Context-Switch-Directed Verification in DIVINE

Vladimír Štill, Petr Ročkai*, and Jiří Barnat

Faculty of Informatics, Masaryk University
Brno, Czech Republic
{xstill,xrockai,barnat}@fi.muni.cz

Abstract. In model checking of real-life C and C++ programs, both search efficiency and counterexample readability are very important. In this paper, we suggest context-switch-directed exploration as a way to find a well-readable counterexample faster. Furthermore, we allow to limit the number of context switches used in state-space exploration if desired. The new algorithm is implemented in the DIVINE model checker and enables both unbounded and bounded context-switch-directed exploration for models given in LLVM bitcode, which efficiently allows for verification of multi-threaded C and C++ programs.

1 Introduction

Finding concurrency-related errors, such as deadlocks, livelocks and data races and their various consequences, is extremely hard – the standard testing approach does not allow the user to control the precise timing of interleaved operations. As a result, some concurrency bugs that occur under a specific interleaving of threads may remain undetected even after a substantial period of testing. To remedy this weakness of testing, formal verification methods, explicit-state model checking in particular, can be very helpful.

The model checking procedure [7] systematically explores all configurations (states) of a program under analysis. For programs without I/O – such as automated test-cases – the entire state space is derived from the program source code itself. For programs *with* IO, there are multiple options – either consider states reachable for a given input data vector (this method is usually employed with multi-threaded programs and explicit-state model checking), or attempt to analyse the program under all possible inputs (a method preferred by symbolic tools and static analysers). In either case, the procedure can easily reveal states of the program that are reachable under any thread interleaving, even though these states may be very hard to reach with testing alone. Examples of explicit-state model checkers include SPIN [9], DIVINE [3], or LTSmin [12].

Unfortunately, the application of model checkers in general software development practice is burdened by the state-space explosion problem – the size of the state space is exponential: the number of threads is the exponent and their length (the number of discrete steps they make) is the base. In a state space

* The contribution of Petr Ročkai has been partially supported by Red Hat, Inc.

constructed directly from fine-grained, assembly-level programs, this problem becomes prohibitive very quickly. Verification of LLVM bitcode is a prime example where naive scheduling leads to extremely large state spaces even for small programs.

When an explicit-state model checker decides that the particular property it is verifying does not hold, it does so because it has encountered a counterexample disproving the property. This counterexample is usually part of the output of the model checking tool, and is often very valuable. Besides being a witness of the erroneous behaviour of the program, it provides a guide to the user, describing in *what particular way* the property is violated. For safety properties, the counterexample consists of a linear sequence of program states that lead to a safety violation. For liveness properties, the sequence is not linear but forms a "lasso" shape, with a linear prefix and a cycle, representing an infinite program run.

In the role of a witness – where the counterexample is simply simulated to show that the error behaviour indeed happens as described – any counterexample is as good as any other. However, this is not true when we consider its role as a guide for the developer. Some counterexamples are easier to understand and make it easier to fix the problem in the system, while others are hard to understand. In a multi-threaded program, one of the criteria for the desired simplicity in counterexample traces is the number of context switches (that is, the number of times the control moves from one thread or process to another). Oftentimes, a counterexample with more discrete steps but with fewer context-switches is superior to a shorter, but more "interleaved" one.

In order to discover simpler and more useful counterexamples, then, we suggest and evaluate a context-switch-driven strategy of exploration of states. This idea takes advantage of the fact that most bugs require only a few context switches to be demonstrated. By prioritising search direction toward traces with fewer context switches, we make our model checker produce more readable counterexamples.

While the improvement in counterexample readability may be a subjective criterion, there are other – more objective – benefits that can be derived from such a context-switch-directed search. First, we can expect the time it takes to discover a counterexample to decrease. This is in line with the assumption that most errors only need a small number of context switches to exhibit [1]. Thus, orchestrating the search to first completely explore the areas of the state space that are reachable under few context switches actually increases the chance of discovering an error.

Second, we can expect the performance of our model checker to be more consistent. Since DIVINE runs a parallel search, the exact exploration order is not preserved across multiple runs. If a state space contains multiple error states (which is common), any of those could be discovered first and terminate the verification. Actually, due to the different exploration order the running times of model checker as well as counterexamples obtained differ significantly even if the same verification task is executed repeatedly [11]. This non-determinism makes

comparisons difficult for the user (especially in cases where there are multiple versions of the model with the same bug) and is therefore undesirable. With the introduction of context-switch directed search, we expect the consistency of both the running time and the counterexamples discovered to be substantially better than with the conventional approach.

The contribution of this work lies primarily in the implementation and evaluation of context-switch-driven model checking within the context of the well-established, explicit-state model checker DIVINE. Since DIVINE can be readily applied in practice, this extension imparts direct practical benefits to its users. Even though the arguably most useful effect (counterexample readability) is not easily quantifiable, we show improvements in a number of objective categories.

The rest of the paper proceeds as follows. In Section 2 we briefly describe relevant related work. Section 3 presents the algorithm for context-switch-driven verification as implemented in DIVINE and puts the new extension in the context of other aspects related to explicit-state model checking with our model checker. Section 4 presents experimental evaluation of our new DIVINE extension, and finally, Section 5 concludes the paper.

2 Related Work

Summarising the experience with Microsoft Research ZING model checker [1], it has been established that many subtle concurrency bugs may be manifested using a limited number of context switches. Based on that observation, context-switch-bounded approach has become a valid and respected formal verification technique for analysis of concurrent (multi-threaded) programs.

The technique has been analysed from various points of view. The problem of reachability analysis bounded in the number of context-switches has been shown decidable for pushdown systems [17]. The result has been later extended to the class of pushdown systems with dynamic memory [5] and systems with dynamic thread creation [2].

From a practical point of view, bounding the number of context switches plays an important role in the Microsoft Research model checker CHESS [13]. CHESS has shown that the model checking strategy of iteratively increasing the bound on the number of context switches is actually quite successful in discovering previously unknown concurrency-related errors [13]. The success of this approach has been further enhanced by its combination with partial order reduction [14].

Context-switch-bounded model checking has also been applied in the area of stateless model checking [8]. In [15], authors show how to incorporate a fair scheduler in a stateless exploration to increase efficiency of the CHESS model checker and to enable livelock detection in its stateless model checking mode.

More recent results include reports on experience of introducing context switch bounded mode of verification into the SPIN model checker [10] and a report on context-switch bounded model checking on multicore architectures [16].

Algorithm 1 Pseudocode of Context Switch Directed Reachability algorithm

```
 1: V_next ← {v_init}
 2: V_seen ← ∅
 3: Q ← empty queue
 4: while V_next ≠ ∅ do
 5:     Q ← V_next
 6:     V_next ← ∅
 7:     while Q not empty do
 8:         v ← pop(Q)
 9:         V_seen ← V_seen ∪ {v}
10:         if v is goal state then
11:             return "Property does not hold"
12:         for all edge e = (v, v') ∈ succ(v) do
13:             if v' ∉ V_seen then
14:                 if e does context-switch then
15:                     V_next ← V_next ∪ {v'}          ▷ will be expanded in next level
16:                 else
17:                     push(Q, v')                     ▷ will be expanded now
18: return "Property holds"
```

3 Algorithm

Our algorithm – Context Switch Directed Reachability, or CSDR for short – explores the state space in layers delimited by the number of context switches. Within any particular layer, the algorithm works like a parallel breadth-first search, except when it encounters a vertex which requires an extra context switch to get to. In such a case, this vertex is added to the set of states to be explored in the next layer, V_{next}. After an entire layer is explored, the algorithm restarts using V_{next} as the set of initial vertices. The pseudocode for a single-threaded version of the algorithm is given in Algorithm 1.

3.1 Implementation

Although the aforementioned algorithm seems pretty straight-forward, there are non-obvious subtleties in its implementation in DIVINE that require further analysis.

First of all, unlike other algorithms in DIVINE, CSDR requires support from the state space generator – it needs to detect whether context switch had occurred or not. State space generators do not track which thread ran last – that would differentiate otherwise equivalent states. This is undesirable for two reasons – firstly, it would dramatically increase the size of the state space; secondly, it would interfere with liveness verification. Nevertheless, the state space generator can provide a label for each transition, specifying which thread was responsible for that particular transition (this requires only one thread to move at once, which is true for the current LLVM generator, as well as for many others).

Using this transition label, the CSDR algorithm can detect context switches by associating an incoming thread-id with each state. However, since a given state may be reachable by different threads performing the last step, there is a trade-off between exactness, which would require us to associate a list of incoming thread ids with each state, and reduced memory use, when we store only one of the suitable thread-ids (clearly, we only need to choose among transitions that originate in the shallowest layer, i.e. with the minimal number of previous context switches). Since CSDR is already a heuristic algorithm (in the sense that its speedup is not guaranteed even if there is a counterexample in the model), we decided to implement the inexact version to avoid the increase in memory usage. More specifically, we save the first encountered thread-id and change it only if another encountered thread-id would allow us to reach the same state in fewer context switches (and therefore explore it immediately, instead of delaying it to the next layer).

The disadvantage of this inexact approach is that some states will not be reachable in their minimal number of context switches, but this problem is not very significant in an unbounded exploration (as states will be reached eventually). In bounded exploration, it can occasionally cause results to be nondeterministic, as the exploration order within one iteration, and therefore incoming thread-id associated with each state, can vary.

3.2 Complexity analysis

It is easy to see from the pseudocode that the asymptotic complexity is same as for any reachability analysis based on breadth-first search. However, we are more interested in real-life complexity of the implementation, as adding significant overhead to the model checking procedure is undesirable.

The handling of incoming thread-ids described in Section 3.1 is relatively cheap, since DIVINE stores states in a hash table in all algorithms. Also, adding a state to V_{next} can be done without extra overhead by using state-associated data. On the other hand, it is now required to iterate over the entire closed set at the start of each new iteration of the algorithm (to retrieve the associated information of all states and to queue those in V_{next} as initial states of the iteration). This has to be done as many times as there are iterations in the algorithm and therefore could cause significant overhead for models which have many small layers.

On the other hand, the CSDR algorithm can keep the open set smaller compared to a straightforward reachability implementation, since the states that belong to the next layer are not included in it. Therefore, at any point of the algorithm, the open set is at most as big as the current layer (while in an unrestricted breadth-first search, it can become as large as the entire state space). This can both decrease memory usage and possibly speed up exploration due to improved memory locality and decreased communication overhead from concurrent access to the open set.

Finally, there is a constant overhead of 8 bytes of associated data per state (thread-id and number or context switches). This might be significant when approaching limits of physical memory.

3.3 Extension to Full LTL Model Checking

The algorithm and the implementation we have discussed so far are limited to reachability analysis – that is, to model checking of *safety* properties. Since DIVINE provides full LTL verification, it is interesting to ask whether an algorithm for accepting cycle detection can be combined with context-switch bounding, or even better, if we can use a context-switch-directed search for liveness verification. Our first observation is that iteratively deepening the search bound and starting the algorithm from scratch will always work, in the sense that no spurious counterexamples will be produced. Of course, a counterexample may be missed unless the search is exhaustive. The question therefore is whether a better option is available.

It is easy to see that cutting a depth-first algorithm such as Nested DFS at a particular depth and re-starting it from boundary states is incorrect. Unfortunately, a simple argument shows that *no* algorithm for accepting cycle detection can avoid an unbounded revisit. Let us consider a state space that is generated by n threads that can only progress in a very constrained order. All the threads share a global atomic counter, and each thread increments the counter if (and only if) its value mod n is equal to the thread's identifier. When the counter reaches some value k it is reset to 0. Let the state in which the "reset" transition originates be an accepting state (this is easy to arrange). Now the distance of a vertex from the initial vertex is the same as the (minimal) number of context switches required to reach this particular state.

Nevertheless, a heuristic approach can be used to combine a context-switch-directed search for an accepting cycle using a non-repropagating variant of the MAP algorithm [6], in a way very similar to standard on-the-fly OWCTY [4]. Since this phase is entirely heuristic, it is not a problem when a counterexample is present but not discovered. When either the state space is fully explored or when the context-switch bound is encountered, a full accepting cycle discovery is executed. The heuristic context-switch-directed search is linear and hence the combined 2-phase algorithm has the same complexity as the algorithm used for accepting cycle detection in the second phase.

3.4 Related Issues

There are some non-obvious consequences of combining the CSDR algorithm with LLVM and pthreads support in DIVINE. First, starting a thread forces context switch from thread calling `pthread_create` to newly created thread. This also applies to models using C++11 thread interface as it is implemented in terms of pthreads.

Furthermore, the combination of the τ family of reductions [18] with context-switch directed search is also interesting from a theoretical point of view. Unlike more traditional partial order reduction techniques [14], the τ reductions combine with context-bounding very straightforwardly, since τ reduction always collapses transitions of a single thread (even though this may delay transitions of other threads, this does not affect the logic of tracking context switches). This is important as DIVINE employs τ reduction when model checking LLVM bitcode by default – the unreduced state space is extremely large even for very simple multi-threaded programs.

4 Evaluation

In this section, we evaluate context-switch-directed exploration and compare it with DIVINE's conventional reachability analysis, which uses a parallel breadth-first search. As CSDR introduces additional overhead when compared to straight reachability, we expect CSDR to be slightly slower in the cases when the model does not contain a counterexample. In the cases where the model contains a counterexample, we expect to get a speedup from CSDR at least in some cases, as it is expected that property violations do not usually require many context switches. However, we also expect to encounter models with a counterexample where CSDR will be slower, because they contain a short counterexample (therefore easily found by reachability), but with a large number of context-switches.

We have used several models, both with and without a counterexample. All of them were verified for the `safety` property of the LLVM generator, which combines all the supported error states (assertion violation, memory safety, arithmetic safety, memory leak detection and pthread mutex deadlock detection). Descriptions of the models we have used can be found in Table 1. In all cases, we executed DIVINE's `verify` command in shared memory mode with lossless compression enabled[1] and the tests were performed on a server-class computer with 2 Intel Xeon E5-2630 v2 CPUs running at 2.60 GHz and with 128 GB of main memory (since the maximal memory usage of all the models was less then 1 GB, none of them was affected by memory swapping). Time measurements were taken from DIVINE's report and represent wall time (that is, the total real time of the run).

We tested each model on 1 to 6 threads and we ran 20 tests with standard reachability and 20 with CSDR for each thread count[2]. As the exploration time can vary due to non-determinism caused by shared memory pseudo-BFS exploration (used in both normal reachability and in CSDR), our figures show the first and third quartiles and the median of the values measured in those 20 tests.

4.1 Results

For models without a counterexample (in Figures 1 and 2), CSDR has similar results as normal reachability, showing that the overhead of this exploration

[1] commandline options `--shared` and `--compression`
[2] options `--reachability` and `--csdr` respectively

barrier-1-bug	a simple one-time barrier based on a pthread conditional variable; the bug is in not expecting spurious wakeup (`pthread_cond_wait` is not running in a cycle)
barrier	correct one-time barrier using 1 conditional variable
barrier-1-re-bug	a buggy auto-resetting barrier, the reset cannot happen and the barrier stays locked after first use
barrier-1-re-bug-2	another buggy auto-resetting barrier, the reset can interleave with another thread waiting for the barrier again
barrier-1-re-bug-test	a correct implementation of an auto-resetting barrier, with a bug in the test
barrier-1-re	same as barrier-1-re-bug-test but with a fixed test
barrier-n-bug	a pthread barrier implemented using n conditional variables, bug is a typo that causes only one conditional variable to be signalled when unlocking the barrier
fifo-bug	an older version of DIVINE's IPC queue with a subtle race condition
fifo	a correct version of DIVINE's IPC queue
mutex-partial-deadlock	a model demonstrating pthread mutex deadlocks, three threads are started, each acquiring 2 mutexes to create a circular wait; an extra thread manipulates a global variable in an endless loop while holding no locks
mutex-partial-deadlock-2	another circular mutex wait example, 4 threads are running in endless loops locking 2 mutexes each; one of them can create a deadlock with either 1 or 2 other threads

Table 1. Description of benchmark models.

strategy is not significant. This might be partly due to fact that LLVM models tend to have relatively small state spaces in terms of number of states and the LLVM generator is fairly slow. Therefore, most time is spent in successor generation and the overhead introduced by CSDR (such as the iteration over the hash table at the beginning of each algorithm iteration) is small in comparison. In some cases, CSDR can be faster than normal reachability, even though it must explore the same number of states. This is most likely because it keeps the open set smaller and therefore decreases inter-thread communication and improves memory locality.

Models with a counterexample that we used show a variety of outcomes, ranging from 23-fold speedup for barrier-1-re-bug-2 (Figure 3) to 1000-fold slowdown in mutex-patial-deadlock-2 (Figure 7). It is to be expected that carefully crafted models could exhibit even more significant differences in favour of either of the algorithms. Therefore, detailed analysis of the models follows.

Table 2 shows statistics of counterexamples produced by standard reachability and by CSDR. It can be seen that CSDR produces fewer distinct results. This is likely due to the fact that many counterexamples produced by reachability have more context switches than counterexamples produced by CSDR; the effect which can also be seen in the table. The number of context switches in reachability counterexamples varies heavily and in many cases reachability does not

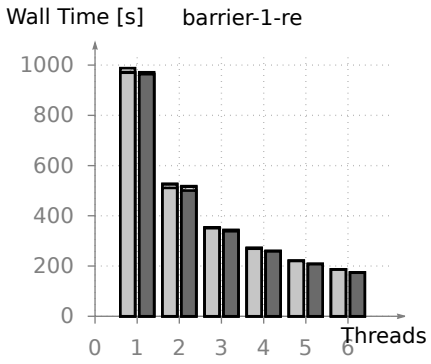

Fig. 1. Verification time of the barrier-1-re model (no counterexample).

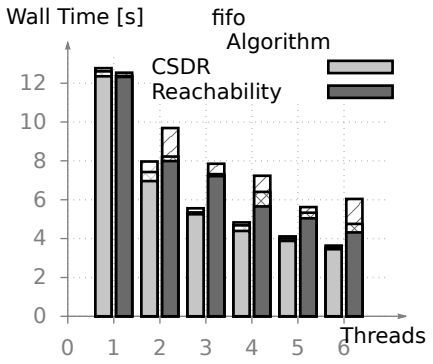

Fig. 2. Verification time of the fifo model, the correct version of an IPC queue from DIVINE (no counterexample).

model	# of different CEs		CE length		# of CSs in CEs	
	CSDR	reachability	CSDR	reachability	CSDR	reachability
barrier-1-bug	1	3	22	22 – 27	1	1 – 4
barrier-1-re-bug-2	1	97	68	65 – 98	4	8 – 18
barrier-1-re-bug	4	99	90 – 91	90 – 98	12	16 – 28
barrier-1-re-bug-test	4	100	120	120 – 123	8	14 – 31
barrier-n-bug	1	41	38	38 – 53	1	1 – 10
fifo-bug	8	48	165 – 171	164 – 172	4	10 – 16
mutex-part-deadlock	1	56	29	23 – 31	8	9 – 15
mutex-part-deadlock-2	13	90	32 – 78	22 – 37	9	10 – 17

Table 2. Number of different counterexamples, counterexample length, and number of context switches in counterexamples for models with a counterexample. Each algorithm + model configuration ran through 120 tests.

find a counterexample with minimal number of context switches. On the other hand, CSDR matches the reachability's ability to find short counterexample quite well, in all but one case falling into range of lengths produced by reachability. This is good as keeping counterexamples short is almost as important for their readability as a low number of context switches.

In the worst performing model, mutext-patial-deadlock-2, there are 5 threads, one of them is running independently of the other 4, which are all running an endless loop: locking two mutexes, incrementing a global variable (modulo 3) and unlocking the mutexes again. There are three mutexes in the model, and each thread is assigned two of them in a manner that one of the threads can cause a circular wait with one or two of the other threads. Since each of the mutex-locking threads runs in an endless loop, and CSDR will not make a context switch unless it is necessary, it will first explore all the states the first thread can be in, then all combinations of the first and the second thread and so on. As the thread which can deadlock with others is launched last, CSDR will first explore many possible interactions of other threads before it can find the deadlock. This

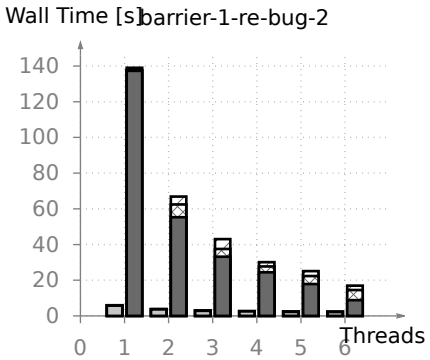

Fig. 3. Verification time of the barrier-1-re-bug-2 model, which shows significantly better results when using CSDR.

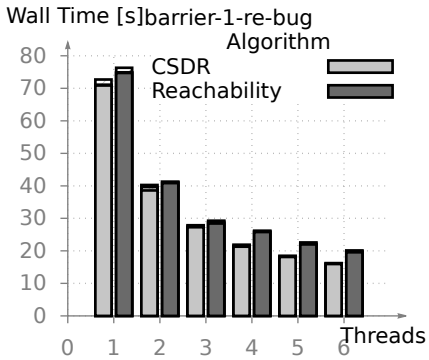

Fig. 4. Verification time of the barrier-1-re-bug model.

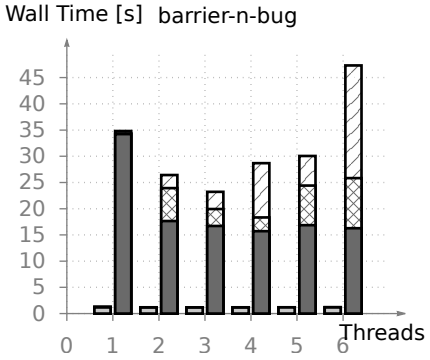

Fig. 5. Verification time of the barrier-n-bug model.

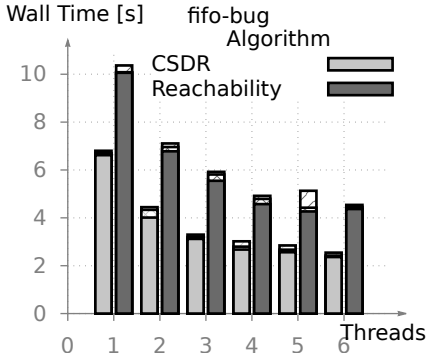

Fig. 6. Verification time of the fifo-bug model, an IPC queue from DIVINE.

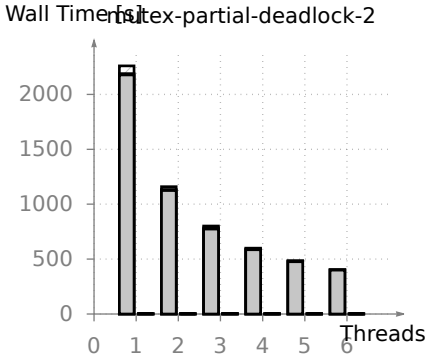

Fig. 7. Verification time of the mutex-partial-deadlock-2 model, reachability performs much better than CSDR (times for reachability are around 2 seconds).

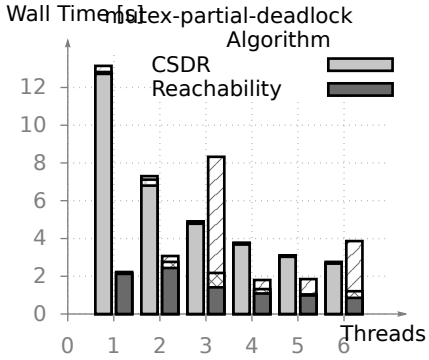

Fig. 8. Verification time of the mutex-partial-deadlock model.

means a much larger part of the model will be explored by CSDR – indeed CSDR explored roughly 830 thousand states of the total 19 milion, while reachability explored only 2 to 6 thousand states. CSDR needed 9 context switches, while the counterexamples from reachability had 10 to 17 context switches and the model can be fully explored in 53 context switches.

In general, CSDR is likely to perform worse than reachability if the shortest counterexample is short and a lot of work which does not lead towards a goal state can be done with few context switches. That is, if the error state can be reached only via many context switches.

It is important to note that there is no bound on how much CSDR can be slower than reachability. If we imagine a model in which an assertion violation is reachable by two context switches, but the state space limited to one context switch is infinite, CSDR will fail to terminate, even though reachability will find the assertion violation. Such a model can be crafted easily in C: it is sufficient to start a thread which will do some infinite-state work (such as incrementing a global variable forever)[3] and put `assert( false )` right after the call to `pthread_create`. However, it is important to note that the commonly used depth-first reachability has the same problem; moreover, verification of infinite-state models using (unassisted) explicit-state model checking is impossible as the model checker can terminate only if the model contains an error.

On the other hand, the barrier-1-re-bug-2 model performs much better with CSDR than with reachability. CSDR explores about 890 states in 4 context switches, while reachability explores 20 to 35 thousand. Exploring the entire model is infeasible in this case as there is a 32 bit variable which is repeatedly incremented. This model is well-suited for CSDR verification – it has three threads which test an auto-resetting barrier implemented using a pthread conditional variable. Even though these threads run in an infinite loop, the earlier-started threads will become blocked on the barrier, which will force CSDR to allow a context switch as there are no more states available otherwise. Furthermore, the thread which unlocks the barrier is also the one to trip the assertion if it continues running (although any other thread can trip this assertion too).

5 Conclusion

We have described an algorithm for context-switch-directed verification suitable for implementation in a parallel, explicit-state model checker. We have implemented the algorithm based on the design outlined in this paper and used this implementation to evaluate its properties. Our experiments show that directing the counterexample search based on context switch count is a useful strategy that improves both counterexample quality and counterexample discovery speed. We have briefly considered the extensions of the algorithm for full LTL model checking and found a favourable heuristic compromise to speed up discovery of liveness counterexamples. Finally, we gave a detailed analysis of various model

[3] Strictly speaking, this is not infinite-state behaviour as the global variable will overflow, but from the perspective of model checking, reaching this overflow is unfeasible.

types from the perspective of context switching and process interleaving, and the behaviour of context-switch-driven exploration on these models.

References

1. T. Andrews, S. Qadeer, S. K. Rajamani, and Y. Xie. Zing: Exploiting Program Structure for Model Checking Concurrent Software. In *CONCUR*, volume 3170 of *LNCS*, pages 1–15. Springer, 2004.
2. M. F. Atig, A. Bouajjani, and S. Qadeer. Context-Bounded Analysis for Concurrent Programs with Dynamic Creation of Threads. In *TACAS*, volume 5505 of *LNCS*, pages 107–123. Springer, 2009.
3. J. Barnat, L. Brim, V. Havel, and J. Havlíček et al. DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs. In *CAV*, volume 8044 of *LNCS*, pages 863–868. Springer, 2013.
4. J. Barnat, L. Brim, and P. Ročkai. On-the-fly Parallel Model Checking Algorithm that is Optimal for Verification of Weak LTL Properties. *Science of Computer Programming*, 77(12):1272–1288, 2012.
5. A. Bouajjani, S. Fratani, and S. Qadeer. Context-Bounded Analysis of Multithreaded Programs with Dynamic Linked Structures. In *CAV*, volume 4590 of *LNCS*, pages 207–220. Springer, 2007.
6. L. Brim, I. Černá, P. Moravec, and J. Šimša. Accepting Predecessors are Better than Back Edges in Distributed LTL Model-Checking. In *FMCAD'04*, volume 3312 of *LNCS*, pages 352–366. Springer, 2004.
7. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT press, 1999.
8. P. Godefroid. Model Checking for Programming Languages using Verisoft. In *POPL*, pages 174–186. ACM Press, 1997.
9. G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.
10. G. J. Holzmann and M. Florian. Model checking with bounded context switching. *Formal Aspects of Computing*, 23(3):365–389, 2011.
11. G. J. Holzmann, R. Joshi, and A. Groce. Swarm verification techniques. *IEEE Transactions on Software Engineering*, 37(6):845–857, 2011.
12. A. W. Laarman, J. C. van de Pol, and M. Weber. Multi-Core LTSmin: Marrying Modularity and Scalability. In *NFM*, volume 6617 of *LNCS*, pages 506–511. Springer, 2011.
13. M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *ACM SIGPLAN, PLDI*, PLDI '07, pages 446–455, New York, NY, USA, 2007. ACM.
14. M. Musuvathi and S. Qadeer. Partial-Order Reduction for Context-Bounded State Exploration. Technical Report MSR-TR-2007-12, Microsoft Research, 2007.
15. M. Musuvathi and S. Qadeer. Fair Stateless Model Checking. *ACM SIGPLAN Notices*, 43(6):362–371, 2008.
16. S. Nagarakatte, S. Burckhardt, M. M. K. Martin, and M. Musuvathi. Multicore acceleration of priority-based schedulers for concurrency bug detection. In *ACM SIGPLAN, PLDI*, pages 543–554. ACM, 2012.
17. S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In *TACAS*, TACAS'05, pages 93–107, Berlin, Heidelberg, 2005. Springer.
18. P. Ročkai, J. Barnat, and L. Brim. Improved State Space Reductions for LTL Model Checking of C & C++ Programs. In *NFM*, volume 7871 of *LNCS*, pages 1–15. Springer, 2013.