude of the difference suggests

[1] Note that reconstruction of histories required adding a global index variable which would not normally be used in checking with linearization points and inflates the state space for reasons explained later in this section. The number of states and number of histories listed for checking with linearization points are therefore overestimates.

more than just one state is introduced by each recording. Table 5 also reveals some implications not immediately evident from previous explanations – that checking with global internal recordings generates and checks many more histories than checking with linearization points. Because this is not encoded manually by the different approaches, it suggests an optimisation by SPIN which allows checking with linearization points to shrink the state space and remove histories which are unnecessary for verifying linearizability.

An interesting trend from the results was the ratio of 'matched' (revisited) to 'stored' (total distinct) states, which was higher for checking done with linearization points. For example, in the case of 2 operations, even though checking with linearization points has 4000 fewer states, it revisits them almost as much as global internal recordings checking. This provides some insight as to why it checks many fewer histories and has vastly better performance.

It was found that the histories checked with linearization points are a strict subset of those checked using global internal recordings. The histories missing from linearization points checking were due to the model checker stopping and backtracking in the middle of a history. That is, SPIN would generate the start of the history but stop before generating some of the recordings for the end of the history. For example, Fig. 4 shows a history that is missed when checking a concurrent set using linearization points. The point 'X' shows where checking for this history stops.

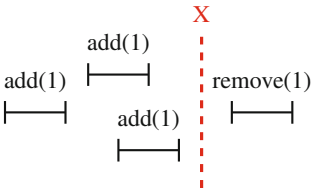

Fig. 4. A missing history when model checking with linearization points.

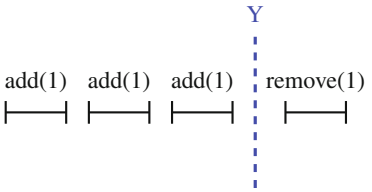

Fig. 5. A history that precedes the missing history.

After examining such histories and considering the algorithm applied by SPIN for model checking it became apparent that the reason SPIN stopped preemptively in some histories was the presence of repeated states. Explicit-state model checking algorithms optimise state space exploration by not returning to a state if all of the possibilities extending from that state have been previously checked (see, for example, [1]).

For example, when checking with global internal recordings, the history in Fig. 4 occurs (in the search process) after the history shown in Fig. 5. When checking with linearization points, at the point X the global state in the history of Fig. 4 matches the global state at point Y in Fig. 5, so the model checker does not proceed any further.

This explains the large number of states and histories generated by global internal recordings. Because of the recordings, states which would otherwise appear identical to SPIN are differentiated. SPIN therefore continues to search down the branch of the state space, whereas with linearization points it would backtrack.

3 A Technique for Improving Scalability of Automatic Checking

We now describe a new automatic checking technique. The key insight is to improve scalability by storing less global data, allowing SPIN to optimise state space exploration by backtracking. The technique is referred to as 'external checking' because it outputs recordings which are stored by the model checker in the automatic checking of [24].

The description provided in this section is for a proof-of-concept implementation using machinery built to work with SPIN. Unfortunately, subtle issues in the state space exploration technique make this implementation an unsound checking procedure for verification. In Sect. 5 we describe the reasons for this unsoundness and present a sound and complete checking procedure that extends the basic idea. Implementing the extension would require altering the SPIN source code and is left for future work.

3.1 External Checking: Preliminary Implementation

The general concept is similar to that of automatic checking with global internal recordings because each history is checked for a linearization. The implementation is also similar, viz., client threads non-deterministically invoke operations on the concurrent data structure to generate the histories. The key difference is that the external checking method outputs information about the operations to an external linearization checker as they occur, rather than keeping an internal list of recordings until the end of each history.

A simplistic approach was taken to outputting recordings externally. An embedded printf statement was included in the Promela model whenever an invocation or response occurred. For example,

```
c_code{printf("%d %d %d %d %d %d\n", now.gix,
    Pclient->par, Pclient->op, Pclient->retval,
    Pclient->arg, Pclient->type);}
```

outputs the index of the recording in the history (`gix`), the parent recording (i.e., invocation) of the operation if it was a response (`par`), the operation (`op`), argument (`arg`), return value (`retval`) and whether this was an invocation or response (`type`) for the thread 'client' (`Pclient`).

External checking requires that output recordings be assembled into complete histories, since the recordings are output in the order in which SPIN explores the state space. Since SPIN uses a depth-first search of the state space, this simply

requires iterating over the list of recordings and outputting a history whenever the last recording (a complete history) is reached.[2] In pseudocode,

```
Recording current_history[history_length];
for (Recording recording : output_recordings) {
    current_history[recording.index] = recording;
    if (recording.index == history_length) {
        //leaf node in the search tree
        outputHistory(current_history);
    }
}
```

A process takes the output from SPIN and reconstructs the histories as shown above. It then passes the histories to the linearization checker which checks each history for a linearization. The entire external checking procedure is illustrated in Fig. 6. Compare this to Fig. 3 for checking with global recordings. Note that the external linearization checker runs concurrently with the model checker. If a failure (non-linearizable history) occurs, it notifies the model checker and both stop.

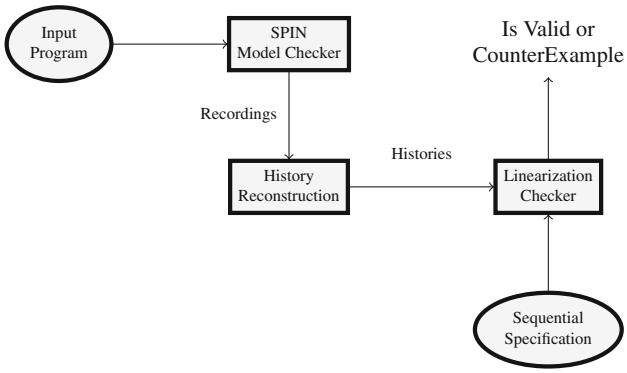

Fig. 6. The external checking procedure.

Note that at present external checking is only suitable for use with single-core SPIN checking. Using several cores changes how the state space is explored and therefore how recordings are output, so it requires understanding a different state space exploration algorithm and also the capacity to determine from which core the recordings originated. Further work could explore implementing these features.

[2] Note that histories are limited to a given length to make model checking feasible.

4 Results

Three popular data structures from the literature with known defects were used for testing the effectiveness of the external checking method. These data structures are summarised in Table 6. It is important to note that both the buggy queue and the Snark deque were originally published with proofs of correctness, and only later found to be defective. They therefore represent realistic examples of bugs in concurrent data structures. The ABA problem, tested for in both the Treiber Stack and Snark deque, is also a common problem with concurrent data structures which use the compare-and-swap primitive.

Table 6. Faulty data structures used for testing external checking.

Data structure	Source	Description of bug
Treiber stack	[21]	Suffers from the ABA problem in non-memory managed environments. Excellent explanation in [26, Sect. 1.2.4]
Buggy queue	[18]	When a dequeue is interrupted by two enqueues at critical sections, the dequeue returns a value not from front of the queue. See [6, Sect. 3.3]
Snark deque	[7]	Two bugs, the first of which can cause either popLeft or popRight to return empty when the queue is nonempty, and the second of which is an ABA-type error resulting in the return of an already popped value. See [8, Sect. 3] for detailed descriptions

Promela models of the data structures in Table 6 were created and instrumented to allow automatic checking both externally and via global internal recordings. In cases where more than one bug existed in a single data structure, each bug was repaired after being flagged so that others could be tested. Experiments were performed on a machine running Ubuntu 14.04.3 with 32 GB RAM and a 4-core Intel Core i7-4790 processor, with the exception of the final Snark deque bug. Its tests were executed on a machine running Oracle Linux 6 with two 22-core Intel Xeon CPU E5-2699 v4 processors and 378 GB RAM due to high memory requirements. SPIN was used with a single core to avoid time overhead for small tests and memory overhead for large tests. Also, external checking does not currently support checking with multiple cores.

External checking located all bugs. Global checking found all except the final Snark deque bug - after 2.87×10^7 ms ($\sim$ 8 h) the memory limit of 300000 MB was reached and SPIN exited without locating the bug. The results of testing for detected bugs are shown in Table 7. For the first 3 bugs, no state compression flags were needed, and only 2 threads and 4 operations were required for detection. Times shown are an average of 10 executions for both methods. For

Table 7. Bugs detected by external checking and global recordings checking.

Data structure	Bug number	External checking		Global recordings checking	
		Time (ms)	Memory (MB)	Time (ms)	Memory (MB)
Treiber stack	1/1	373	172	1346	342
Buggy queue	1/1	248	159	774	252
Snark deque	1/2	86	139	123	145
Snark deque	2/2	2.71×10^7	248227	–	–

the final snark deque bug, the COLLAPSE memory compression flag was used (see [11] for details), as the failure trace required 3 threads and 7 operations. Trials for this bug were run once due to resource constraints.

4.1 Discussion of External Checking Performance

The results in Table 7 illustrate the utility of the external checking method. It was able to locate all bugs, even without the improvements described in Sect. 5. This suggests it is uncommon in practice that a bug cannot be detected by the method.

In addition, external checking was both faster and used less memory than global checking in all cases. For the Treiber stack and buggy queue, memory use was roughly half that of global recordings checking, and checking was around three times faster.

In the case of the second Snark deque bug, there was sufficient memory for external checking to find the bug, but not enough for global recordings checking. Of course, global recordings checking would detect the bug if sufficient memory or time were available, since it is a verification technique. However, the results show it requires *at least* 50 GB more memory than external checking (or the equivalent amount of time with a stronger compression), which illustrates the benefit of a faster bug-finding technique for bugs with long failure traces.

For comparison, tests with linearization point-based checking show that this bug can be located in under 30 min with COLLAPSE state compression, illustrating that automatic methods are not as scalable as linearization points-based methods.

The two automatic methods are closest in performance for the first bug of the Snark deque. This is because the failing history occurs very early in the model checking process. External checking takes longer to check any individual history because it must be reconstructed and then passed to the linearization checker. Its performance benefit comes from checking far fewer histories. Therefore when a bug occurs after only very few histories, external checking does not have time to yield a significant performance benefit. Conversely, the deeper the execution required to locate a bug, the greater the improvement in performance compared to global internal recordings.

5 Potential Improvements: Integration with SPIN

The technique described in Sect. 3.1 is in fact unsound. Recall from Sect. 2.2 that checking with linearization points covers fewer histories due to SPIN optimisations that cause it to stop at repeated states. This is valid with linearization point-based checking because such approaches include an abstract specification that runs in parallel with the model of the concrete implementation. The state variables of the abstract specification ensure that the sub-history encountered before backtracking is truly equivalent to one checked earlier.

However, in external checking no abstract specification is kept by SPIN. This means there are cases where SPIN stops preemptively and this prevents it checking a history that could violate linearizability.

For example, consider the sequential specification of a data structure as shown in Fig. 7. Suppose this specification was incorrectly implemented as shown in Fig. 8. If checking on a single thread is used, the SPIN output (shown diagrammatically) is as in Fig. 9.

```
int x = 0;
atomic operation 1:
    x++;
    return x;
atomic operation 2:
    return True;
```

Fig. 7. Abstract specification.

```
int x = 0;
operation 1:
    x++;
    return x;
operation 2:
    if (x == 0)
        x = 1;
    return True;
```

Fig. 8. Incorrect implementation.

Checking stops before the end of the third (faulty) history, and therefore it is not checked and no error is raised. The model checker stops because of the repeated global state `x = 1`. It reaches this state after `operation1` in the first two histories and from those histories has explored all states extending from that

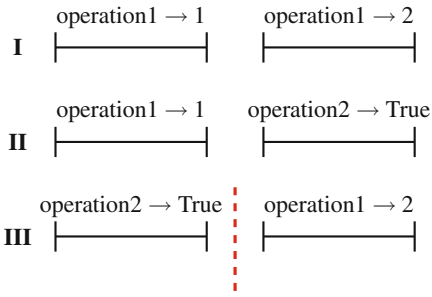

Fig. 9. Histories output by SPIN when using external checking on the data structure of Fig. 8. The dashed line indicates SPIN stopping.

point. When SPIN encounters the same state after `operation2` completes in the third history, it stops, despite the global state being incorrect for an execution of `operation2`.

Note that checking with linearization points, where an abstract specification is included, would prevent this error, since the abstract specification's `operation1` and `operation2` will alter the global data differently.

5.1 A Sound Verification Algorithm

We now describe a means of extending our technique for verification, which requires modifying the SPIN source. Doing this would also lead to a significant performance benefit.

Outputting recordings requires keeping track of a global index. As Sect. 2.2 showed, global variables tracked by SPIN can unnecessarily inflate the state space. If SPIN were modified it would not be necessary to keep a global index as a global variable in the model – it could be kept as metadata instead.

Likewise, the machinery of Sect. 3.1 could be implemented in a very similar fashion in SPIN. Instead of outputting recordings, it could be stored as metadata separate from the state vector and model checking process. Complete histories would still have to be passed to an external linearization checker, as was done in [24].

We now outline the extra checking necessary to prevent the missing histories described in the previous section, making the approach sound. It was noted that repeated global states cause the lack of soundness. This problem does not occur when checking with linearization points because the abstract specification is present in the global state and represents the expected behaviour of the implementation. Therefore a repeated state always indicates identical behaviour.

Incorrect backtracking in external checking could therefore be prevented by using the abstract specification to decide when a repeated global state represents correct behaviour of the implementation. We propose the following method: whenever a repeated global state is reached, ensure that the current sub-history has a linearization which leads to the same state in the abstract specification as the sub-history which originally created that global state. This would require keeping track of the valid linearizations for previously encountered histories.

For example, suppose during checking histories for a stack implementation, the model checker had explored all states extending from the global state G, as shown in Fig. 10. This implies every full history reached from G with the sub-history shown in Fig. 10 had a linearization. In verifying this, the linearization checker would have found that the operations before G have the valid linearization [push(1) → `True`, pop() → 1]. Therefore the abstract state at G was an empty stack in all of the checked histories represented by Fig. 10.

Suppose the sub-history shown in Fig. 11 then occurred, repeating the global state G. To determine whether backtracking is correct, it suffices to check that the sub-history up to G has a linearization which would lead to an empty stack in the abstract specification. In this case it is possible by the linearization [push(1) → `True`, pop() → 1, pop() → `empty`]. This means SPIN can backtrack safely.

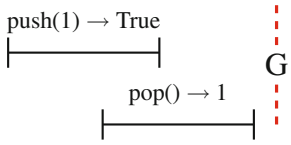

Fig. 10. Example history.

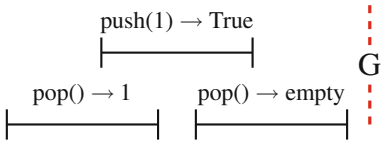

Fig. 11. Second example history.

In contrast, recall the counter-example to verification from Fig. 9. In this example, the histories verified by the model checker (histories `I` and `II`) have linearizations [`operation1` → 1, `operation1` → 2] and [`operation1` → 1, `operation2` → `True`], respectively. That is, in both cases the linearization up to the repeated state is `operation1` → 1, meaning the abstract specification state at that point is `x = 1`. When the same global state is reached in history `III`, there is no linearization of `operation2` → `True` which leads to the abstract state `x = 1`. Only `x = 0` is possible. Therefore in the proposed implementation SPIN cannot backtrack after `operation2` → `True` and the entire history would be checked and found invalid.

Note that this extended approach requires checking for linearizations, metadata caching and the usual state exploration of model checking. Performance could be improved by a high degree of parallelism between these separate functions.

6 Conclusions

We have described in detail the scalability issues of automatic linearizability checking in [24]. The main cause is a lack of state space traversal optimisations due to a large amount of global data in the model checking state. This identified cause makes explicit a fact which is widely assumed in the literature but whose explanation is often omitted or unclear.

These observations motivated a new, more scalable technique for automatic checking with SPIN. The key insight is to *not store the recordings* in the model checker for checking at the end of each history, but instead to output them immediately. This allows the model checker to optimise the state space exploration. The algorithm we have implemented reconstructs the histories from the recordings and determines if these histories satisfy the linearization conditions. Our experiments show that the extra cost of generating the history from the recordings that are output directly is smaller than the speed-up gained from the more efficient execution of the model checker.

This external checking technique reduces the number of histories that need exploration and thus is able to explore longer traces. As a consequence bugs that occur on long traces are detected more efficiently than when using the global internal recordings technique in the literature. External checking does detect bugs that occur after a few histories but the performance benefits are

not significant. In other words, the more states the model checker is required to explore before it can detect a bug, the more effective our technique will be.

We have also presented a limitation of the implemented external checking technique (namely, that it can be used for bug detection but not verification). We have developed an algorithm that overcomes this limitation, and intend to implement this in SPIN as future work. Note that if only an efficient bug detection technique is desired, the external checking algorithm described in Sect. 3 would suffice.

Acknowledgments. The authors would like to thank Martin Vechev for providing extra materials that allowed evaluation of the automatic checking in [24]. This work is partially supported by ARC Discovery Grant DP160102457.

References

1. Baier, C., Katoen, J.-P.: Principles of Model Checking. The MIT Press, Cambridge (2008)
2. Bouajjani, A., Emmi, M., Enea, C., Hamza, J.: Verifying concurrent programs against sequential specifications. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 290–309. Springer, Heidelberg (2013). doi:10.1007/978-3-642-37036-6_17
3. Bouajjani, A., Emmi, M., Enea, C., Hamza, J.: On reducing linearizability to state reachability. In: Halldórsson, M.M., Iwama, K., Kobayashi, N., Speckmann, B. (eds.) ICALP 2015. LNCS, vol. 9135, pp. 95–107. Springer, Heidelberg (2015). doi:10.1007/978-3-662-47666-6_8
4. Burckhardt, S., Dern, C., Musuvathi, M., Tan, R.: Line-up: a complete and automatic linearizability checker. In: PLDI 2010, Proceedings of 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 330–340. ACM, New York (2010)
5. Černý, P., Radhakrishna, A., Zufferey, D., Chaudhuri, S., Alur, R.: Model checking of linearizability of concurrent list implementations. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 465–479. Springer, Heidelberg (2010). doi:10.1007/978-3-642-14295-6_41
6. Colvin, R., Groves, L.: Formal verification of an array-based nonblocking queue. In: ICECCS 2005, pp. 507–516. IEEE, Los Alamitos (2005)
7. Detlefs, D.L., Flood, C.H., Garthwaite, A.T., Martin, P.A., Shavit, N.N., Steele, G.L.: Even better DCAS-based concurrent deques. In: Herlihy, M. (ed.) DISC 2000. LNCS, vol. 1914, pp. 59–73. Springer, Heidelberg (2000). doi:10.1007/3-540-40026-5_4
8. Doherty, S., Detlefs, D.L., Groves, L., Flood, C.H., Luchangco, V., Martin, P.A., Moir, M., Shavit, N., Steele Jr., G.L.: DCAS is not a silver bullet for nonblocking algorithm design. In: Gibbons, P.B., Adler, M. (eds.) SPAA 2004, pp. 216–224. ACM, New York (2004)
9. Filipovic, I., O'Hearn, P.W., Rinetzky, N., Yang, H.: Abstraction for concurrent objects. Theor. Comput. Sci. **411**(51–52), 4379–4398 (2010)
10. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990)
11. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley, Reading (2003)

12. Liu, Y., Chen, W., Liu, Y.A., Sun, J.: Model checking linearizability via refinement. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 321–337. Springer, Heidelberg (2009). doi:10.1007/978-3-642-05089-3_21
13. Long, Z., Zhang, Y.: Checking linearizability with fine-grained traces. In: SAC 2016, pp. 1394–1400. ACM, New York (2016)
14. Moir, M., Shavit, N.: Concurrent data structures. In: Mehta, D.P., Sahni, S. (eds.) Handbook of Data Structures and Applications, Chap. 47, pp. 1–30. Chapman and Hall, CRC Press (2004)
15. Research in Software Engineering Group (RiSE). http://chesstool.codeplex.com/
16. Schellhorn, G., Derrick, J., Wehrheim, H.: A sound and complete proof technique for linearizability of concurrent data structures. ACM Trans. Comput. Log. **15**(4), 31:1–31:37 (2014)
17. School of Computing, National University of Singapore. http://pat.comp.nus.edu.sg/
18. Shann, C.H., Huang, T.L., Chen, C.: A practical nonblocking queue algorithm using compare-and-swap. In: ICPADS 2000, pp. 470–475. IEEE, Los Alamitos (2000)
19. Smith, G.: Model checking simulation rules for linearizability. In: De Nicola, R., Kühn, E. (eds.) SEFM 2016. LNCS, vol. 9763, pp. 188–203. Springer, Cham (2016). doi:10.1007/978-3-319-41591-8_13
20. Sun, J., Liu, Y., Dong, J.S., Pang, J.: PAT: towards flexible verification under fairness. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 709–714. Springer, Heidelberg (2009). doi:10.1007/978-3-642-02658-4_59
21. Treiber, R.K.: Systems Programming: Coping with Parallelism. International Business Machines Incorporated, Thomas J. Watson Research Center, New York (1986)
22. Vafeiadis, V., Parkinson, M.: A marriage of rely/guarantee and separation logic. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 256–271. Springer, Heidelberg (2007). doi:10.1007/978-3-540-74407-8_18
23. Vafeiadis, V.: Shape-value abstraction for verifying linearizability. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 335–348. Springer, Heidelberg (2008). doi:10.1007/978-3-540-93900-9_27
24. Vechev, M., Yahav, E., Yorsh, G.: Experience with model checking linearizability. In: Păsăreanu, C.S. (ed.) SPIN 2009. LNCS, vol. 5578, pp. 261–278. Springer, Heidelberg (2009). doi:10.1007/978-3-642-02652-2_21
25. Vechev, M., Yahav, E., Yorsh, G.: Paraglide: SPIN Models. http://researcher.watson.ibm.com/researcher/view_group_subpage.php?id=1290
26. Wolff, S.: Thread-modular reasoning for heap-manipulating programs: exploiting pointer race freedom. Master's thesis, University of Kaiserslautern (2015)
27. Zhang, L., Chattopadhyay, A., Wang, C.: Round-up: runtime checking quasi linearizability of concurrent data structures. In: Denney, E., Bultan, T., Zeller, A. (eds.) ASE 2013, pp. 4–14. IEEE, Los Alamitos (2013)
28. Zhang, S.J.: Scalable automatic linearizability checking. In: ICSE 2011, Proceedings of 33rd International Conference on Software Engineering, pp. 1185–1187. ACM, New York (2011)