Lazy-CSeq-SP: Boosting Sequentialization-based Verification of Multi-Threaded C Programs via Symbolic Pruning of Redundant Schedules[⋆]

Vladimir Herdt[1], Hoang M. Le[1], Daniel Große[1], and Rolf Drechsler[1,2]

[1] Group of Computer Architecture, University of Bremen, 28359 Bremen, Germany
{vherdt,hle,dgrosse,drechsle}@cs.uni-bremen.de
[2] Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany

Abstract. Sequentialization has been shown to be an effective symbolic verification technique for concurrent C programs using POSIX threads. Lazy-CSeq, a tool that applies a lazy sequentialization scheme, has won the Concurrency division of the last two editions of the Competition on Software Verification. The tool encodes all thread schedules up to a given bound into a single non-deterministic sequential C program and then invokes a C model checker. This paper presents a novel optimized implementation of lazy sequentialization, which integrates symbolic pruning of redundant schedules into the encoding. Experimental evaluation shows that our tool outperforms Lazy-CSeq significantly on many benchmarks.

Keywords: Formal Verification, Concurrency, Sequentialization

1 Introduction

Verifying concurrent programs is a difficult problem, due to the large state space caused by all possible thread schedules. Context-bounded analysis (CBA) [12,13] in combination with sequentialization [14,11,10], especially the lazy sequentialization scheme employed by Lazy-CSeq [8,9], has been shown to be a particular effective symbolic verification technique for concurrent C programs using POSIX threads. Lazy-CSeq has won the Concurrency division of the last two editions of the Competition of Software Verification (SV-COMP) [4,5]. Essentially, Lazy-CSeq works as follows: It transforms the multi-threaded program into an unrolled form, by inlining functions, unwinding loops and cloning instantiated threads into thread functions. The existing *main()* function becomes a thread and is replaced by a new *main()* function, which contains a round-robin scheduler. In each round, each enabled thread is executed for a non-deterministic number of steps. For this execution, context switch logic is placed inside the thread functions. It allows to preempt a thread execution at statements that can interfere with

[⋆] This work was supported in part the German Federal Ministry of Education and Research (BMBF) within the project EffektiV under contract no. 01IS13022E and by the German Research Foundation (DFG) within the Reinhart Koselleck project DR 287/23-1.

other threads and therefore ensures that all relevant behaviors will be explored. The local state of the threads is preserved across function calls by making local variables static and initializing them to non-deterministic values to preserve the original semantics. Finally, the scheduler is bounded to execute a fixed number of rounds to create a single non-deterministic bounded sequential C program, which is then verified using a sequential C model checker. More details can be found in [8].

This paper presents a novel optimized round-robin scheduler encoding that incorporates symbolic pruning of redundant schedules and is used as replacement for the Lazy-CSeq scheduler. Furthermore, an extensive static analysis is employed to reduce the number of context switches placed within the threads, by detecting interfering statements more accurately. Experimental evaluation shows that both techniques can boost the performance of lazy sequentialization significantly.

2 Optimized Scheduler Encoding

Basic Round-Robin Scheduler Encoding A round-robin scheduler, as employed by Lazy-CSeq, encodes a fixed number of thread execution rounds, thus effectively implements a variant of CBA. In every round all enabled threads are executed in a representative total order, each one for a non-deterministic number of steps. This non-determinism models the preemptive semantics of POSIX threads where a thread execution can be arbitrarily preempted by another thread. Thus, in each round, there are three possibilities for a thread: *1.* It is skipped (zero execution step); *2.* It is executed to completion (in this case, the thread becomes disabled and will not be executed in the next rounds); *3.* It is executed up to a point where its execution can be resumed in the next round; In the following the number of threads is denoted as m. Every thread is assigned a unique index $i \in \{1, .., m\}$ according to the representative execution order, and we define $t_i < t_j$ iff $i < j$. A schedule is denoted as a sequence of thread executions with the symbol $\mid$ as round delimiter. Skipped threads in each round will be omitted for convenience.

Encoding Symbolic Pruning of Redundant Schedules The main idea of the encoding is based on *round merging.* Consider $m = 3$ and the schedule $t_3 \mid \; \mid t_2$ (all threads are skipped in the second round). The three rounds can be merged to $t_3 \mid t_2$, but $t_3 \mid t_2$ cannot be merged further, because it is not possible to execute t_3 before t_2 in the same round. Another schedule $t_1 t_2 \mid t_2 t_3$ can be merged into a single round $t_1 t_2 t_3$, because in the second round, t_2 resumes at the point where it stopped before. For every mergeable schedule, there exists an equivalent unmergeable schedule. Thus mergeable schedules are redundant and our encoding aims to prune them. The key observation is the scheduler is only required to maintain a very simple invariant to do so: before t_j is considered, the last executed thread (with non-zero steps) must not be t_j. This invariant ensures that no more than $m - 1$ subsequent thread executions are skipped. Thus, it eliminates all empty rounds and ensures that a new round starts with a thread $t_j < t_i$ which ended the last round, as they could otherwise be merged. This enables a very lightweight encoding of round merging, which only introduces a single new variable to keep track of the last executed thread, and a set of assumptions. Fig. 1 shows a comparison of a single execution of a thread t_j for a non-deterministic number of steps in Lazy-CSeq (left side) and Lazy-CSeq-SP (right side). The functions *guess_pc* and *NC(j)* return a non-deterministic non-negative value and the number of context switches in t_j, respectively. The variables *pc[j]* and *pc_cs* store the current location in t_j and the location of the next context switch, respectively.

```
thread_index = j;
if (active_thread[j] == 1) {
    pc_cs = pc[j] + guess_pc();
    assume (pc_cs >= 0 &&
                pc_cs <= NC(j));
    thread_j (threadargs[j]);
    pc[j] = pc_cs;
}
```

```
thread_index = j;
assume (j != last_thread_index);
if (active_thread[j] == 1) {
    pc_cs = guess_pc();
    assume (pc_cs <= NC(j));
    if (pc_cs > pc[j]) {
        thread_j (threadargs[j]);
        pc[j] = pc_cs;
        last_thread_index = j;
    }
}
```

Fig. 1. Execution of a single thread in Lazy-CSeq (left) and Lazy-CSeq-SP (right)

3 Implementation Details

We have implemented our tool Lazy-CSeq-SP in Python (version 3.4). A web-interface is available at `www.systemc-verification.org/LazyCSeqSP` for evaluation. An architecture overview is shown in Fig. 2. It expects a C program sequentialized by Lazy-CSeq, which is then further processed in two steps as shown in the lower part of Fig. 2, to produce an optimized C program. In the first step the context switch logic inside the threads, the main function with the scheduler encoding and some other auxiliary definitions generated by Lazy-CSeq are removed by using some simple scripts with regular expression matching. Essentially this results in a program, where all functions (except those marked atomic) have been inlined, loops unwound, and all threads cloned the number of times they are instantiated (the number of *pthread_create* statements is statically known due to the inlining and unwinding performed by Lazy-CSeq). In the second step a static analysis is applied to collect informations, which are then used to compute locations for the context switches. Our analysis is more accurate than Lazy-CSeq and often generates a smaller number of context switches, which also reduces the state space.

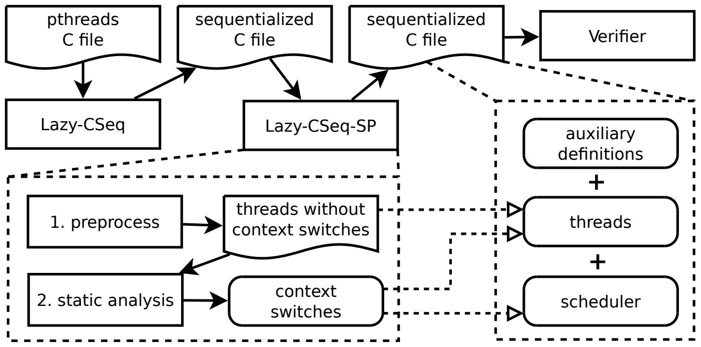

Fig. 2. Tool implementation overview

Reducing Context Switches by Static Analysis Essentially, the static analysis works as follows: First, the *pycparser* library is used to parse the preprocessed sequentialized file into an AST, which is then transformed and annotated in subsequent phases. This includes a declaration analysis (binding names to declarations), a simple type analysis, a pointer analysis [3,7] and an escape analysis. The collected informations are then used to compute the following sets of data (called *effects*) for every statement: reads, writes, assumptions and *pthread_join* calls. Based on these statement effects a context switch is placed before a statement s if one of the following conditions holds:

1. s has a read-write dependency with any statement from another thread, i.e. both access the same variable with at least one of them writing;
2. s contains a *pthread_join* call or an assumption;
3. s is the initial statement of a thread.

The first condition can be efficiently checked by first combining all statement effects for every thread separately. The third condition ensures that initial non-interfering statements of a thread are not re-executed, when the thread resumes its execution. As an optimization, no context switches are placed due to condition (1) or (2) in the main function thread until a *pthread_create* statement is reached, since initially only the main function thread is enabled. Please note that no context switches are placed before any *pthread_create* statement at all, since creating a new thread does not interfere with the execution of the already available threads. The reason is that enabling another thread can only add additional behaviors, not limit existing ones. Furthermore, only a single context switch is placed before a group of multiple consecutive *pthread_join* calls. Once every created thread has been joined, no more context switches will be placed. Other (*pthread*) library calls are naturally handled with the first condition, e.g. a *pthread_mutex_lock* call is modeled to have a write access to the mutex argument. Finally no context switches are placed before (loop unwinding) assumptions, can be proven to be satisfiable (and have no other dependencies), since they cannot terminate the program execution.

4 Experimental Evaluation and Conclusion

We have evaluated Lazy-CSeq-SP on benchmarks from the Concurrency division of the SV-COMP 2015 [2]. All experiments are performed on a 3.4 GHz AMD machine running Linux. The time and memory limits are set to 600 seconds and 4GB, respectively. The abbreviation T.O. denotes that the time limit has been exceeded. The runtime results in seconds for a set of representative benchmarks are shown in Table 1. A table including results for a larger set of benchmarks is also available at `www.systemc-verification.org/LazyCSeqSP`.

Table 1. Benchmark results

Benchmark	Lazy-CSeq	+SA	Lazy-CSeq-SP
fib_bench_longest_false.c.r11.u11	278.157	166.388	**12.061**
fkp2013_false.c.r2.u50	4.265	**3.934**	4.298
qw2004_false.c.r1.u1	0.250	**0.238**	0.240
sigma_false.c.r1.u16	7.523	**4.087**	4.244
fib_bench_longest_true.c.r11.u11	243.630	221.486	**15.621**
fk2012_true.c.r3.u8	T.O.	**54.875**	56.358
fkp2013_true.c.r3.u48	529.447	**1.895**	2.206
fkp2014_true.c.r5.u5	31.913	12.884	**12.599**
queue_ok_true.c.r4.u20	337.787	278.580	**65.224**
queue_ok_true.c.r5.u20	T.O.	T.O.	**97.408**
read_write_lock_true.c.r30.u1	311.012	159.506	**5.415**
sssc12_true.c.r3.u7	123.239	76.220	**71.479**
stack_true.c.r5.u5	67.071	21.541	**10.529**

The table shows three different configurations: the original Lazy-CSeq (version 2.0beta), our implementation with optimized context switch placement by static analysis (+SA), and additionally with symbolic pruning of redundant schedules (Lazy-CSeq-SP). To ensure a fair comparison, we patched Lazy-CSeq to place context switches before (potentially unsatisfiable) *while-loop* unwinding assumptions. Otherwise the Lazy-CSeq unwinding would effectively be reduced by one, since the last execution of the loop body would not be considered. The table is divided into two halves. The upper (lower) half shows the results on the unsafe (safe) benchmarks. The number of rounds and the unwinding are encoded into the benchmark name. CBMC version 5.0 is used as backend [1,6]. Only the CBMC runtimes are reported, since the time required to generate the sequentialized file is comparatively negligible.

On the presented benchmarks, clear improvements can be observed for the configuration +SA compared to Lazy-CSeq. Lazy-CSeq-SP improves the results further with some exceptions, where +SA shows better results. These are due to the simpler encoding of +SA, but the runtime differences are not significant. The results clearly show the advantages of symbolic pruning of redundant schedules and encourage further research in this direction. For future work we plan to investigate more aggressive pruning, e.g. symbolic Partial Order Reduction [15].

References

1. CBMC 5.0. http://www.cprover.org/cbmc/download/cbmc-5-0-linux-64.tgz.
2. SV-COMP 2015. http://sv-comp.sosy-lab.org/2015/.
3. L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, 1994.
4. D. Beyer. Status report on software verification - (competition summary SV-COMP 2014). In *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, pages 373–388, 2014.
5. D. Beyer. Software verification and verifiable witnesses - (report on SV-COMP 2015). In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, pages 401–416, 2015.
6. E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.
7. B. Hardekopf and C. Lin. The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 290–299, New York, NY, USA, 2007. ACM.
8. O. Inverso, E. Tomasco, B. Fischer, S. La Torre, and G. Parlato. Bounded model checking of multi-threaded c programs via lazy sequentialization. In A. Biere and R. Bloem, editors, *Computer Aided Verification*, volume 8559 of *Lecture Notes in Computer Science*, pages 585–602. Springer International Publishing, 2014.
9. O. Inverso, E. Tomasco, B. Fischer, S. L. Torre, and G. Parlat. *Lazy-CSeq 0.6c: An Improved Lazy Sequentialization Tool for C.* University of Southampton, 2014.
10. S. La Torre, P. Madhusudan, and G. Parlato. Reducing context-bounded concurrent reachability to sequential reachability. In A. Bouajjani and O. Maler, editors, *Computer Aided Verification*, volume 5643 of *Lecture Notes in Computer Science*, pages 477–492. Springer Berlin Heidelberg, 2009.
11. A. Lal and T. Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Form. Methods Syst. Des.*, 35(1):73–97, Aug. 2009.
12. M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 446–455, New York, NY, USA, 2007. ACM.
13. S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. In N. Halbwachs and L. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *Lecture Notes in Computer Science*, pages 93–107. Springer Berlin Heidelberg, 2005.
14. S. Qadeer and D. Wu. Kiss: Keep it simple and sequential. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 14–24, New York, NY, USA, 2004. ACM.
15. C. Wang, Z. Yang, V. Kahlon, and A. Gupta. Peephole partial order reduction. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 382–396, Berlin, Heidelberg, 2008. Springer-Verlag.