Improving the Scalability of Automatic Linearizability Checking in SPIN

Patrick Doolan[1,2], Graeme Smith[2], Chenyi Zhang[1] and Padmanabhan Krishnan[1]

[1]Oracle Labs, Brisbane, Australia
[2]School of Information Technology and Electrical Engineering,
The University of Queensland, Australia

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 [VYY09] 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 [MS07]. This motivates programmers to create objects with fewer safety mechanisms (such as locks) to achieve greater concurrency. However, as noted by [MS07], 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., [SHC00] and [DFG$^+$00]). 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 [HW90], is the appropriate notion of correctness. The definition of linearizability given by Vechev et al. [VYY09] 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*

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() $\rightarrow 1$, enqueue(2)]. Conversely, consider Figure 2, which is also a history

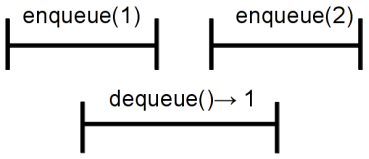

Fig. 1. Concurrent history with a linearization.

of a concurrent queue. This does not have a linearization, because, by the partial order conditions, 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 [SHC00].

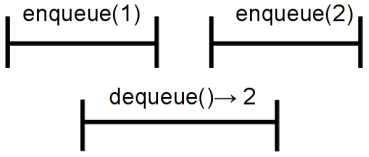

Fig. 2. Concurrent history with no linearization.

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*) [MS07]. Furthermore, [FORY10] 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.

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 [DGLM04] and [VP07] respectively for examples with and without a theorem prover), to static and runtime analysis (e.g., [Vaf09] and [ZCW13], respectively) and model checking (e.g., [VYY09], [LCLS09] and [BDMT10]).

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, whereas *automatic checking* does not. The latter has two advantages, viz., it does not have errors arising from incorrectly specified linearization points and also has greater flexibility for data structures with non-fixed 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. [VYY09] 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 [Hol04]. 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. [LCLS09] use the model checker PAT [Sch14] for automatic checking of linearizability. However, to operate on larger state spaces, a linearization point-based approach is required. This situation is improved on by Zhang [Zha11] by using symmetry to narrow the potential state space, and in doing so they are able to verify concurrent data structures (albeit simple ones) for three to six threads. In contrast, automatic checking reported by Vechev et al. [VYY09] only allows two threads.

Burckhardt et al. [BDMT10] describe the tool Line-Up, built on top of the model checker CHESS [Res10], 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 [Zha11], 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.

1.2 Contributions

A notable theme in the related work is that automatic methods are considered to have inherent scalability issues for verification [VYY09] [LCLS09], though they can be used effectively when limits are placed on types or numbers of operations checked [VYY09]

[BDMT10] or advanced state compression techniques are used [Zha11]. 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. [VYY09] 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 Section 2 we present our analysis of the scalability issues in the work of Vechev et al. [VYY09]. A technique for overcoming these issues is presented in Section 3, and the results of applying an implementation of this technique to data structures from the literature with known bugs is described in Section 4. Also in Section 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 [VYY09], 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.

Figure 3 depicts the process of checking with global internal recordings (based on the top right section of [VYY09, Figure 1]). Data structure models to be tested are

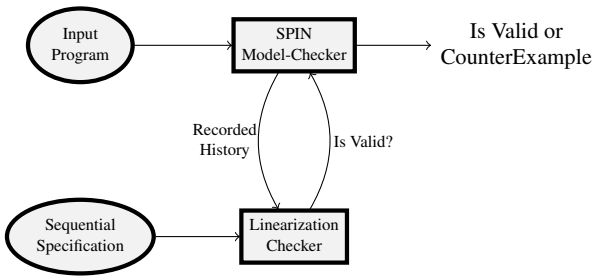

Fig. 3. Checking linearizability using global internal recordings.

instrumented so that client threads non-deterministically invoke operations on the data structure. Invocations and responses of operations are recorded. These recordings are then passed to a 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 of the Scalability Issues of Automatic Checking

Though well-acknowledged in the literature, explanations for the scalability issues of automatic checking in [VYY09] are not comprehensive. In [VYY09], the authors note that "every time we append an element into [*sic*] the history, we introduce a new state", explaining that the recordings create scalability issues related to state space explosion.

However, [LCLS09] consider linearization checking to be the performance-limiting factor of automatic checking in [VYY09], stating that "Their approach needs to find a linearizable sequence for each history, **whose worst-case time is exponential in the length of the history, as it 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." (emphasis added).

Long and Zhang [LZ16] describe heuristics for improving linearization checking. Their approach 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 of 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. [VYY16]. All experiments were performed on a machine running Ubuntu 14.04.3 with 32 GB RAM and 8 Intel Core i7-4790 processors. The first compared the performance of automatic checking with and without the linearization checker; see Table 1 where checking with a linearization point-based approach is also shown for comparison. Two threads queried the data structure. For 6 operations, both automatic methods were given a moderate state compression (using DCOLLAPSE – see [Hol04]) but failed to complete due to memory requirements. In those cases, times shown are the amount of time until the memory limit was reached. 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.

Table 1. Comparison of automatic and non-automatic checking methods of Vechev et al.

Number of	Execution time (ms)			Memory usage (MB)		
operations	lin pts	no lin checker	automatic	lin pts	no lin checker	automatic
2	22	33	33	131.0	136.2	136.2
4	257	10240	10590	204.4	3744.2	3780.8
6	2160	457000	467000	773.3	Out of memory (30GB)	

The results clearly indicate the model checking is the performance-limiting factor, as using automatic methods while disabling linearization checking offers little performance benefit, particularly compared to checking with linearization points.

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 Table 2. 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 [Hol04]. Together they give an indication of how much state space exploration occurred.

Table 2. Comparison of state and history exploration between checking with global internal recordings and with linearization points. †Memory use exceeded 30GB with DCOLLAPSE.

Factor	Method	Number of operations		
		2	4	6
Number of states stored	linearization points	21198	1215501	12899275
	global internal recordings	25740	12693435	–†
Number of states matched	linearization points	4514	329884	3765699
	global internal recordings	4699	2570412	–†
Number of histories checked	linearization points	165	2876	9783
	global internal recordings	296	133536	–†

Table 2 confirms the statement of [VYY09] – many more states are explored using automatic checking. However, the magnitude of the difference suggests more than just one state is introduced by each recording. The results also reveal 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 that the biggest difference in states explored by the two methods was in states 'stored' – that is, unique states in the state space. However, states 'matched' (revisited) were much closer (note in the case of 2 operations that though checking with linearization points has 4000 fewer states, it revisits states 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 checking process stopping and backtracking in the middle of a history. That is, SPIN would generate the start of the history but stop

[1] Note that reconstruction of histories required a global index variable, which inflates the state space for reasons explained in Section 2.2. The number of histories listed for checking with linearization points is therefore an over-estimate.

before generating some of the recordings for the end of the history. For example, Figure 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.

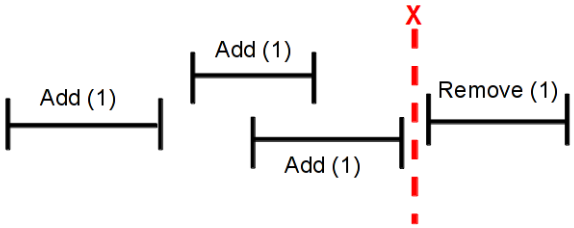

Fig. 4. A missing history when model checking with linearization points.

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. State-based 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, [BK08]).

For example, when checking with global internal recordings, the history in Figure 4 occurs (in the search process) after the history shown in Figure 5. When checking with

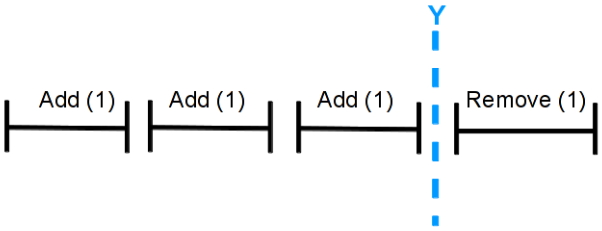

Fig. 5. A history that precedes the missing history.

linearization points, at the point X the global state in the history of Figure 4 matches the global state at point Y in Figure 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 explo-

ration 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 [VYY09].

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 Section 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 alteration of 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 where each history is checked for a linearization. The implementation is also similar, viz., client threads non-deterministically query 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);
    }
}
```

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

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 Figure 6. Compare this to Figure 3 for checking with global recordings.

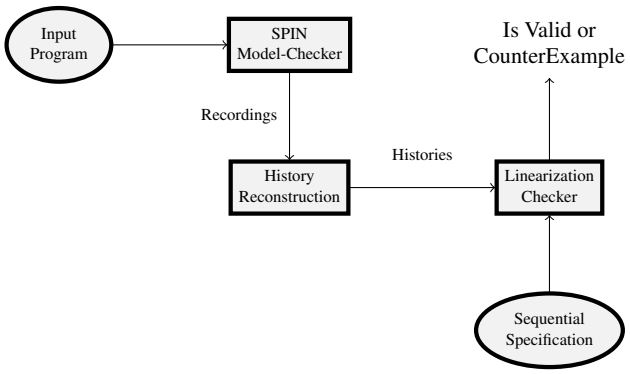

Fig. 6. The external checking procedure.

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.

4 Results

Several 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 3. 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.

Promela models of the data structures in Table 3 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. All experiments were performed on a machine running Ubuntu 14.04.3 with 32 GB RAM and 8 Intel Core i7-4790 processors. 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.

Table 3. Faulty data structures used for testing external checking.

Data structure	Source	Description of bug
Treiber stack	[Tre86]	Suffers from the ABA problem in non-memory managed environments. Excellent explanation in [Wol15, Section 1.2.4].
Buggy queue	[SHC00]	When a dequeue is interrupted by two enqueues at critical sections, the dequeue returns a value not from front of the queue. See [CG05, Section 3.3].
Snark deque	[DFG+00]	Two bugs, the first of which can cause either pop to return 'empty' when the queue contains elements, and the second of which is an ABA-type error resulting in the return of an already popped value. See [DDG+04, Section 3] for detailed descriptions.

Three out of four bugs tested were located. The results of testing for detected bugs is shown in Table 4. In the cases where bugs were located, 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. Only the second bug for the Snark deque was unable to be detected after running out of memory in 50 hours with the strong state compression flag DMA = 496 (see [Hol04] for details) and a memory limit of 21GB.

Table 4. Results of external checking and global recordings checking on faulty data structures.

Data structure	Bugs Detected	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

4.1 Discussion of External Checking Performance

The results in Table 4 illustrate the utility of the external checking method. Finding three of the fours bugs, even without the improvements described in Section 5, indicates that this method alone has promise as a bug-finding approach.

In the case of the missed bug, the second in the Snark deque, it seems likely that the failing history was skipped by external checking. Tests with linearization point-based checking show that this bug can be located in under 30 minutes with DCOLLAPSE state compression. Hence it is possible that the failing history was missed early on in external checking, which then proceeded to check the remainder of the (very large) state space, which is time and memory consuming.

However, when bugs are detected, external checking appears more scalable than checking with global internal recordings based on the results of Table 4. For all bugs

detected it was both faster and used less memory. For the Treiber stack and buggy queue, memory use was roughly half that of global internal recordings, and checking was around three times faster.

For the first bug of the Snark deque the two methods are closest in performance. 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 Section 3.1 is in fact unsound. Recall from Section 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 Figure 7. Suppose this specification was incorrectly implemented as shown in Figure 8. If checking on a single thread is used, the SPIN output (shown diagrammatically) is as in Figure 9.

```
int x = 0;
atomic operation1:
    x++;
    return x;
atomic operation 2:
    return True;
```

Fig. 7. Abstract specification.

```
int x = 0;
operation1:
    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 point. When SPIN encounters

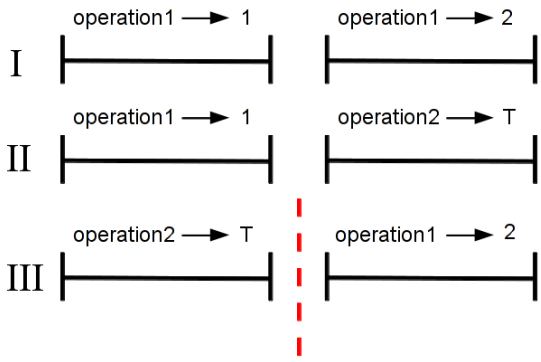

Fig. 9. Histories output by SPIN when using external checking on the data structure of Figure 8. The dotted line indicates SPIN stopping.

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 Section 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 Section 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 [VYY09].

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. For external checking, preventing incorrect backtracking therefore requires a method for deciding 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 in common with the history which originally created that global state.

For example, suppose the history shown in Figure 10 occurred during checking, followed by the history shown in Figure 11. Here G is a global state, and `op1` and `op2`

are operations. At the global state G in the history of Figure 11, it can be checked that

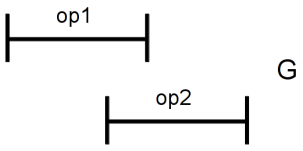

Fig. 10. Example history.

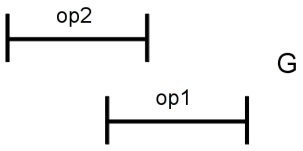

Fig. 11. Second example history.

the two histories share two potential linearizations: `op1` then `op2`, and `op2` then `op1`. This means SPIN can backtrack safely.

In contrast, recall the counter-example to verification from Figure 9. In this example there is no linearization of the history containing just `op2` which is also a linearization of the history containing just `op1`. Therefore in the proposed implementation SPIN would not stop after `op2` and the entire history would be checked, not missed.

Equivalently, SPIN could check that the sub-history up to the stopping point had a linearization which could lead to that global state. This would require having an abstract specification of the data structure, and being able to check that the abstract state matched the concrete one. This is, however, more dependent on the data structure being checked and therefore is not favoured as an automatic approach to checking.

6 Conclusions

We have described in detail the scalability issues of automatic linearizability checking in [VYY09]. The main cause of this is a lack of state space traversal optimisations in the presence of a large amount of global data. 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* till the end of history in the model-checker but to output them directly. 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 recording 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, but it is currently not implemented. Future work should focus on means of implementing the verification technique in SPIN, as described in Section 5. Note that if only an efficient bug detection technique is desired, the external checking algorithm described in Section 3 would suffice but if verification is desired the extended algorithm can be used.

Acknowledgments. The authors would like to thank Martin Vechev for providing extra materials that allowed evaluation of the automatic checking in [VYY09].

References

[BDMT10] Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. Line-Up: a complete and automatic linearizability checker. In *ACM SIGPLAN Notices*, volume 45, pages 330–340. ACM, 2010.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press Cambridge, 2008.

[CG05] Robert Colvin and Lindsay Groves. Formal verification of an array-based nonblocking queue. In *10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)*, pages 507–516. IEEE, 2005.

[DDG+04] Simon Doherty, David L Detlefs, Lindsay Groves, Christine H Flood, Victor Luchangco, Paul A Martin, Mark Moir, Nir Shavit, and Guy L Steele Jr. DCAS is not a silver bullet for nonblocking algorithm design. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 216–224. ACM, 2004.

[DFG+00] David L Detlefs, Christine H Flood, Alexander T Garthwaite, Paul A Martin, Nir N Shavit, and Guy L Steele Jr. Even better DCAS-based concurrent deques. In *Distributed Computing*, pages 59–73. Springer, 2000.

[DGLM04] Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. Formal verification of a practical lock-free queue algorithm. In *Formal Techniques for Networked and Distributed Systems–FORTE 2004*, pages 97–114. Springer, 2004.

[FORY10] Ivana Filipović, Peter O'Hearn, Noam Rinetzky, and Hongseok Yang. Abstraction for concurrent objects. *Theoretical Computer Science*, 411(51):4379–4398, 2010.

[Hol04] Gerard J Holzmann. *The SPIN model checker: Primer and reference manual*, volume 1003. Addison-Wesley Reading, 2004.

[HW90] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[LCLS09] Yang Liu, Wei Chen, Yanhong A Liu, and Jun Sun. Model checking linearizability via refinement. In *FM 2009: Formal Methods*, pages 321–337. Springer, 2009.

[LZ16] Zhenyue Long and Yu Zhang. Checking linearizability with fine-grained traces. In *ACM Symposium on Applied Computing*, pages 1394–1400. ACM, 2016.

[MS07] Mark Moir and Nir Shavit. Concurrent data structures. *Handbook of Data Structures and Applications*, pages 47–14, 2007.

[Res10] Research in Software Engineering Group (RiSE). CHESS: Systematic concurrency testing, 2010. Available at `http://chesstool.codeplex.com/license`; accessed 16-Jan-2016.

[Sch14] School of Computing, National University of Singapore. PAT: Process analysis toolkit, 2014. Available at `http://pat.sce.ntu.edu.sg/`; accessed 18-Jan-2016.

[SHC00] Chien-Hua Shann, Ting-Lu Huang, and Cheng Chen. A practical nonblocking queue algorithm using compare-and-swap. In *Proceedings of the Seventh International Conference on Parallel and Distributed Systems*, pages 470–475. IEEE, 2000.

[Tre86] R Kent Treiber. *Systems programming: Coping with parallelism*. International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.

[Vaf09] Viktor Vafeiadis. Shape-value abstraction for verifying linearizability. In *Verification, Model Checking, and Abstract Interpretation*, pages 335–348. Springer, 2009.

[VP07] Viktor Vafeiadis and Matthew Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR 2007–Concurrency Theory*, pages 256–271. Springer, 2007.

[VYY09] Martin Vechev, Eran Yahav, and Greta Yorsh. Experience with model checking linearizability. In *Model Checking Software*, pages 261–278. Springer, 2009.

[VYY16] Martin Vechev, Eran Yahav, and Greta Yorsh. Paraglide: Spin models, 2016. Available at `http://researcher.watson.ibm.com/researcher/view_group_subpage.php?id=1290`; accessed 11-Jan-2016.

[Wol15] Sebastian Wolff. *Thread-modular reasoning for heap-manipulating programs: exploiting pointer race freedom*. PhD thesis, University of Kaiserslautern, 2015.

[ZCW13] Lu Zhang, Abhiroop Chattopadhyay, and Chao Wang. Round-Up: Runtime checking quasi linearizability of concurrent data structures. In *2013 IEEE/ACM 28th International Conference on Automated Software Engineering (ASE)*, pages 4–14. IEEE, 2013.

[Zha11] Shao Jie Zhang. Scalable automatic linearizability checking. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 1185–1187. ACM, 2011.