Chiefly Symmetric: Results on the Scalability of Probabilistic Model Checking for Operating-System Code *

Christel Baier[1], Marcus Daum[1], Benjamin Engel[2], Hermann Härtig[2], Joachim Klein[1]
Sascha Klüppelholz[1], Steffen Märcker[1], Hendrik Tews[2], Marcus Völp[2]
[1]Institute for Theoretical Computer Science and [2]Operating-Systems Group
Technische Universität Dresden, Germany

Reliability in terms of functional properties from the safety-liveness spectrum is an indispensable requirement of low-level operating-system (OS) code. However, with evermore complex and thus less predictable hardware, quantitative and probabilistic guarantees become more and more important. Probabilistic model checking is one technique to automatically obtain these guarantees. First experiences with the automated quantitative analysis of low-level operating-system code confirm the expectation that the naive probabilistic model checking approach rapidly reaches its limits when increasing the numbers of processes. This paper reports on our work-in-progress to tackle the state explosion problem for low-level OS-code caused by the exponential blow-up of the model size when the number of processes grows. We studied the symmetry reduction approach and carried out our experiments with a simple test-and-test-and-set lock case study as a representative example for a wide range of protocols with natural inter-process dependencies and long-run properties. We quickly see a state-space explosion for scenarios where inter-process dependencies are insignificant. However, once inter-process dependencies dominate the picture models with hundred and more processes can be constructed and analysed.

1 Introduction

Safety-critical software in space, flight, and automotive control systems need not only produce correct results. It must produce these results in time and, to the extent possible, despite component failures. Worst-case execution-time (WCET) analyses (cf., [3, 16, 23]) are able to produce the former kind of guarantees whereas reliability techniques (cf., [22, 4, 7]) are designed to rule out negative effects from the latter. However, WCET analyses only produce guarantees in the form of upper bounds on the execution times of all involved components, which hold even in the most extreme situations. Yet, most computer systems are either not safety critical or they include fail safe mechanisms that prevent damage in highly exceptional situations. For such systems, it is much more appropriate to look at the 99.9% quantile of the execution time (i.e., the execution time that is not exceeded with a probability of 99.9%), ignoring exceptional cases, that are dealt with by other means anyway.

Many software systems, especially operating systems, contain optimisations that are geared towards the average case, incurring additional costs under exceptional circumstances. To justify such optimisations one has to look at the probability at which the optimisation is advantageous as well as the additional costs that apply with a (hopefully) low probability. Here it is again often more appropriate to investigate

*This work was in part funded by the German Research Council (DFG) through the QuaOS project, the collaborative research center 912 Highly-Adaptive Energy-Efficient Computing (HAEC), the European Union Seventh Framework Programme under grant agreement no. 295261 (MEALS), the DFG/NWO-project ROCKS, the cluster of excellence cfAED (center for Advancing Electronics Dresden) and by the European Union and the state Saxony through the ESF young researcher group IMData 100098198.

F. Cassez, R. Huuck, G. Klein and B. Schlich (eds.):
Systems Software Verification Conference 2012 (SSV 2012)
EPTCS 102, 2012, pp. 156–166, doi:10.4204/EPTCS.102.14

a suitable quantile of the additional costs, because pathological cases are either not interesting or solved by other means.

In previous work [2], we presented a combined simulation and model checking based approach to determine the probabilistic quantitative properties similar to what we described in the two preceding paragraphs. As a part of this first case study we have extended the model checker PRISM [17] with an operator for steady-state dependent properties and performed an analysis of the long run behaviour of a simple test-and-test-and-set spinlock [1]. However, scaling the number of processes competing for the lock proved difficult, as the state space to be considered by the model checker rapidly increases.

In this paper, we report on our experience in applying symmetry reduction [6, 10, 13, 11] to mitigate the state space explosion and thus scale our spinlock analysis. As our model is highly symmetrical, symmetry reduction techniques allow the analysis to be carried out on a potentially much smaller quotient model instead of the full model. To evaluate the benefit of applying symmetry reduction to our model and to gain further insight about its behaviour for an increasing number of processes, we have implemented a specialised tool that explores the reachable state space of the symmetry reduced quotient model and generates the transition matrix as input for a probabilistic model checker to carry out its analysis. This approach yielded a drastic improvement in the scalability of our analysis, in some cases allowing an analysis of models with 10,000 processes and more in contrast to the 4 to 5 processes achievable with the non-reduced model.

Outline. In the next section, we introduce our general approach, present our model and demonstrate the scalability problems of the non-reduced model. Then, in Section 3 we discuss the symmetry reduction techniques that we have applied. We interpret the drastic improvements in scalability and the results of the analysis for an increasing number of processes. Section 4 concludes.

2 The QuaOS Approach for Probabilistic Quantitative Properties

In this section, we first give a short overview of the general approach that we use in the QuaOS project for determining probabilistic quantitative properties of operating-system code. Then, we introduce our cache-aware model of the test-and-test-and-set spinlock.

One very difficult task in modelling the quantitative behaviour of low-level operating-system code is to design the necessary abstractions such that the model agrees in its behaviour with reality. Our approach is to compare the model checking results with measurements from specifically designed test cases. In case there are deviations, we investigate the reasons and adjust the model and, if necessary, the test cases to extract further details. For [2] we found that a specific effect of cache coherence has a substantial influence on the timing of spinlocks.

Our ultimate aim is to make predictions for hardware configurations that do not yet exist. With our approach, we could for instance investigate which synchronisation primitives are suitable for systems with more than 100 cores, depending on the work load and the performance of certain memory operations.

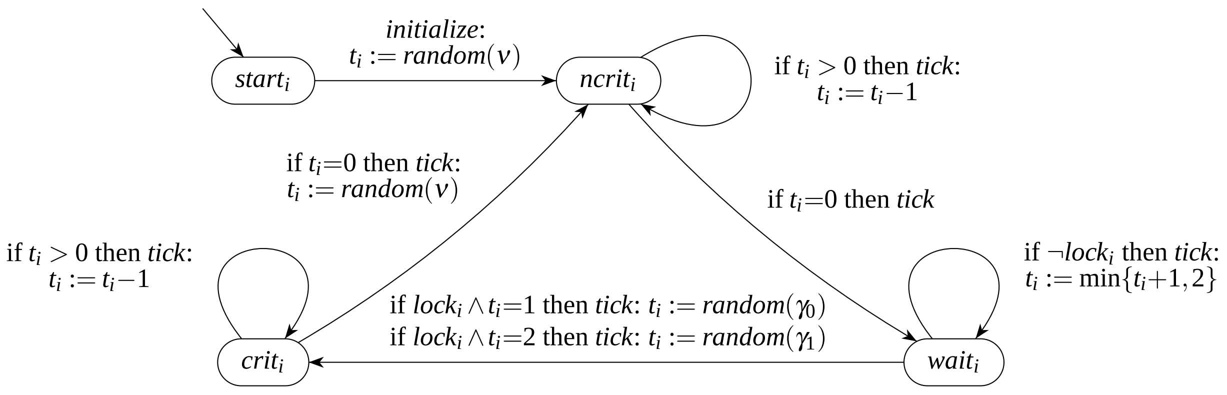
A probabilistic cache-aware model of test-and-test-and-set spinlocks. Figure 1 shows the source code for a test-and-test-and-set lock [1]. In the following, we refer to this lock simply as spinlock. The `atomic_swap` in line 3 writes the second argument into the location of the first argument and returns the old value of the first argument. Therefore, the lock has been acquired, if `atomic_swap` returns `false`. Otherwise, the caller spins in line 4 until the lock becomes free before it executes the more expensive `atomic_swap` again. Note that each access of the shared variable `occupied` might cause the transfer of the respective cache line into the core-local cache.

In our investigation we consider n identical parallel processes P_i that use the spinlock to synchronise their critical sections. Each process P_i performs an uncritical section and a critical section in an endless loop. The durations of the uncritical and the critical sections are determined by probabilistic distributions, which are parameters in our model.

```
1 volatile bool occupied = false;
2 void lock(){
3    while(atomic_swap(occupied,true)){
4       while(occupied){}
5    }}
6 void unlock(){
7    occupied = false
8 }
```

Figure 1: Pseudo Code for a TTS spinlock

In [2] we developed a discrete time Markov chain (DTMC) model that results from the synchronous parallel composition of n processes P_i (Figure 2) and the spinlock itself (Figure 3). Having a separate process for the spinlock facilitates the uniform probabilistic choice of the process that acquires the lock next, in case several processes are spinning.

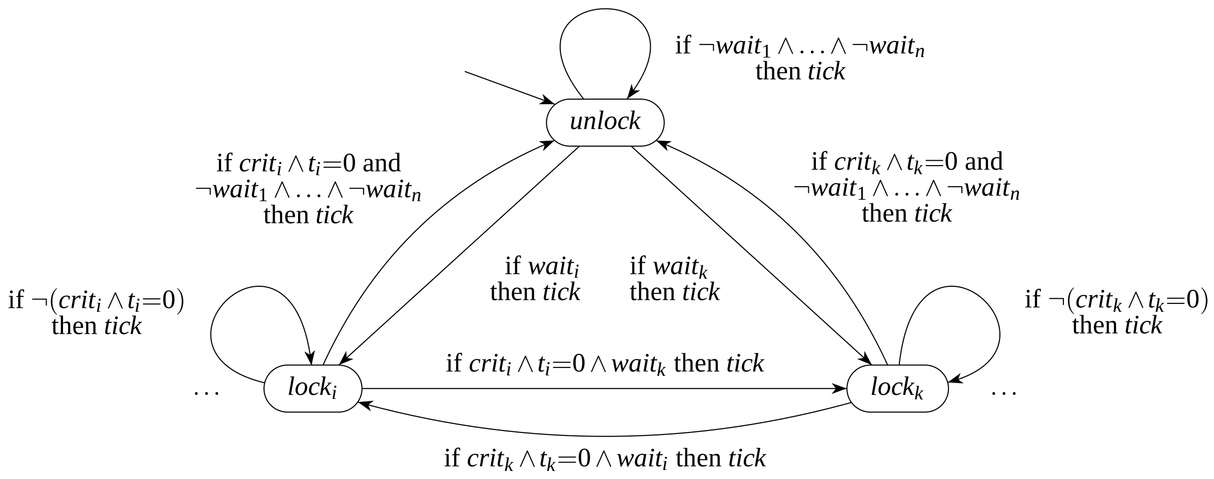


initialize:
$t_i := random(\nu)$
$start_i$
$ncrit_i$
if $t_i > 0$ then *tick*:
$t_i := t_i - 1$
if t_i=0 then *tick*:
$t_i := random(\nu)$
if t_i=0 then *tick*
if $t_i > 0$ then *tick*:
$t_i := t_i - 1$
if $\neg lock_i$ then *tick*:
$t_i := \min\{t_i+1, 2\}$
if $lock_i \wedge t_i$=1 then *tick*: $t_i := random(\gamma_0)$
if $lock_i \wedge t_i$=2 then *tick*: $t_i := random(\gamma_1)$
$crit_i$
$wait_i$

Figure 2: Control flow graph of process P_i

The control flow graph for each process P_i (see Figure 2) has the location *ncrit* for the non-critical section, *wait* for the time it waits for the lock, *crit* for the critical section and *start* for initialisation. In the locations *ncrit* and *crit* the variable t_i is a timer that determines how long the process has to remain in the respective location. The time for the non-critical section is determined by the distribution ν. The time for the critical section length is determined by the distributions γ_0 and γ_1. In the location *wait* the variable t_i records whether the lock could be entered immediately ($t_i = 1$) or whether spinning was necessary before obtaining the lock ($t_i = 2$).

We need the two distributions γ_0 and γ_1 to account for the following cache effect. In the usual case where the last lock owner was a different process (on a different core), the atomic_swap in Line 3 (see Figure 1) transfers the cache line containing the variable occupied into the local cache with state *modified*. In case the process was spinning when the lock was released, Line 4 causes an additional inter-core message that transfers the respective cache line into state *shared*. This additional inter-core message delays processes that were spinning before acquiring the lock by about 200 cycles. For simplicity we model this delay by using the distribution γ_1 for the critical section, which offsets γ_0 by this delay. For small critical sections (in the order of 1000 cycles) the delay significantly changes the behaviour of the complete system. The values for the distribution ν are chosen to be at least one order of magnitude larger than γ_0. Smaller ratios between critical and uncritical are uncommon in reality. In [2] we compare our



if $\neg wait_1 \wedge \ldots \wedge \neg wait_n$ then *tick*
unlock
if $crit_i \wedge t_i{=}0$ and $\neg wait_1 \wedge \ldots \wedge \neg wait_n$ then *tick*
if $crit_k \wedge t_k{=}0$ and $\neg wait_1 \wedge \ldots \wedge \neg wait_n$ then *tick*
if $wait_i$ then *tick*
if $wait_k$ then *tick*
if $\neg(crit_i \wedge t_i{=}0)$ then *tick*
if $\neg(crit_k \wedge t_k{=}0)$ then *tick*
if $crit_i \wedge t_i{=}0 \wedge wait_k$ then *tick*
. . .
$lock_i$
$lock_k$
. . .
if $crit_k \wedge t_k{=}0 \wedge wait_i$ then *tick*

Figure 3: Control flow graph of the spinlock

results with a model that uses only one distribution for the critical section.

To reduce state-space size, we scale down the numerical values of all distributions as far as possible while maintaining their ratios. The main example in this paper will be parametrised with the distributions $\gamma_0(5) = 1$, $\gamma_1(6) = 1$ and $\nu(40) = \nu(50) = \frac{1}{2}$.

The interaction of P_i with the spinlock (see Figure 3) is a bit subtle. When P_i moves into state $wait_i$ it signals the spinlock its desire to acquire the lock. In case the lock is free (state *unlock*) the spinlock moves *in the next step* to state $lock_i$, while P_i stays in $wait_i$. Having acquired the lock, P_i moves into state $crit_i$ one step later. The spinlock has additional transitions between all the n *lock* states to permit a direct change from $lock_i$ to $lock_k$.

In [2] we have formalised four conditional long run properties, such as the probability to acquire the lock without waiting, the expected average waiting time, and the 95% quantile of this waiting time, and computed their results with the model checker PRISM. With PRISM, we were able to obtain results for all the considered distributions only for up to 4 processes, with 5 processes results for the simpler distributions were achievable. For more processes, PRISM did not terminate within several days.

Table 1 summarises the statistics for the model with our example distributions. All calculations in the paper were carried out using a dual-socket Intel Xeon L5630 (Quad-Core) machine at 2.13 GHz with 192 GB total amount of RAM. One can clearly see that increasing the number of processes in our original model significantly increases the complexity, resulting in a very fast blowup in the number of reachable

Processes	Reachable states	MTBDD size	Time for building	Time for calculating steady state probabilities
3	67,001	32,772	4.05 s	3.45 s
4	4,082,808	569,042	316.31 s	149.10 s
5	198,808,720	7,632,342	≈ 20 hours	≈ 90 minutes

Table 1: Statistics for building the model (with $\gamma_0(5) = 1$, $\gamma_1(6) = 1$ and $\nu(40) = \nu(50) = \frac{1}{2}$) and calculating the steady state probabilities for n processes using the model checker PRISM with the sparse engine. The MTBDD size column lists the number of nodes used to store the symbolic transition relation in PRISM.

states as well as in the time spent analysing the model. In this paper we mainly focus on the two factors – building the model and calculating the steady state probabilities – that dominate the analysis time of the investigated conditional long run properties. We will now detail our findings in applying symmetry reduction as a method to scale the analysis.

3 Applying Symmetry Reduction to the Spinlock Model

Our model exhibits a high amount of symmetry. For example, the control flow graph for each of the n processes (Figure 2) arises from a simple renaming of the processes (by their index i). Furthermore, the control flow graph for the spinlock (Figure 3) treats each process identically, i.e., there is a uniform choice between the spinning processes to determine which particular process is awarded the lock. We have exploited this symmetry at a basic level already in [2] by concentrating on a particular process (P_1) for the properties we analysed. Due to the symmetry between the processes, the results will be identical for the other processes.

For our model, symmetry reduction is thus a natural candidate to approach the state space explosion problem because the n processes that compete for the spinlock behave identically. The potential of symmetry reduction for reducing the state space has been extensively researched in the literature in non-probabilistic as well as in probabilistic settings, see e.g. [6, 10, 13, 11, 18, 21, 8]. The basic idea is to perform the analysis on a smaller quotient model that arises from the identification of symmetrical states in such a way that the relevant behaviour for the considered properties remains unchanged.

We will now exploit this symmetry to achieve better scalability in the number of processes for which the model can be effectively analysed.

The model checker PRISM used in our previous analysis provides a built-in implementation of a variant of symmetry reduction [18] for component symmetry, i.e., where multiple components are completely indistinguishable from each other. This approach relies on a canonicalisation of the symmetric states and is applied directly on the symbolic MTBDD representation of the model. Unfortunately, the explicit use of the process index in the spinlock module, necessary to ensure that exactly one of the waiting processes acquires the lock, prevented an application of this techniques.

Symmetry reduction using generic representatives. To apply symmetry reduction, we transformed our model following the notion of *generic representatives* as described in [11, 12, 8, 9]: We pick one arbitrary process, P_1, and treat the other $n-1$ processes as indistinguishable. This allowed us to encode the relevant information in a more succinct form by just counting how many of the $n-1$ processes are in each of their respective local states. As an example, let process P_2 be in location $ncrit_2$ with timer value $t_2 = 10$, similar to process P_3 which is in location $ncrit_3$ with timer value $t_3 = 10$, while process P_4 is in its location $crit_4$ with $t_4 = 2$. The symmetry reduced encoding just records that there are two processes in a local state with location *ncrit* and timer value $t = 10$ and that a single process is in the location *crit* with timer value $t = 2$. It is easy to see that any other permutation of the process indices $2,3$ and 4 for this configuration would result in the same state in the symmetry-reduced quotient model. The transitions in the quotient model then arise by combining the effect of the transitions for a given configuration. For example, in a configuration where all processes are in their non-critical section with timer values ≥ 0, a *tick* transition is enabled that corresponds to all processes staying in their non-critical section and decreasing their timer value by one. In the quotient model, this transition leads to the configuration where the number of processes in their non-critical section with timer value t corresponds to the number of processes in the previous configuration that are in their non-critical section with timer value $t+1$. The other possible combinations of state changes in the processes are handled in a similar fashion. The symmetry-reduced spinlock model, again a discrete time Markov chain, then results from the synchronous product of P_1 with the quotient model representing the processes P_2 to P_n and a slightly adapted version of the spinlock process that correctly handles the uniform distribution of the lock access between the waiting processes by assigning the lock to a waiting process P_1 with probability $\frac{1}{waiting}$ and with $1 - \frac{1}{waiting}$ to one of the other processes, where *waiting* is the total number of processes currently waiting for the lock.

Initial distribution. In the non-reduced model, the initial length for the non-critical section of each process is chosen according to the distribution ν, i.e., for our example distribution $\nu(40) = \nu(50) = \frac{1}{2}$ each process starts with probability $\frac{1}{2}$ with a non-critical section of length 40 and with probability $\frac{1}{2}$ with a length of 50. In the symmetry-reduced model, the initial configuration of the symmetric processes 2 to n is uniformly distributed for the generic representatives instead, i.e., for $\nu(40) = \nu(50) = \frac{1}{2}$ each assignment of x and y satisfying $x + y = n - 1$ is equally likely, where x represents the number of symmetric processes starting with length 40 and y with length 50. This modification is benign in our case, as we are interested in the probabilities in the long run and every state that occurs infinitely often is reached with probability 1 no matter the initial state. For properties that depend on the precise configuration of the initial states, we can straightforwardly weigh the initial generic representatives of the symmetry-reduced model to match the initial distribution of the original model.

Matrix generation for MRMC. We first considered an approach similar to [8, 9], where the symmetry-reduced model is generated in the PRISM input language. This approach yielded some improvement, for example for 5 processes and $\nu(40) = \nu(50) = \frac{1}{2}$ as in Table 1, the number of reachable states shrank from 198,808,720 to 8,606,543, the MTBDD size to 5,192,338, the time for building shrank from around 20 hours to about 5.5 hours and the time for calculating the steady-state probabilities was reduced from around 90 minutes to about 15 minutes. On the other hand, further increasing the number of processes again resulted in prohibitively long times needed to build the model due to the construction of the internal symbolic representation of the model. In addition to using an internal calculation engine, PRISM supports the export of the transition matrix from its internal representation of the state space. This, together with the exported state labels, allows the use of alternative model checkers such as the probabilistic model checker MRMC [15]. MRMC operates on a sparse representation of the transition matrix, i.e., an efficient, compact encoding of matrices that predominantly consist of entries with the value zero. The use of MRMC provides a speed-up for calculating the steady-state probabilities, but the bottleneck of building the internal symbolic representation by PRISM remains and blocks the successful scaling of the number of processes in the model.

We have thus implemented an approach that instead generates the reachable state space of the symmetry-reduced model in an explicit manner and directly outputs the transition matrix and state labels of the DTMC in the format usable by MRMC. While our tool is tailored to this model, i.e., given the number of processes n and the parameters γ_0, γ_1 and ν the tool enumerates the reachable state space and calculates the transition probabilities between the states, the matrix could likewise be generated by an explicit state probabilistic model checker like LiQuor [5] or even by an explicit state space generator for the PRISM input language.

Improved scalability. Table 2 demonstrates the vast improvements in scalability resulting from our approach of directly generating the matrix and using MRMC for the calculation of the steady-state probabilities. The table shows the time spent by our tool to generate the matrix and the number of reachable states which were enumerated by our tool. This number corresponds to the number of rows and columns of the generated matrix. The second-to-last column lists the time spent by MRMC for calculating the steady-state probabilities for each state and writing the results to a file. The last column presents the time

Processes	Reachable states (rows/cols in matrix)	Time for matrix generation	Time for MRMC steady state calc.	Time for properties
3	33,768	0.29 s	0.13 s	0.52 s
4	694,907	6.90 s	2.68 s	10.26 s
5	8,606,544	90.07 s	33.98 s	128.08 s
6	58,911,750	677.44 s	390.91 s	913.33 s
7	213,497,440	2,578.15 s	2,601.72 s	3,405.98 s
8	387,320,107	4,605.24 s	5,709.98 s	6,568.41 s
9	1,211,760	11.42 s	4.20 s	3.50 s
10	189,311	1.77 s	0.68 s	0.49 s
11	192,910	1.82 s	0.69 s	0.48 s
100	205,637	2.17 s	0.79 s	0.53 s
1,000	334,337	7.96 s	1.16 s	0.86 s
10,000	1,621,337	255.14 s	5.18 s	4.01 s

Table 2: Statistics for directly generating the matrix for the symmetry-reduced model and calculating the steady state probabilities for n processes and distributions $\gamma_0(5) = 1$, $\gamma_1(6) = 1$ and $\nu(40) = \nu(50) = \frac{1}{2}$ using the model checker MRMC.

spent in a post-processing step to calculate the steady-state probabilities for a number of state properties from the output of MRMC, as detailed below. We have chosen this separate post-processing step to allow for a more detailed analysis of the results and to facilitate a simple implementation of the conditional long run operators of [2] in the future. As our purpose was an initial evaluation of the potential of symmetry reduction to this and similar models, both the matrix generator and the post processing tool are not yet heavily optimised for speed and space efficiency.

As can be readily seen in Table 2, the complexity of the model for these parameters increases for up to 8 processes, with the model becoming drastically simpler afterwards. To illustrate this phenomenon and to gain further insights in the behaviour of the spinlock model when the number of processes and the distributions are varied, we have calculated the steady-state probabilities of a number of state properties using our tool and MRMC.

Figure 4 shows the effect of increasing the number of processes in our model. The figure on the left depicts the steady-state probability that at a given moment exactly k out of the n processes are in their non-critical section for our example distribution. As can be seen, increasing the number of processes at first increases the number of processes expected to be in their non-critical section because processes requesting the lock have a high probability to immediately acquire the lock and thus quickly return to the non-critical section. At some point the number of processes reaches a threshold where the lock becomes saturated. At this point, the probability that there is already a process waiting for the lock when the lock is released reaches 1. This limits the number of processes that can simultaneously be in their non-critical section to a fixed number, which is related to the ratio of the longest time a process may spend in the non-critical section and the shortest time a process needs to pass through its critical section. The figure on the right in Figure 4 details the effect of lock saturation for an increasing number of processes. For the chosen values for γ_i and ν, saturation is reached for 10 or more processes, as the probability that in the long run some process is waiting for the lock at any given moment reaches 1. This entails that each

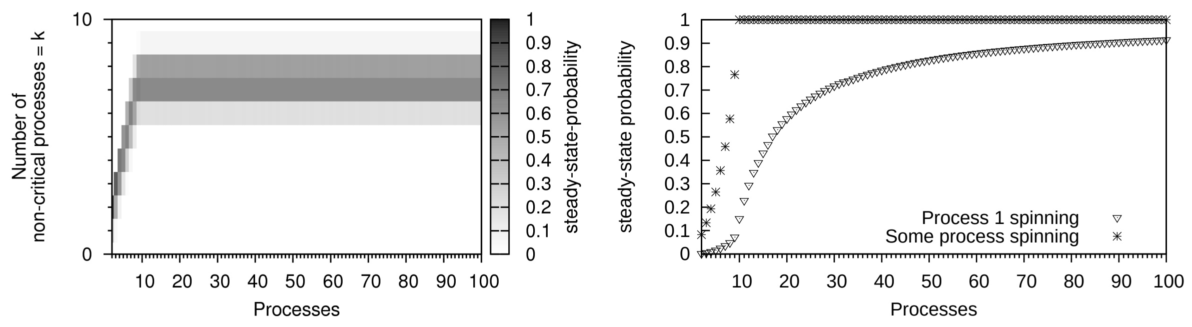
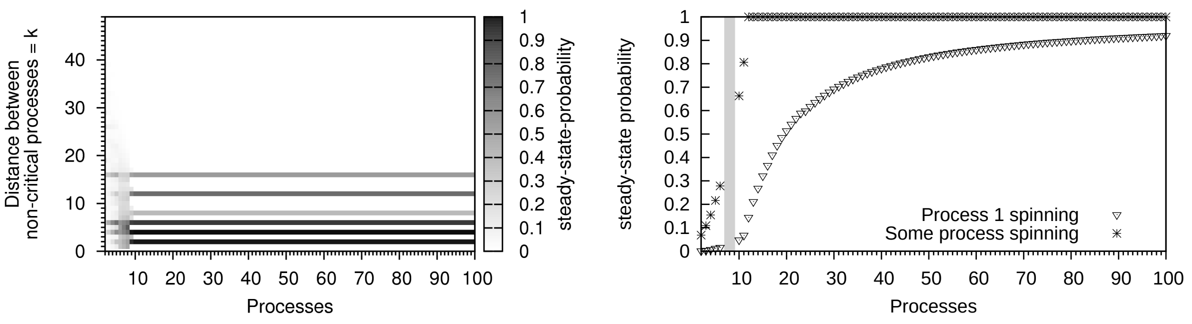


Figure 4: The left diagram shows the steady-state probabilities that at a given moment there are exactly k processes in their non-critical section. Darker grey indicates higher probability, as indicated in the legend. The right diagram shows the steady-state probabilities that at a given moment (1) process P_1 is "spinning", i.e., waiting for the lock and (2) that at least one process is "spinning". 2 to 100 processes.



(a) The steady-state probabilities that at a given moment there are two (or more) processes with distance equal to k in their non-critical section, for 2 to 100 processes and with $\gamma_0(5) = 1$, $\gamma_1(6) = 1$ and $\nu(40) = \nu(50) = \frac{1}{2}$. Darker grey represents a higher probability as indicated in the legend on the right.

(b) Steady-state probabilities that at a given moment (1) process P_1 is "spinning", i.e., waiting for the lock and (2) that at least one process is "spinning". 2 to 100 processes, $\gamma_0(5) = 1$, $\gamma_1(6) = 1$ and $\nu(50) = \nu(60) = \frac{1}{2}$. The grey bar indicates the gap, where the model exceeded the 190GB RAM limit.

Figure 5

individual process, here represented by P_1, spends a larger and larger amount of its time waiting for the lock.

Figure 5(a) provides a different view on the same phenomenon. There, we analyse the regularity of the pattern in which processes pass through their non-critical section. The figure shows the steady-state probabilities that at a given moment two (or more) processes in their non-critical section have a distance of k time units, with no other process in between. E.g., two processes in their non-critical section with timer values of 12 and 16 and no process in between have a distance of 4. As can be expected, if the number of processes is sufficiently low to ensure that the lock is not saturated, the distances between the non-critical processes take a variety of values as the probabilistic choice of the time spent in the non-critical section and the two different lengths of the critical sections "shuffle" the order and distance of the processes. Increasing the number of processes then leads to a much more regular structure, as the regularity with which processes pass through the lock induces less variety in the distances between processes.

Varying the distribution. We have seen that our approach of using symmetry reduction and the model checker MRMC allowed us to successfully scale the number of processes for the example distribution $v(40) = v(50) = \frac{1}{2}$. Unfortunately, for other, more challenging distributions considered in [2], the picture is not as rosy. Figure 5(b) shows some results for the distribution $v(50) = v(60) = \frac{1}{2}$. As can be seen, the values for $n = 7$ to 9 processes could not be calculated, as the corresponding matrix could not be generated before reaching the memory limit of 190 GB. Similar gaps where generating the matrix exceeds the available RAM exist also for other distributions we considered.

Scalability of saturated locks. The drastic improvements in scalability beyond our expectations and the collapse in complexity that we observe is linked to the complete or almost complete saturation of the lock. Increasing the number of processes further only increases the number of processes that wait spinning for the lock.

Of course, spinlocks are known to not scale well in high-contention scenarios and are therefore not widely used in situations where lock contention can become high. For practical purposes, parameters that lead to lock saturation then strongly indicate that locking realised via a spinlock is probably not the right choice and more sophisticated locking schemes or even architectural redesign of a given system is required. However, we expect that the approach presented here should apply equally well to other synchronisation primitives for highly contended resources and other scenarios such as the analysis of thread pools where by construction the majority of worker threads await further requests.

4 Conclusions and Future Work

In this paper, we have investigated how symmetry reduction helps scaling a previous case study on model checking conditional long run properties of low-level operating-system code. Using a model specific adaption of the technique of generic representatives, we were able to significantly scale the number of processes in the model beyond the point were adding more processes does not significantly increase the complexity of the analysis anymore. This allows an analysis of models with $10,000$ processes and beyond using the model checker MRMC. There are still situations where the analysis proves elusive for certain number of processes due to the high number of reachable states.

Future work. The results presented in this paper represent work in progress and allowed us to evaluate the potential benefit of using symmetry reduction techniques for the scalability in the number of processes of a probabilistic analysis of locking mechanisms and similar low-level constructs. We plan to further improve the time and space efficiency of the tool we use to generate the state space. We also plan to investigate refined notions of symmetry reduction, the compatibility with symbolic methods, i.e., the encoding of the states and corresponding variable orders, as well as other reduction techniques such as bisimulation quotienting/lumping [14, 15]. As our results show the feasibility of an analysis approach using symmetry reduction and the model checker MRMC, we further plan to extend MRMC by the capability to calculate conditional long-run properties of the type considered in [2].

A second topic for future work are more complicated case studies. Interesting candidates are thread pools and queue-based locks [1, 20] as they enforce a strong dependency between waiting processes. Particularly interesting due to their potential feedback on the design and use of these algorithms are reactive locking schemes [19]. Probabilistic model checking results could guide when the lock switches to an alternative implementation. Overcoming the complexity gap remains a top priority.

References

[1] Thomas E. Anderson (1990): *The Performance of Spin Lock Alternatives for Shared-Money Multiprocessors*. *IEEE Transactions on Parallel and Distributed Systems* 1(1), pp. 6–16, doi:`10.1109/71.80120`.

[2] Christel Baier, Marcus Daum, Benjamin Engel, Hermann Härtig, Joachim Klein, Sascha Klüppelholz, Steffen Märcker, Hendrik Tews & Marcus Völp (2012): *Waiting for Locks: How Long Does It Usually Take?* In: *Formal Methods for Industrial Critical Systems - 17th International Workshop (FMICS'12)*, *Lecture Notes in Computer Science* 7437, Springer, pp. 47–62, doi:`10.1007/978-3-642-32469-7_4`. Extended version available at `http://wwwtcs.inf.tu-dresden.de/ALGI/spinlock-FMICS2012.pdf`.

[3] Guillem Bernat, Antoine Colin & Stefan M. Petters (2002): *WCET Analysis of Probabilistic Hard Real-Time Systems*. In: *Proceedings of the 23rd Real-Time Systems Symposium (RTSS'02)*, IEEE Computer Society, pp. 279–288, doi:`10.1109/REAL.2002.1181582`.

[4] David Bernick, Bill Bruckert, Paul Del Vigna, David Garcia, Robert Jardine, Jim Klecka & Jim Smullen (2005): *NonStop: Advanced Architecture*. In: *Dependable Systems and Networks (DSN'05)*, IEEE Computer Society, pp. 12–21, doi:`10.1109/DSN.2005.70`.

[5] Frank Ciesinski & Christel Baier (2006): *LiQuor: A tool for Qualitative and Quantitative Linear Time analysis of Reactive Systems*. In: *Third International Conference on the Quantitative Evaluation of Systems (QEST'06)*, IEEE Computer Society, pp. 131–132, doi:`10.1109/QEST.2006.25`.

[6] Edmund M. Clarke, Somesh Jha, Reinhard Enders & Thomas Filkorn (1996): *Exploiting Symmetry in Temporal Logic Model Checking*. *Formal Methods in System Design* 9(1-2), pp. 77–104, doi:`10.1007/BF00625969`.

[7] Björn Döbel & Hermann Härtig (2012): *Who Watches the Watchmen? Protecting Operating System Reliability Mechanisms*. In: *8th Workshop on Hot Topics in System Dependability (HotDep'12)*. Available at `https://www.usenix.org/system/files/conference/hotdep12/hotdep12-final1.pdf`.

[8] Alastair Donaldson & Alice Miller (2006): *Symmetry Reduction for Probabilistic Model Checking Using Generic Representatives*. In: *Automated Technology for Verification and Analysis, 4th International Symposium (ATVA'06)*, *Lecture Notes in Computer Science* 4218, Springer, pp. 9–23, doi:`10.1007/11901914_4`.

[9] Alastair Donaldson, Alice Miller & David Parker (2009): *Language-Level Symmetry Reduction for Probabilistic Model Checking*. In: *6th International Conference on the Quantitative Evaluation of Systems (QEST'06)*, IEEE Computer Society, pp. 289–298, doi:`10.1109/QEST.2009.21`.

[10] E. Allen Emerson & A. Prasad Sistla (1996): *Symmetry and Model Checking*. *Formal Methods in System Design* 9(1-2), pp. 105–131, doi:`10.1007/BF00625970`.

[11] E. Allen Emerson & Richard J. Trefler (1999): *From Asymmetry to Full Symmetry: New Techniques for Symmetry Reduction in Model Checking*. In: *Correct Hardware Design and Verification Methods (CHARME'99)*, *Lecture Notes in Computer Science* 1703, Springer, pp. 142–156, doi:`10.1007/3-540-48153-2_12`.

[12] E. Allen Emerson & Thomas Wahl (2003): *On Combining Symmetry Reduction and Symbolic Representation for Efficient Model Checking*. In: *Correct Hardware Design and Verification Methods (CHARME'03)*, *Lecture Notes in Computer Science* 2860, Springer, pp. 216–230, doi:`10.1007/978-3-540-39724-3_20`.

[13] C. Norris Ip & David L. Dill (1996): *Better Verification Through Symmetry*. *Formal Methods in System Design* 9(1-2), pp. 41–75, doi:`10.1007/BF00625968`.

[14] Joost-Pieter Katoen, Tim Kemna, Ivan Zapreev & David N. Jansen (2007): *Bisimulation Minimisation Mostly Speeds Up Probabilistic Model Checking*. In: *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference (TACAS'07)*, *Lecture Notes in Computer Science* 4424, Springer, pp. 87–101, doi:`10.1007/978-3-540-71209-1_9`.

[15] Joost-Pieter Katoen, Ivan S. Zapreev, Ernst Moritz Hahn, Holger Hermanns & David N. Jansen (2011): *The ins and outs of the probabilistic model checker MRMC*. *Performance Evaluation* 68(2), pp. 90–104, doi:`10.1016/j.peva.2010.04.001`.

[16] Steffen Knapp & Wolfgang Paul (2007): *Realistic Worst-Case Execution Time Analysis in the Context of Pervasive System Verification*. In: *Program Analysis and Compilation*, *Lecture Notes in Computer Science* 4444, Springer, pp. 53–81, doi:`10.1007/978-3-540-71322-7_3`.

[17] Marta Kwiatkowska, Gethin Norman & David Parker (2004): *Probabilistic symbolic model checking with PRISM: a hybrid approach*. *International Journal on Software Tools for Technology Transfer (STTT)* 6(2), pp. 128–142, doi:`10.1007/s10009-004-0140-2`.

[18] Marta Kwiatkowska, Gethin Norman & David Parker (2006): *Symmetry Reduction for Probabilistic Model Checking*. In: *Computer Aided Verification, 18th International Conference (CAV'06)*, *Lecture Notes in Computer Science* 4144, Springer, pp. 234–248, doi:`10.1007/11817963_23`.

[19] Beng-Hong Lim & Anant Agarwal (1994): *Reactive Synchronization Algorithms for Multiprocessors*. In: *ASPLOS-VI*, ACM, pp. 25–35, doi:`10.1145/195473.195490`.

[20] John M. Mellor-Crummey & Michael L. Scott (1991): *Scalable reader-writer synchronization for shared-memory multiprocessors*. In: *3rd Symposium on Principles and Practice of Parallel Programming (PPOPP'91)*, ACM, pp. 106–113, doi:`10.1145/109625.109637`.

[21] Alice Miller, Alastair Donaldson & Muffy Calder (2006): *Symmetry in Temporal Logic Model Checking*. *ACM Computing Surveys* 38(3), doi:`10.1145/1132960.1132962`.

[22] Michael M. Swift, Brian N. Bershad & Henry M. Levy (2005): *Improving the Reliability of Commodity Operating Systems*. *ACM Transactions on Computer Systems* 23(1), pp. 77–110, doi:`10.1145/1047915.1047919`.

[23] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat & Per Stenström (2008): *The Worst-Case Execution-Time Problem – Overview of Methods and Survey of Tools*. *ACM Transactions on Embedded Computing Systems* 7(3), pp. 36:1–36:53, doi:`10.1145/1347375.1347389`.